DMSwitcher: Boosting Data Access Performance with Multi-backend Disaggregated Memory

ABSTRACT

Dynamic resource mapping between computing resources and memory resources remains a great challenge in composable disaggregated architecture. Although previous works allow applications to access different far memory components, they still face a "swap wall" problem. Existing single-backend swap mechanism becomes the performance and utilization bottleneck since it relies on a shared LRU queue. Monolithic server resources equipped with heterogeneous far memory backends are underutilized due to the lack of multi-way far memory access.

To break the swap wall and support adaptive resource mapping, we take the first step to build a disaggregated memory management strategy for multiple storage backends. We propose DMSwitcher to scale out heterogeneous disaggregated memory access channels in virtualization environments by providing efficient backend switching supports. Specifically, DMSwitcher features a well-crafted program analyzer for fine-grained page behavior tracing and application profiling. With a deep understanding of the program, it supports a smart backend selection strategy based on application-system co-design. Finally, it employs a high-performance backend parameter adaptor to better configure and adjust hardware parameters on far memory backends. We implement our design and compare it with state-of-the-art far memory platforms. DMSwitcher shows up to 2.1x speedup of swap performance and it can save up to 5.1x memory space within the same time span of evaluated workloads.

Keywords: far memory, multi-backend, switcher, swap

1. INTRODUCTION

From the view of data center, dynamic resource mapping between heterogeneous disaggregated computing resources and memory resources is always a challenge in composable disaggregated architecture [30,32,33,35,37]. Recent network research has opened up multiple far memory (FM) access channels, as Figure 1 shows. They build larger memory resource pools [7,25,26,34] and additional memory devices including non-volatile persistent memory [20, 39], storage class memory (SCM) [22], solid-state drive (SSD) [25] on local servers. They also share memory resources among servers through high-speed networks like RDMA [2,16,31, 43], CXL [9,19,27], OpenCAPI [8,10,35], smart-NICs [6, 18], and smart switches [26], etc. Dynamic connection and adaption of heterogeneous disaggregated memory backends spatially and temporally is essential.

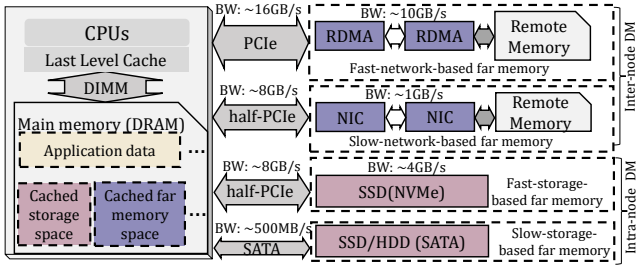


Figure 1: Multi-backend far memory channels.

By offloading part of the cold memory pages to far memory, the software tries to retain more critical data locally for better performance on per bit of memory (PPB). In the early years, works utilize user-defined remote memory access APIs to program the process and network communication [1,4,13,40]. Some recent works [36, 42] build simpler object-oriented APIs with new data structures for higher performance. In addition, some application-specific frameworks [24, 44, 47] are proposed for higher performance. For transparent far memory access, plenty of works utilize swap mechanism to replace the disk-based swap backend to SSD and RDMA-based far memory as swap backends [2, 3, 16, 25, 48]. In summary, existing memory disaggregation works focus on single-end far memory expansion, which lacks the way of far memory backend switch for resource mapping dynamicity.

From the perspective of the operating system, the performance bottleneck lies on the single-backend swap mechanism, which we name it *"swap wall"*. The system cannot build separated swap channels transparently, because they must use the same swap Frontend [44]. Since all applications share a single swap partition, every swap-out operation has a lock of swap entry allocation, which significantly limits the overall throughput [43]. Some works [3, 31, 48] try to solve the swap wall problem by providing a multi-layer swap scheme in virtualized system. Canvas [43] adopt swap isolation and allocation for multiple applications to increase page cache hit rate. The most related works [7, 46] allow the system to adapt to a different far memory backend. However, they fail to support multiplexing far memory configuration on the host system so that each VM can not access the far memory backend independently. The whole system can not access multiple far memory backends at the same time.

To concurrently access far memory in parallel, this paper breaks the swap wall and scales up the heterogeneous disaggregated memory access channels. We propose DMSwitcher

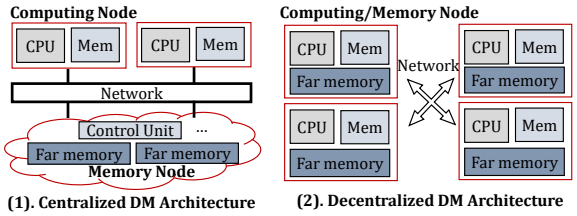


Figure 2: Centralized and decentralized DM architecture.

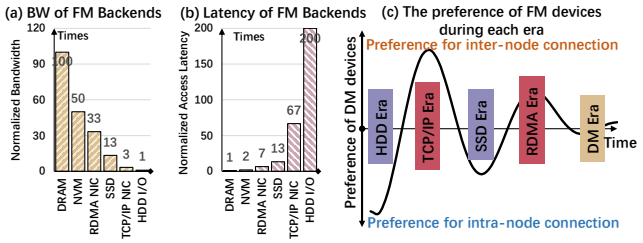


Figure 3: Performance and preference comparison of different far memory backends.

system that injects smart switchers into each virtual machine to access multiple far memory backends in VM-level parallel. Our design addresses the following challenges:

Dynamic switching. Existing systems still lack the dynamic management of switchable far memory backends. Some works [7, 46] adopt a straightforward approach with a large switching granularity. The system must reboot OS to change the far memory backend so that the backend switching costs of these systems are very high. Application-level distributed shared memory works [15, 29] on multiple backends uses a small switching granularity, which brings a lot of memory overhead and runtime overhead due to fault-tolerant data backup and consistency handling. A smart switcher needs to support high-performance and on-demand backend switching with low overhead and high flexibility.

Application awareness. Existing general-purpose far memory systems do not yet consider fine-grained awareness of application behavior. Some work is limited to proprietary applications such as computing and neural networks, but works in non-transparent application levels is hard to migrate to general applications. Existing disaggregated memory scheduling systems typically allocate resources to applications by detecting current resource pressure [23, 25], without application behavior analysis. Related works [2, 45] analyze the application sensitiveness on far memory platform and allocate resources under QoS. However, they lack hardware information analysis to fully extract the application behavior.

Automated configuration. Existing work is not efficient enough to support automatic configuration of far memory. Far memory perfromance is associated with the multi-dimensional far memory parameters, and there are many turning knobs that have not been considered yet. Existing work [2, 44, 45, 46] mainly adjusts the far memory ratio and memory size allocated for tasks. TMO [46] proposes adaptive allocation strategies based on heterogeneous backends. Fargraph [44] takes the size of transferred data chunk into account. This inspires us to consider various parameters including page size, page type buffer sizes, and data transfer channels, etc. to generate optimal far memory configurations.

Our contribution is listed as follows.

* We propose a FM switcher that enables multi-backend inter-node and intra-node far memory access in parallel.
* The program page analyzer features fine-grained program behavior for accurate FM backend selection.
* The high-performance resource adaptor generates optimal parameters of heterogeneous FM backends.
* We show up to 2.1x speedup of swap performance and save up to 5.1x memory space within the same time span compared with state-of-the-art baselines.

The remainder of this paper is organized as follows. Section 2 give backgrounds and motivates our design. Section 3.1 gives the overview of system. Section 3.2, 3.3 and 3.4 describes our system design, including the program page analyzer, far memory switcher, and backend parameter adaptor. Section 3.5 shows the system workflow. Section 4 presents the experimental results. Section 5 discusses related works and Section 6 concludes this paper.

2. BACKGROUND AND MOTIVATION

In this section we present the background and challenges that motivate us to design a smart and parallel multi-backend far memory system.

2.1 DM Architectures & Preference

Centralized and decentralized DM architecture. To the best of our knowledge, there are two types of disaggregated memory (DM) architecture, which we named centralized DM architecture and decentralized DM architecture as shown in Figure 2. In general, DM architecture consists of computing nodes and memory nodes [30]. The centralized DM architecture [14, 34] maintains a central large memory pool with multiple memory nodes to provide far memory resources to each computing node. In decentralized DM architecture [7, 26, 35], the memory resource is distributed physically and shared with each other virtually. The centralized DM architecture is easy for resource management and low-cost memory updating, while the decentralized architecture performs better for resource sharing and cost efficiency.

Inter-node and intra-node FM preference. Memory capacity can be expanded in two ways: inter-node extension [1,2,6,13,16,31,36,40,44] named inter-node far memory (Inter-FM) and intra-node expansion [3, 7, 22, 25, 39, 46, 48] named intra-node far memory (Intra-FM). Inter-FM connects two server nodes through network, whose performance is determained by PCIe and network. Intra-FM uses storage devices on the local server based on the I/O protocols support. Figure 3-(b) and (c) gives the detailed difference of memory bandwidth and read/write latency of FM backends. Nowadays, network achieves bandwidth with 1GB/s to 20GB/s while SSD bandwidth is 0.5GB/s to 7GB/s. Furthermore, new fabric-attached NVlink [11], CXL [9] technologies achieve higher performance but are underdeveloped yet. We summarize the preference changes in each eras, as shown in Figure 3-(a). The performance of Inter-FM and Intra-FM is getting closer so combining Inter-FM and Intra-FM would provide better performance per bit.

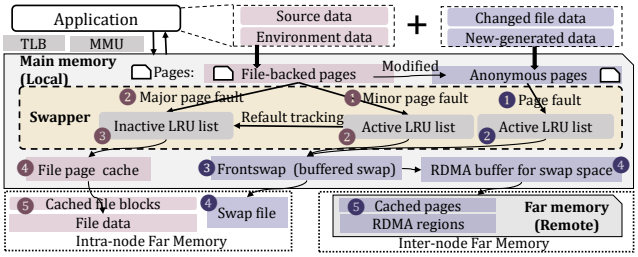


Figure 4: Memory reclaim and page fault procedure in kernel level of Linux operating system.

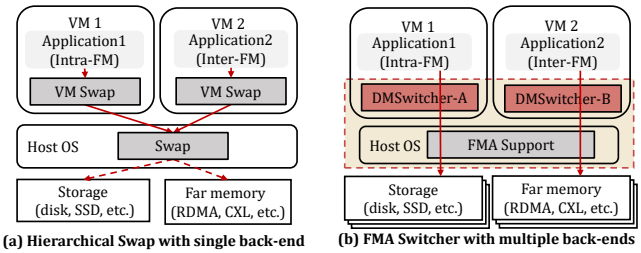


Figure 5: System with single and multiple FM backends.

2.2 Swap Wall

The existing transparent far memory environment relies on the original swap mechanism in Linux OS, which faces performance overhead and scalable deployment problems.

Swap becomes the performance bottleneck of parallel far memory access. The current memory hierarchy can not fit well into today's DM architecture. Usually, the system caches file-backed pages in the page cache, i.e. a specific space in local memory. Multiple processes use a shared global Least-Recent Used (LRU) list and a shared swap cache in the system to buffer the offloaded data [7, 43]. The page data is managed together without isolation thus the data access bandwidth is hard to control and manage. Moreover, swap requests from different applications go through the same shared data path, This usually causes severe performance interference, such as 6.4x slowdown [43], which is unacceptable for QoS-guaranteed applications.

It is non-trival to adapt to multiple far memory backends based on original swap mechanism in a scalable way. First, due to the limit of current swap mechanism, the existing system can only support one type of swap backend at the same time. If one needs to use another far memory, one need to reboot the machine and reconfigure the swap kernel. Figure 4 presents the memory reclaim and page fault handling procedure. The Frontswap provides a "transcendent memory" interface for swap pages to cache the page in DRAM instead of a swap disk. The existing works [2, 16] support the backend replacement by unloading the swap-based far memory access module and loading the RDMA-based far memory access module, along with the buffered region for caching data from RDMA. Different far memory backends serve the only Frontswap to support faulted pages handling, page mapping and page entry caching, which is unscalable to process pages with different locations from different far memory backends simultaneously.

2.3 Far Memory Virtualization

Virtualized far memory. To utilize the inter- and intra-far memory synchronously and efficiently, virtualization technique is involved. Virtualization gives the chance to switch the far memory backend without physical machine shutdown. Furthermore, virtualization allows the system to start concurrent and shared-channel far memory access on different virtual machines on the same hardware architecture. Furthermore, it can help to deal with the isolation of swap space and page fault handling. Designing a smart switcher based on swap mechanism and virtualized environment can parallelize far memory access on physical servers.

Scalable far memory environment. Scalable far memory system in virtualization environment is necessary but requires sophisticated design. There is no hybrid far memory management layer between virtual machines and host OS. As Figure 5 shows, related works design hierarchical swap architecture [7,31,48] and swap between VM swap space and host OS swap space, which then swap data with far memory space. Oftentimes, we use Cgroup, namespace, or zone to limit the memory usage, network channel, and swap space of the processes or threads. However, Cgroup maintains virtual memory limitation and triggers page fault handling with several LRU list for each process adn a shared global LRU list. This causes the current virtual machine lacks the isolation of cache and swap space.

In summary, designing a smart switcher in virtualized environments is the right way to break the single-channel swap wall. Carefully configuring the turning knobs of far memory backends helps to achieve higher performance and efficiency.

3. DMSWITCHER SYSTEM DESIGN

3.1 System Overview

Towards high application performance and system efficiency, we propose DMSwitcher to scale out the heterogeneous disaggregated memory access channel in virtualization environment by backend switching supports. We implement a multi-backend far memory system that supports parallel backends with multiplexed data transmission based on program and system information co-analysis. Our system consists of three parts, as Figure 6 shows. 1) We design a fine-grained *program page analyzer* based on page behavior tracing and application profiling information for higher efficiency. 2) We design a low-overhead *far memory switcher* with high-performance direct-connected far memory backends to switch proper far memory backend for each application intelligently and flexibly. 3) We also build a high-performance *backend parameter adaptor* to better configure and adjust hardware parameters on far memory backends with several turning knobs to fully utilize overall resources.

3.2 Program Page Analyzer

This section traces the page behaviors and analyzes the relationships with far memory backends for high efficiency.

3.2.1 Page Type

There are mainly two types of pages for memory management, including file-backed pages and anonymous pages.

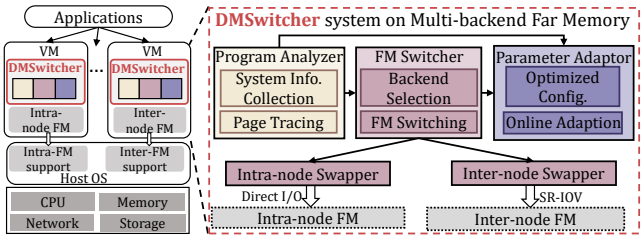


Figure 6: The overview of DMSwitcher system.

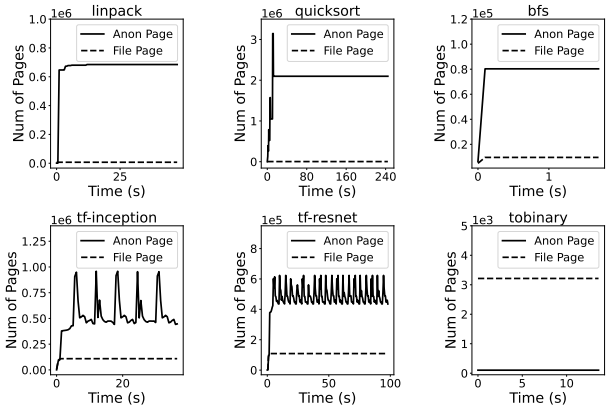


Figure 7: Number of anonymous pages and file-backed pages during execution on different workloads.

File-backed pages represent the pages that read from the file system, such as to-be-used source datasets, environment library data, configuration files, and so on. Anonymous pages mainly consist of the changed file-backed pages to write back to files and the newly generated data which has no data backup. As Figure 4 shows, if the memory is inadequate, the system will evict file-backed pages to block devices and fetches them back when needed through storage I/O access. Since there is no backup for anonymous pages, the system allocates a specific file named swap file that stores in local block devices as the spaces for evicted pages.

We measure the two type of pages by recording the memory access times of anonymous pages and file-backed pages in the example workloads, as shown in Figure 7. Detailed workload description refers to Table 3. We use the probe sampling method with the `pmap <pid> -d` tool to obtain the page information in memory periodically and extract the page type and access number. The application examples show different type-based page distributions spatially and temporally. The linpack and quicksort contain large amounts of anonymous pages and no file-backed pages. The tf-inception and tf-resnet have obvious file-backed pages due to the model file read and periodically accessed anonymous pages due to the batch inference.

Since the memory occupation of file-backed pages and anonymous pages is diverse in different applications, one may have different memory offloading preferences. We observe that the distribution of pages with different type affects the deployment preference on different far memory backends. Applications with large parts of file-backed pages prefer to

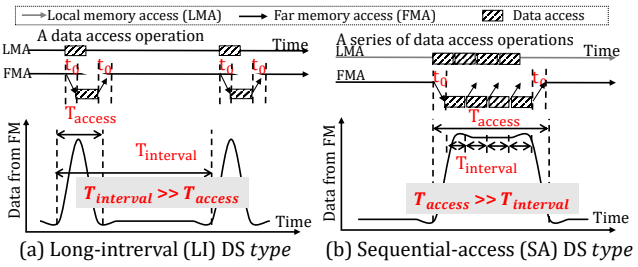


Figure 8: The two conditions of far memory access.

use vertical far memory like SSD-based far memory space. Thus, we classify the application into three groups, the low, medium and high AP-intensive groups according to the low, medium and high proportions of anonymous pages. We then label each application by the specific page access phases.

3.2.2 *Page Hotness*

We measure the changes of page hotness and coldness in the applications. We use `lackey` provided by `valgrind` [12] to get the address of each memory access and filtered out the statistics of the pages with a high number of accesses as shown in Figure 9, which shows the distribution of page accesses heatmap in the application runtime. There are obvious data segments that consist of pages with sequential addresses. We find hot data segments such as 3095, 3588 page ID in linpack and 1025, 20988 page ID in quicksort.

Practically, we summarize that there are two kinds of *representative data segments (DS)* in each applications, which we name them long-interval (LI) type DS such as tailed cold pages as Figure 8 (a) shows and sequential-access (SA) type DS such as read-once data chunks as Figure 8 (b) shows. The overhead for each memory operation comes from the latency difference between the far memory access latency and local memory access latency, which is t_0 in the Figure 8. Suppose that we have the interval time between two DS operations $T_{interval}$ and the DS access time T_{access}. We classify the data segment as LI type when $T_{interval} >> T_{access}$, while the data segment is classified as SA type when $T_{access} >> T_{interval}$.

LI data segments are more friendly to far memory access since the duration of memory access for LI data segments is much shorter than other data segments. SA data segments are important to data offloading strategies, especially data segments SA_i with bar shapes. The performance of programs changes greatly if we detect and offload the frequent-accessed SA data segments to far memory. We build a list of SA data segments $LI = \{SA_1, SA_2, ..., SA_n\}$. For each SA_i, we record the data size and the memory access frequency level from low to high. We give a threshold and we add the data size of SA data segments with lower level than the threshold to show the overall data segment size to be offloaded. The SA data segments are detected by profiling and redirected to the data access operation in the original program. By tracing these, we can estimate the overall hot page ratio so that instruct the far memory allocation operations. The page tracing also inspires us to add probes for better data offloading in the compile layer, which we leave this part in our future works.

3.2.3 *Page Access Stages*

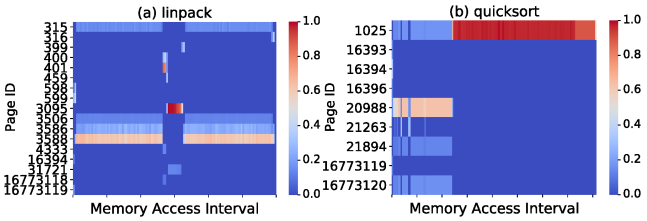


Figure 9: Heat map of the number of page hits in the runtime of different workloads

As Figures 7 and 9 show, the programs feature obvious page access phases. Complex applications are combinations of visible phases. In the data preparation stage, the application requests memory and the number of anonymous pages increases. During the execution stage, the total amount of memory owned by the application does not change significantly. When separating the application into different stages by page behavior, the granularity should be carefully considered, since too fine-grained page fault detection may cause additional overhead on the original program. we often use preparation and execution stages for each program to reduce the overhead of stage-based far memory adaption.

Furthermore, on different far memory backends, the same page faults number acts different latency growing due to the remote access latency difference. Given a certain duration (such as 10s), we collect the page fault number for online stage separation. The overall performance can be predicted by the distribution of local pages and remote pages. We can estimate the latency growing for each far memory backend according to the profiled information, which shows the latency difference of remote memory access performance on multiple far memory backends. We can capture the application's remote page access behavior and estimate the performance changes on different far memory backends.

3.3 Far Memory Switcher

We design a swap switcher for flexible backend selection and far memory configuration.

3.3.1 Configurable Backends

Backend injection. The swap switcher can perform differently in each virtual machine and enable various and multi-way far memory access. We inject the multi-backend far memory access into the system swap modules. In the system kernel, we modified the swap front-end and inject several configurable and switchable swap backends into the frontend. First, we prepare the corresponding far memory access interfaces that can fit into the page access patterns in the current operating system. Then, we allow the frontswap to connect with certain far memory backend and activate the corresponding software and hardware configuration. If the current virtual machine needs to change the current far memory backend, we break the connection between frontswap and far memory backend and reload other far memory backends. In our implementation, the swap switcher can be triggered by both the virtual machines themselves and the hypervisor. We keep a daemon process in each virtual machine and the corresponding server process in the hypervisor for overall adaption, which we describe in the next subsection in detail. Once we switch to one backend, the virtual machine can rightly access the corresponding far memory space.

Backend configuration. On *Inter-FM backend*, applications can subscribe memory resources on a remote node through high-speed networks or specific fabrics like RDMA [2], CXL [9], OpenCAPI [10], smart-NICs [18], etc. Inter-FM utilizes limited local memory as far memory buffer and use network resources for remote data access. Inter-FM software environment builds on far memory access interfaces that support diverse network protocols and cooperate with our backend injection module. The advantage of Inter-FM is the high-bandwidth data transferring and fast I/O access performance. So far, Inter-FM is deemed faster than Intra-FM [2, 7, 16]. The *Intra-FM backend* taps into storage-like persistent memory [39] and solid state drive (SSD) [25, 46] etc and generally offers larger capacity with lower cost. Intra-FM utilizes memory and I/O device resources. Intra-FM software environment builds on I/O read and write operations that support diverse file systems in the operating system. Although the total bandwidth and access speed is worse than Inter-FM, the cost of Intra-FM devices is much lower due to the price per bit of storage.

3.3.2 Backend Selection Strategy

We design a smart switcher algorithm for selecting backends better in each virtual machine by analyzing the page behavior of the applications inside and the resource behavior of the system outside. We use offline prediction and online resource pressure to make decisions for each running VMs.

Backend selection strategy. Choosing a proper backend is important due to performance and efficiency requirements. For the same application, different far memory backends bring various performances due to the variety of the end-to-end data access latency, the chunk size, the direct read/write support, memory bandwidth, etc. We propose an algorithm to decide the proper backends for each application. The inputs of this algorithm are the profiled information of applications, the hardware parameter reading from the system and the given limitations. The profiled information includes the page type list (in Section 3.2.1), the DS list by analyzing page hotness (in Section 3.2.2), the page access stage by tracing the page faults, and the buffer usage (in Section 3.4.1). The output of this algorithm is the far memory backend decision, the hot data size that should be resisted locally under the given limitation, and the change stage of the hot data size over execution time. The given limitations include the upper bound of end-to-end latency described as server-level objective (SLO) and the system resource pressure including the available CPU core, memory size, I/O bandwidth, RDMA bandwidth, RDMA channel. etc. We build a design tree with branches as Figure 10 shows. Besides, we use the linear regression model to estimate the outputs of this algorithm. Generally, we train specific models for different applications so that the latency can be quickly inferred.

Application page type analysis. Applications prefer different far memory backends due to page access behavior in local memory. Linux operating system divides pages into two types according to the swapping direction, the file-backed pages and anonymous pages as shown in Figure 11-(b). For memory-hungry applications, some rely on large-scale source

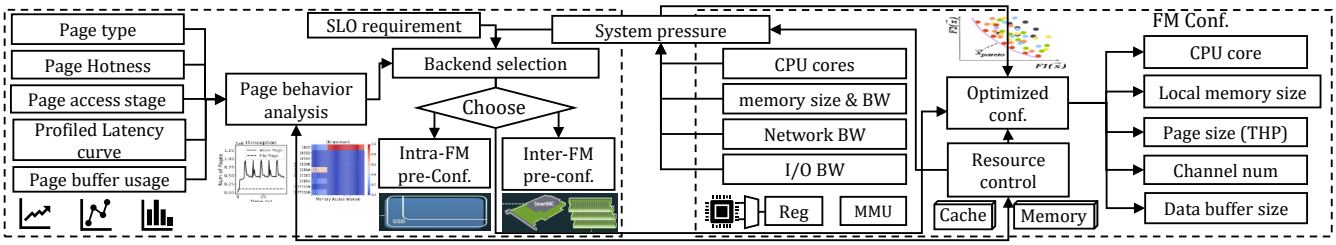


Figure 10: The backend selection strategy and optimized configuration in DMSwitcher system.

data during calculation thus accessing file-backed pages very often. The other applications prefer to update their working set frequently and generalize many anonymous pages. The size of file-backed page size reflects the preference for intra-nod memory access while the anonymous page size is the preference for inter-nod memory access.

Page buffer usage analysis. Applications often use buffers to cache pages into the main memory from slow storage for high performance. Page buffer is large since the system will load all the file data into local memory initially. If we use Cgroup to limit the memory of each progress, the process will quickly stop since the buffered file-backed pages are cleared immediately. Fortunately, if the progress use VFS (virtual file system) in Linux operating system and call `read()` and `write()` , the page buffer size can be adjusted dynamically by Cgroup, while the page buffer size can not be adjusted dynamically by group if the progress use pointer address mapping between memory and call `mmap()` to cache pages. Thus in this work, we label the processes as buffer-friendly by identifying the storage I/O operations to control page buffer size without program failure.

Backend switching modes. There are two modes to switch the backends, the periodical mode and the event-driven mode. The *time-based periodical adaption mode* is straightforward, that is to switch periodically with a fixed interval. Generally, we use the time duration which is the integer multiple of the total execution time of applications to reduce program live migration overhead. To efficiently switch with low overhead of configuration and higher performance of applications, we also propose the *event-driven iterative adaption node.* The backend parameter adaptor responses for the overall resource detection, reclaim, synchronize, and allocation in physical. It reclaims the resource of each VM and provides new resource conditions to the hypervisor scheduler. It also synchronizes each resource and handles the resource conflict in the multi-backend far memory environment. during VM running, we add or limit a unit of local memory size if the system detects stage changes, such as page faults changes larger than a certain threshold. Specifically, if memory bandwidth is free enough, we use more local memory as much as possible. If we detect the available memory bandwidth is small and the current application is stalled, we should add far memory resources for saving local memory usage as well as start far memory access. 2)If the storage I/O or network bandwidth is busy, we should switch the data offloading backends even if that backend is sub-optimal.

3.4 Backend Parameter Adaptor

We design the Backend Parameter Adaptor to smartly swap data with optimized far memory access configuration.

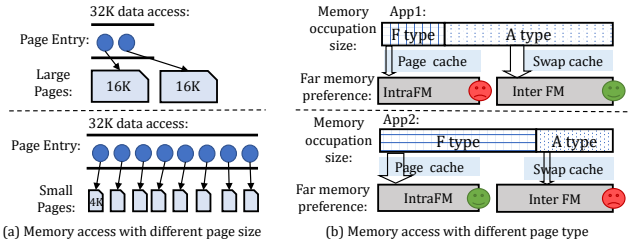
3.4.1 *Adaptive Local Memory Ratio*

For general applications, the turning knob is the local memory limitation, which determines how much parts of data can be retained in local memory thus the cold data can be offloaded to far memory backends. We use memory limitation tools cgroup to limit local memory use and thus swap the offloaded parts to far memory. Local memory parts consist of the original resistant memory of programs and the buffer space needed by far memory access. First, we estimate the hot data size according to the program with specific algorithms and datasets. Basically, the LRU queues in the original swap mechanism help to offload the least-recently-used data to far memory. We build offline trends of runtimes on different far memory ratios of various algorithms and datasets as offline information. Based on the profiled information, we can choose the most high-performance or efficient data distribution for each application. We try to find the associations of the profiled data by building the fitting curves of runtime growth and programs. We also find that the trends of the performance vary between different algorithms but the runtime growth often has a linear relationship with dataset size.

Second, we also estimate a proper far memory buffer size to optimize data buffering. When offloading to and fetching data from far memory, an optimized buffer size is important for improving cache hit success and hiding far memory access latency. For different far memory backends, the buffer setting is totally different. To make full use of the buffer size as well as gain high performance, we collect the memory access latency on various buffer size and configure buffers in far memory access progress. We give higher buffer size to the programs with higher parallelism since high-parallelism applications have more sequential memory access and can benefits more from remote memory access.

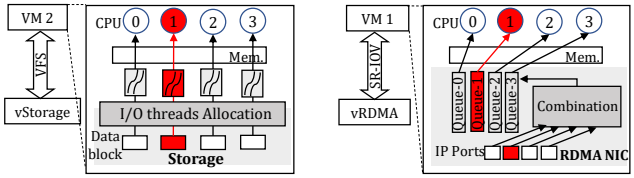
3.4.2 *Adaptive Transparent Large Page*

Page size is a significant parameter that determines the execution performance on far memory environments in our system. Page size is the memory access units that applications use in the system, as shown in Figure 11-(a). The system usually adopts fixed-length contiguous pages (memory access unit) of virtual memory with a single entry in the page table. Recently, the page size is 4KB by default in commercial servers. Large pages may improve memory access performance due to less overhead on page table entries searching on larger TLB coverage [17] and high data



(a) Memory access with different page size (b) Memory access with different page type

Figure 11: Application behavior on different page size and page type brings different preference and performance on far memory.



(a) Intra-node FM channel configuration (b) Inter-node FM channel configuration

Figure 12: An example of far memory access channel configuration on Intra-FM and Inter-FM.

access bandwidth [17, 41]. Small pages may reduce latency and memory occupation by reducing overall duplicated data size and less redundant data access [6, 36]. However, if the program requires a small amount of data, larger pages bring more redundant data wasted when fetching each pages. Since page sharing is very common among VMs, data duplication method is proposed to reduce redundancy and save memory space. Unfortunately, large pages reduce the deduplication opportunities, thus there exists a trade-off between access performance and deduplication rate.

Processor designs often allow larger page sizes due to its benefits, such as adopting transparent large pages (THP) up to 2GB on each page. THP(Transparent huge page) is a mechanism provided by Linux that allows applications to use the huge page during runtime without explicit huge page configuration and use. There are several points that can factor into choosing the best page size, including page table size, TLB usage, internal memory fragmentation, disk access and remote memory access. By analyzing the frequently-accessed page address, we can characterize the large page preference of applications according to the proportion of continuous page address access among the total pages. We label the applications that have better runtime performance when THP is on as THP-friendly programs, while we label the applications that have worse performance when THP is on as non-THP-friendly programs. These labels give specific instructions on whether THP should be turned on or not.

3.4.3 Adaptive Data Transferring Channel

During data communication, the width and length of the far memory access channel are the most important turn knobs. The data transferring width determines the arrived data size in unit time. Usually, larger width and length provide more arrived data in unit time, however, they can also waste the I/O resource since there may be small data transferred very often. Shared resources or changeable data channels is an available way. Also, message-based data transferring benefits from small data chunks and data-based transferring benefits from large data chunks. On storage-based far memory, the width can be adjusted by configuring the block size. On network-based far memory, the width can be configured by the network bandwidth or channel numbers.

Different performance of inter-node far memory environment mainly comes from the network configuration, including NIC devices adjustment and runtime deployment. Figure 12 shows the latency and bandwidth under different network card and far memory runtime. With TCP/IP NIC and RDMA NIC devices, The data transfer throughput various and generally RDMA can be almost 10x faster than TCP/Ip NIC. With different number of queue binding, network faces higher bandwidth with more queue binded,up to binding all CPU cores to the queue. With frame size changing from cache line level(64K) to page granularity (4K), The memory access latency also changes. Oftentimes, Inter-FM has customized bandwidth by binding different CPU cores to the queue of network transfer, while Intra-FM has adaptive bandwidth allocated passively by OS calls with inescapable CPU stall. We can manage network bandwidth and throughput by changing the network queue number, which is binded with CPU core, and the transferring frame size.

Algorithm 1: Algorithm of Far Memory Switching

Input: application set A, online VM set OVs, free VM set FVs
Result: All applications have been dispatched

```
for a in A do
    f_a = feature_extraction(a)
    b_a = backend_selection(f_a, system_pressure)
    for Online_VM in OVs do
        if Online_VM.backend = b_a AND Online_VM.accept(a) then
            dispatch a → Online_VM
            break
        end
    end
    if no available online VM then
        for Free_VM in FVs do
            if Free_VM.backend = b_a AND Free_VM.accept(a) then
                dispatch a → Free_VM
                break
            end
        end
    end
    else if no available free VM with b_a then
        Free_VM← SelectVM(FVs)
        Free_VM.SwitchBackend()
        Free_VM.OptParameters()
        dispatch a → Free_VM
    end
    else if no available free VM AND host resource is available then
        Free_VM← CreateVM(b_a, system_pressure)
        Free_VM.OptParameters()
        dispatch a → Free_VM
        add Free_VM→ Vs
    end
end
```

3.5 System Workflow

Table 1: Backend Parameters

Parameter	Offline Conf.	Offline Conf.	Scale
Total CPU Core	Yes	No	≤ logical core
Total Memory Size	Yes	No	≤ server memory
Page cache Ratio	Yes	No	≤ 0.9
Far memory ratio	Yes	Yes	≤ 0.99
Transparent Large Page	Yes	Yes	ON/OFF
Network channel	Yes	Yes	≤ Total CPU Core
Parallelism	Yes	Yes	≤ logical core

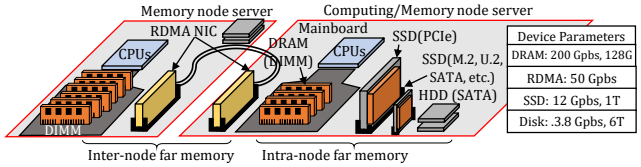


Figure 13: Intra-FM and Inter-FM prototype.

We give the overall system workflow for better understanding and easier deploying our system in virtualized environment. The overall workflow is as Alg. 1 shown.

(1). Feature extraction and labeling. We trace applications page behavior and the runtime behavior on different backends. Based on the profiled information, we extract features of these applications and label them for further procedures, as line 2 in Alg. 1 shows.

(2).VM initialization and warm start. We apply VM with proper CPU cores, memory size, storage size and network for the arrived application groups. By default, each VM processes one kind of application so that the configuration optimization can be more accurate. Since VM creation is time-consuming, we try to search and utilize the proper idle VMs with available hardware resources to warm-start the VM for the arriving requests, as line 4-9 in Alg. 1 shows.

(3). Far memory environment configuring. We prepare the far memory environment when creating the VM, as line 25 in Alg. 1 shows. For storage-based far memory access, we enable direct I/O access with available file read/write APIs. For RDMA-based far memory environment, we use virtualized RDMA technology RS-IOV with PCI passthrough supported, which allows dynamic RDMA sharing among VMs as well as near-bare performance. We also build a configuration shell to build automatically.

(4). Backend selection and switching. as line 18-23 in Alg. 1 shows, the system searches available VM resources for the request and allocate required VMs by the hypervisor. If no available VMs, we load and prepare the related far memory drivers, as line 26-29 in Alg. 1 shows. We check the application offline tables and configure the proper far memory backend, and adapt the optimized configurable parameters. After allocation, we update the resource views of VMs.

4. EVALUATION

Our evaluation answers these questions:

- What's the overall performance and efficiency of our system on real-world testbed? (Section 4.2)
- Does this system bring significant overhead to the original environment? (Section 4.3)

Table 2: System Function Comparison

Baselines	Intra-FM	Inter-FM	Multi-end	Parallelism
Linux swap [25]	✓	×	×	×
Fastswap [2]	×	✓	✓	×
TMO [46]	✓	×	✓	×
XMemPod [7]	✓	✓	✓	×
DMSwitcher (**Ours**)	✓	✓	✓	✓

Table 3: Evaluated Workloads

Abbr	Algorithm Description	Framework	Mem. Footprint
sort	quicksort	C++ std [5]	2G∼10G
lpk	Linpack benchmark	C++ std [5]	2G∼10G
tobinary	fire read to binary	C++ std [5]	2G∼20G
bfs	breadth first search with VFS	Gridgraph [49]	2G∼30G
tf-incep	tensorflow inception	Tensorflow [38]	4G∼20G
tf-infer	tensorflow resnet inference	Tensorflow [38]	4G∼20G

- How does each component of DMSwitcher contribute to the overall Performace, including page analyzer, FM switcher, and resource adaptor? (Section 4.4)

4.1 Experimental Setup

Multi-backend FM testbed. To obtain workload traces, we build both Intra-FM and Inter-FM systems on physical machines equipped with RDMA, SSD, etc., as shown in Figure 13. Each server node is provisioned with two 10-core Xeon CPUs, 128 GB of memory, 1TB SSD, and 6 TB of HDD, and a Dual-Port Mellanox ConnectX-5 RDMA NIC supporting dual-port 100 Gb/s Ethernet connected with copper fabric. We deploy several far memory runtimes as baseline systems including Linux swap, Fastswap [2], TMO [46], XMempod [7], and our DMSwitcher, as shown in Table 2. The RDMA driver we used is version 4.3.0 of the OFED kernel, and it uses the RoCE protocol. We collect the overall runtime using Linux Time, maps, etc., real-time memory usage using Intel VTune, memory access latency using PMU and record the page fault number with Linux perf. Usually, the over latency is the sum of kernel-level time detected by `sys time` and user-level time detected by `cpu time`.

Workload. We use real-world applications that are commonly used in data centers today as Table 3 shows. In our detailed performance evaluation, we take the compute-intensive applications quicksort and linpack, and the I/O-intensive applications tf-inception and tf-resnet as evaluated examples.

4.2 Overall Benefits

We have implemented a multi-backend far memory system on a physical machine. The achievement of our system is the ability to switch multiple swap backends on a single machine, which can provide great scalability. We give the chance to virtualize far memory access in virtual machines with high parallellism. We also provide the optimization of far memory backends for better performance. We can turn on an unlimited number of virtual machines with high parallel access to distant memory paths.

4.2.1 System Swap Speedup

Our system can significantly improve the swap performance with limited overhead to user-level computation performance compared with baselines. Table 4 shows the performance of swap performance speedup and the overall latency speedup. We choose the workload linpack, quicksort

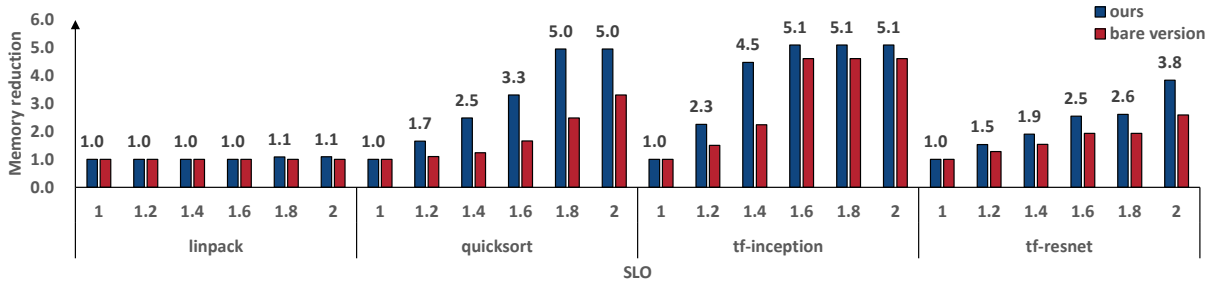


Figure 14: Memory reduction of the bare version (virtualized FM w/o DMSwitcher) and our design on different SLOs.

Table 4: Swap Speedup on different backends

	backends	lpk	qs	tf-i	tf-r	average
Swap speedup	SSD (Linux swap)	1.05×	1.06×	1.45×	1.25×	1.20×
	DRAM (Fastswap)	1.11×	1.19×	**2.13×**	**2.01×**	**1.61×**
	RDMA (Fastswap)	**1.11×**	**1.80×**	1.90×	1.59×	1.60×
Overall speedup	SSD (Linux swap)	1.10×	1.02×	1.17×	1.16×	1.12×
	DRAM (Fastswap)	**1.30×**	**1.50×**	1.08×	1.03×	**1.23×**
	RDMA (Fastswap)	1.15×	1.07×	**1.26×**	**1.37×**	1.21×

with local memory ratio 0.5 and tf-inception, tf-resnet with local memory ratio 0.1 on RDMA backend. We use the sum of kernel-level `sys time` to evaluate swap performance improvement. In the results, our design brings improvement of 1.45× speedup on SSD backend compared with Linuxswap, 2.13× speedup on DRAM backend compared with Fastswap, and 1.9× speedup on RDMA backend compared with Fastswap. For workloads containing file reads such as tf-inception, the swap performance requirement is higher, and the performance of I/O intensive workloads such as tf-inception improvement is greater that of computing intensive workloads.

We use the sum of kernel-level `sys time` and user-level `cpu time` to evaluate overall performance improvement. Usually, the `cpu time` is not affected if far memory access is few, which we show it detail in the next subsection. Computing can be delayed more obviously when far memory access operations are more, which stalls CPU computation. The overall latency speedup still achieves up to 1.5×, taking the far memory overhead into account.

4.2.2 System Memory Efficiency

We show the memory efficiency effects of our system. The memory efficiency can be calculated by the times of memory reduction and workload runtime SLO requirement. Although saving local memory brings runtime overhead, we can save much more local memory at the same runtime. We explored the memory savings of this system with different execution latency SLOs (allowed latency growing times over the original workload latency) and compared it with Linux swap baseline. The results are shown in the figure 14. As the SLO increases, we have increasing memory saving. Our system has nearly 2× memory reduction on quicksort workload.

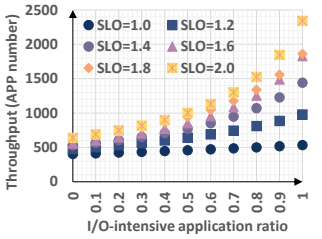


Figure 15: Throughput

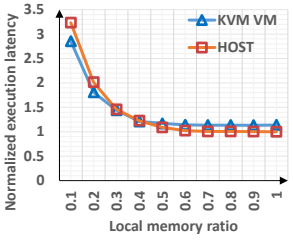


Figure 16: KVM overhead

In applications such as quicksort, tf-inception and tf-resnet, the local memory reduction is more than linpack as SLO requirements arise. This is because linkpack is sensitive to far memory so we can not squeeze more local memory under the SLO limitation. Note that not larger SLO brings memory efficiency improvement. The reason is that the performance down-gradation usually becomes faster when local memory is less than the hot data size. Proper SLO like 1.4 in quicksort and tf–inception brings better user experiments as well as larger memory efficiency.

Impact of Program Distribution. To further evaluate the system throughput and scalability, we show the results of system throughput when deploying different program distributions under the SLO limits. We set the I/O program proportion from 0 to 1 totest the task throughput changes. As Figure 15 shows, Larger SLO brings more obvious throughput improvement. However, not large SLO get the maximum efficiency, for example, SLO 1.6 and 1.8 has close throughput so SLO=1.6 is a much better choice than SLO=1.8. In addition, the larger parts of I/O intensive applications, the high throughput in the system since most of I/O intensive applications are friendly to far memory access.

4.3 System Overhead

The system overhead is mainly generated by the KVM virtualization. The overhead of virtualization consists of cpu virtualization overhead, memory virtualization overhead, network virtualization overhead and I/O virtualization overhead. Kvm technology achieves much smaller overhead, after several years of development.

We measure the running latency on the physical machine and the virtual machine respectively. Figure 16 shows overall the normalized execution time of tf-inception with different local memory ratio. Overall, the system on the virtual machine runs steadily and brings about **15%** speedup with less

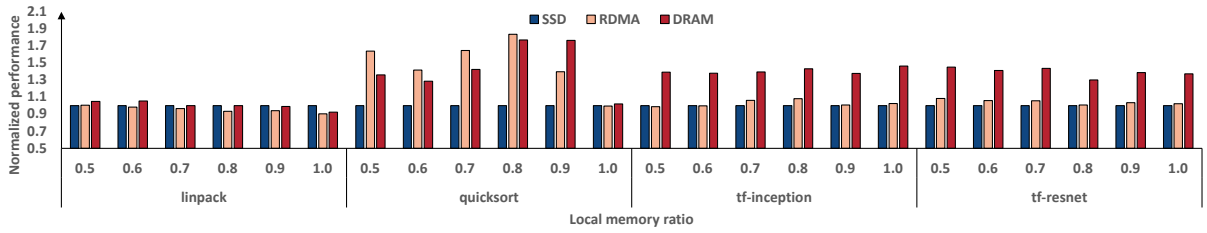
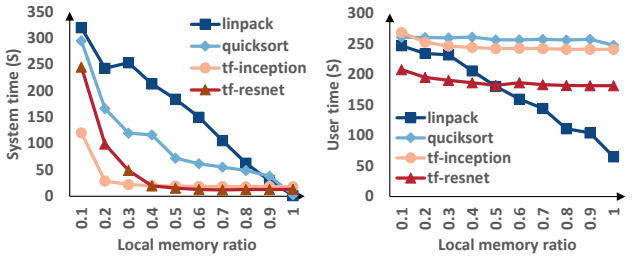
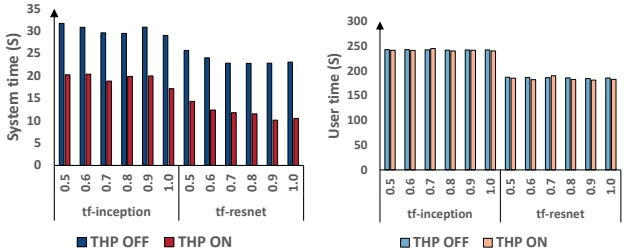


Figure 17: The swap performance speedup of each workload on different swap backends, SSD, RDMA and DRAM.



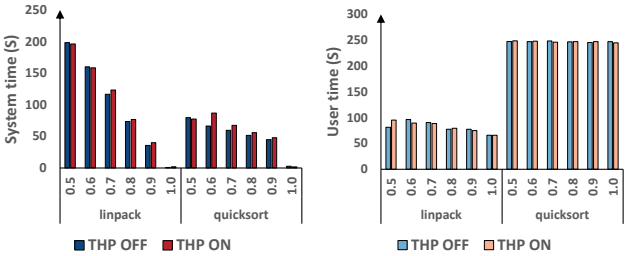
(a) System time (b) User time

Figure 18: Execution time of workloads with different local memory ratio.



(a) Swap performance (b) Execution performance

Figure 19: I/O-intensive workloads' execution with transparent huge page on/off.



(a) Swap performance (b) Execution performance

Figure 20: Compute-intensive workloads' execution with transparent huge page on/off.

swap competing. When local memory is smaller than 0.3, there are lots of two-layer swap on traditional physical machines so the direct-connected FM on virtual machine has higher performance than host. When local memory ratio increases, the number of swap decreases sharply and the overhead of virtualization is starting to appear. The maximum overhead is about **5%**, which consists of **2%** system time overhead and **6%** user time overhead in detail. The ovetrall speedup has easily covered the side-back of virtualization.

4.4 Performance Breakdown

4.4.1 Swap Backend Selection Efficiency

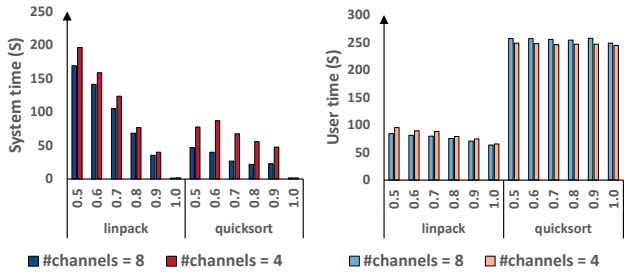
To demonstrate the effects of application-aware backend selection, Figure 17 shows the normalized performance based on SSD latency of evaluated workloads on different swap backends. We tested three types of memory backends, including SSD backend, an RDMA backend, and an RMA-connected DRAM backend. Theoretically, the read/write speed of the three backends should be $DRAM > RDMA > Disk$. According to the results, the `sys time` of different workloads shows different preferences with $DRAM < RDMA < Disk$. Quicksort, tf-inception, and tf-resnet perform better latency growing on RDMA backend than DRAM backend on 0.9 local memory ratio. Backend selection improves up to 2× of overall performance.

4.4.2 System Parameter Effectiveness

We evaluate the effectiveness of three parameters: local memory ratio, THP, and data transfer channels.

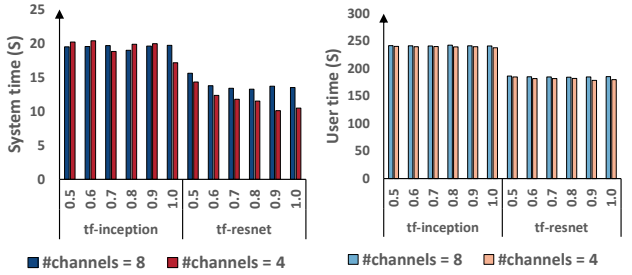
Local memory ratio. In our experiments, we control the local memory ratio by limiting the upper limit of local memory size in `Cgroup`. Figure 18 shows the `sys time` and `cpu time` for different applications on different local memory ratios. Overall, applications have different performance degradation trends when limiting the local memory ratio. In addition, `sys time` has obvious performance change while `cpu time` of each application nearly keeps unchanged in each local memory condition. In detail, `sys time` of compute-intensive applications such as linpack and quicksort decrease linearly as the local memory ratio increases. This is because these two types of applications have no files to read and page accessing is even over time. Applications like tf-inception and tf-resnet that have more file-backed pages have rapidly kernel-level latency decreasing when local memory ratio is 0.1 to 0.2, while they show less sensitivity on local memory ratio from 0.3 to 1. This infers that I/O intensive workloads can save much more local memory compared with compute-intensive workloads.

Transparent huge page. We explore the impact of page size by controlling whether the transparent huge page is turned on or off. According to figure 19 (a), tf-resnet and tf-inception's `sys time` with THP on are smaller than those



(a) Swap performance (b) Execution performance

Figure 21: Compute-intensive workloads' execution with different number of RDMA channels.



(a) System time (b) User time

Figure 22: I/O-intensive workloads' execution with different number of RDMA channels.

with THP off, and it can gain 50% speed up in tf-resnet when local memory ratio is small. This is because when the application memory is large, using larger memory pages can significantly reduce the number of page entries and thus directly reduce the memory access time. For compute-intensive applications like linpack and quicksort (figure 20 (a)), the page access is relatively fixed and THP has very little impact on system time. For user time (figure 20 (b) and 19 (b)), the influence is small too. This demonstrates that the use of THP has different degrees of impact on different applications. When huge page is being swapped out, it will be divided into small pages, and then swapped out. This will generate lots of swaps in a short period of time.

Data Transfer Channels. We allocate the same number of CPUs to each program without affecting the program during the experiment and adjust the RDMA channels by adjusting the number of CPUs. First, the number of data transfer RDMA channels has little influence on applications' execution thus bringing little overhead to the system, as Figure 21 (b) and 22 (b). Second, RDMA channel number affects the performance of applicaitons swap performance. As Figure 21 (a) shows, when the RDMA channels are set to 8, linpack can gain about 20% latency reduction and quciksort can gain about more than 50% latency reduction. That's because compute-intensive applications spend more time on computation, which access far memory frequently. In contrast, I/O-intensive applications have less frequent swap usually swap in the model loading phase as Figure 22 (a) shows. It is possible to allocate fewer CPU resources and RDMA channels to save costs.

5. RELATED WORK

Disaggregated Memory Runtimes. *Composable Disaggregated Infrastructure* (CDI) [30, 32, 34, 35, 37] has attached great attention. Some use OS swap kernel to handle data offloading such as Infiniswap [16], FastSwap [31], and Fastswap [2]. AIFM [36] and Kona [6] offload data in cache-line size. Freeflow [24], FaRM [13], LITE [40] and Fargraph [44] provide user-defined frameworks to gain high performance. The *vFM* takes local non-volatile memory and storage as far memory. Most of them are page-based, such as zswap [25] and Hybridswap [48]. pDPM [39], XMemPod [7], and TMO [46] adopt memory objects with variable size. However, the above works fail to addresse the problem of managing multi-backend far memory backends.

Application Awareness. Related works often configure far memory environment for high performance by fully considering the page behavior. *Space-based working set partition.* Fastswap [2] uses page size 4K to offload to and fetch data from far memory. AIFM [36] use memory object with size of 64B to 4KB to offload data on TCP/IP-based far memory. Fargraph [44] uses data segments of 4K to 2M to transfer data on RDMA-based far memory.*Time-based data shuffle management.* Data shuffle can occur when offloading relatively cold data to far memory without spatial consideration [24,41]. Spatial data behavioris handled by swapping out cold data by memory resource limitation settings [45]. Existing works often offload read-only data to avoid data shuffle [13,44] or adopt a detect-and-act approach [45,46] to process as soon as possible. Our work dives deeper than existing works by detailed analysis of software and hardware parameters.

Hybrid Memory Management. Data in far memory can be considered inclusive or exclusive to the local memory [21]. (1). *Swap-based migration method* can release more local space but requires frequent I/O. There are some works concentrated on swap mechanism design in different service levels, such as VSwapper [3] across virtual machines, Hybridswap [48] on virtual machines and local disks, XMemPod [7] on virtual machines and RDMA-based remote memory, TMO [46] for local heterogeneous SSD backends and Hybrid [41] for SRAM and DRAMs, etc. (2). *Copy-based caching method* can improve performance but requires data consistency design. Caching method prefers making a copy in the near space. CPU processors can quickly fresh data from the buffer and feed them to LLC of CPUs, which better utilize the memory bandwidth [28,41]. Our work fully considers the cache and swap configuration with high efficiency.

6. CONCLUSION

In this work, we design an intelligent DM switcher on multi-backend disaggregated memory. We analyze application page behavior and system information in a fine-grained way. We propose our backend selection strategy that achieves higher efficiency. We build FM switcher that enables multi-backend inter-node and intra-node far memory access on optimized parameters with low overhead. DMSwitcher shows up to 2.1x speedup of swap performance and saves up to 5.1x memory space within the same time span compared with state-of-the-art works. Furthermore, our design greatly scales out the concurrent access of far memory environments and improves the efficiency of resource management.

REFERENCES

[1] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati *et al.*, "Remote regions: a simple abstraction for remote memory," in *USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 775–787.

[2] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *European Conference on Computer Systems (EuroSys)*, 2020, pp. 1–16.

[3] N. Amit, D. Tsafrir, and A. Schuster, "Vswapper: A memory swapper for virtualized environments," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 349–366, 2014.

[4] R. Biswas, X. Lu, and D. K. Panda, "Accelerating tensorflow with adaptive rdma-based grpc," in *IEEE International Conference on High Performance Computing (HiPC)*, 2018, pp. 2–11.

[5] C++, "C++ standard library headers," https://cppreference.com/, accessed on March, 2022.

[6] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 79–92.

[7] W. Cao and L. Liu, "Hierarchical orchestration of disaggregated memory," *IEEE Transactions on Computers (TC)*, vol. 69, no. 6, pp. 844–855, 2020.

[8] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 926–939.

[9] C. Consortium, "Compute express link," https://www.computeexpresslink.org/, accessed on December 1, 2021.

[10] O. Consortium, "Opencapi specification," https://opencapi.org/, accessed on December 1, 2021.

[11] N. Corporation, "Nvlink and nvswitch," https://www.nvidia.com/en-us/data-center/nvlink/, accessed on March 14, 2023.

[12] V. Developers, "Valgrind," https://valgrind.org/ or , accessed on March, 2023.

[13] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 401–414.

[14] J. Gonzalez, M. G. Palma, M. Hattink, R. Rubio-Noriega, L. Orosa, O. Mutlu, K. Bergman, and R. Azevedo, "Optically connected memory for disaggregated data centers," *Journal of Parallel and Distributed Computing*, vol. 163, pp. 300–312, 2022.

[15] R. E. Grant, M. J. Levenhagen, M. G. Dosanjh, and P. M. Widener, "Rvma: Remote virtual memory access," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 203–212.

[16] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 649–667.

[17] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. Lui, "{SmartMD}: A high performance deduplication engine with mixed pages," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 733–744.

[18] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," *arXiv preprint arXiv:2108.03492*, 2021.

[19] W. Huangfu, K. T. Malladi, A. Chang, and Y. Xie, "Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 727–743.

[20] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[21] B. Jacob, D. Wang, and S. Ng, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[22] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. Dreslinski, "Improving performance of flash based key-value stores using storage class memory as a volatile memory extension," in *USENIX Annual Technical Conference (USENIX ATC)*, 2021, pp. 821–837.

[23] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: Elastic far-memory for stateful serverless analytics," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 697–713.

[24] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "Freeflow: Software-based virtual rdma networking for containerized clouds," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 113–126.

[25] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid *et al.*, "Software-defined far memory in warehouse-scale computers," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 317–330.

[26] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 488–504.

[27] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[28] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

[29] Z. Li, N. Liu, and J. Wu, "Toward a production-ready general-purpose rdma-enabled rpc," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019, pp. 27–29.

[30] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *ACM SIGARCH computer architecture news (CAN)*, vol. 37, no. 3, pp. 267–278, 2009.

[31] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu, "Memory disaggregation: Research problems and opportunities," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1664–1673.

[32] D. Montgomery, "The future of data infrastructure: Cdi," https://www.datacenterknowledge.com/industry-perspectives/future-data-infrastructure, accessed on December 1, 2021.

[33] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter," in *European Conference on Computer Systems (EuroSys)*, 2018, pp. 1–12.

[34] M. Ogleari, Y. Yu, C. Qian, E. Miller, and J. Zhao, "String figure: A scalable and elastic memory network architecture," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 647–660.

[35] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "Thymesisflow: a software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 868–880.

[36] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "Aifm: High-performance, application-integrated far memory," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 315–332.

[37] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed os for hardware resource disaggregation," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 69–87.

[38] Tensorflow, "An open source machine learning framework for

everyone." https://github.com/tensorflow, accessed on March, 2022.

[39] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-valuestores," in *USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 33–48.

[40] S.-Y. Tsai and Y. Zhang, "Lite kernel rdma support for datacenter applications," in *Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 306–324.

[41] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Hybrid2: Combining caching and migration in hybrid memory systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 649–662.

[42] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A memory-disaggregated managed runtime," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 261–280.

[43] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu, "Canvas: Isolated and adaptive swapping for multi-applications on remote memory," *arXiv preprint arXiv:2203.09615*, 2022.

[44] J. Wang, L. Chao, T. Wang, L. Zhang, P. Wang, J. Mei, and M. Guo, "Excavating the potential of graph workload on rdma-based far memory architecture," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 375–386.

[45] J. Wang, C. Li *et al.*, "Hyfarm: Task orchestration on hybrid far memory for high performance per bit," in *International Conference on Computer Design (ICCD)*. IEEE, 2022.

[46] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang *et al.*, "Tmo: transparent memory offloading in datacenters," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 609–621.

[47] D. Zahka and A. Gavrilovska, "Fam-graph: Graph analytics on disaggregated memory," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 81–92.

[48] P. Zhang, X. Li, R. Chu, and H. Wang, "Hybridswap: A scalable and synthetic framework for guest swapping on virtualization platform," in *IEEE Conference on Computer Communications (INFOCOMM)*, 2015, pp. 864–872.

[49] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *booktitle=USENIX Annual Technical Conference (USENIX ATC)*, 2015, pp. 375–386.