Disaggregated Cloud Memory with Elastic Block Management

Kwangwon Koh, Kangho Kim, Seunghyub Jeon, and Jaehyuk Huh, *Member, IEEE*

Abstract—With the growing importance of in-memory data processing, cloud service providers have launched large memory virtual machine services to accommodate memory intensive workloads. Such large memory services using low volume scaled-up machines are far less cost-efficient than scaled-out services consisting of high volume commodity servers. By exploiting memory usage imbalance across cloud nodes, disaggregated memory can scale up the memory capacity for a virtual machine in a cost-effective way. Disaggregated memory allows available memory in remote nodes to be used for the virtual machine requiring more memory than its locally available memory. It supports high performance with the faster direct memory while satisfying the memory capacity demand with the slower remote memory. This paper proposes a new hypervisor-integrated disaggregated memory system for cloud computing. The hypervisor-integrated design has several new contributions in its disaggregated memory design and implementation. First, with the tight hypervisor integration, it investigates a new page management mechanism and policy tuned for disaggregated memory in virtualized systems. Second, it restructures the memory management procedures and relieves the scalability concern for supporting large virtual machines. Third, exploiting page access records available to the hypervisor, it supports application-aware elastic block sizes for fetching remote memory pages with different granularities. Depending on the degrees of spatial locality for different regions of memory in a virtual machine, the optimal block size for each memory region is dynamically selected. The experimental results with the implementation integrated to the KVM hypervisor, show that the disaggregated memory can provide on average 6 percent performance degradation compared to the ideal local-memory only machine, even though the direct memory capacity is only 50 percent of the total memory footprint.

Index Terms—Disaggregated memory, cloud computing, virtualization, remote memory

1 Introduction

Proliferation of data-intensive workloads, such as in-memory databases, data caching, bioinformatics, and graph processing, has been tremendously increasing the memory capacity requirements in cloud servers. To accommodate such big memory applications, cloud providers have begun to offer large machine types with more than 1 TB of memory. Google Compute Engine announced a plan to provide 1 TB of memory in 2017 [1]. Amazon already started to support EC2 x1e.32xlarge with 4 TB of memory in four AWS regions and announced a plan to launch EC2 instances with 16 TB of memory [2], [3]. With the ever-increasing demand for fast in-memory big-data processing, the cloud providers are expected to adopt large VM instances more and more widely.

However, offering such large VM instances requires massive investment for the existing infrastructure currently composed of commodity volume servers connected through high-speed interconnects. For example, the EC2 large virtual machine instance with 4 TB memory is supported only in 4 regions, limiting the benefits of large memory machines for remotely located users. Furthermore, large memory machines with TBs of memory and high core counts, provide far less performance per cost or performance per watt than the commodity volume servers. Meanwhile, in the cloud systems, the heterogeneity of workloads commonly incurs the imbalance of memory usages in each node. Such variance in memory usages can cause memory shortages in some machines, while other machines have ample free memory. The inherent memory imbalance can open a new opportunity to provide a large memory virtual machine (VM) cost-efficiently by combining the free memory in multiple servers into a single unified memory [4].

To support large memory with volume commodity servers, *disaggregated memory* allows distributed memory across different physical servers to be used as a single memory, creating an illusion of larger memory than the physical memory limit on a single machine. This paper proposes a new hypervisor-based disaggregated memory, providing large scalable memory for guest VMs transparently. With the disaggregated cloud memory (`dcm`) based on VMs, an application on a VM can use the remote memory in other machines in addition to the local memory without any modification in the application binary and guest operating systems. Instead, the hypervisor hides all the complexity of accessing remote memory via page-oriented remote memory accesses. If the accessed memory page from a guest VM

- *K. Koh is with the School of Computing, KAIST, Daejeon 34141, Korea, and also with the High Performance Computing Research Group, ETRI, Daejeon 34129, Korea. E-mail: wkoh@kaist.ac.kr.*
- *K. Kim is with High Performance Computing Research Group, ETRI 34129, Daejeon, Korea. E-mail: khk@etri.re.kr.*
- *S. Jeon is with Basic Technology Research Center for Next Generation OS, ETRI 34129, Daejeon, Korea. E-mail: shjeon00@etri.re.kr.*
- *J. Huh is with the School of Computing, KAIST, Daejeon 34141, Korea. E-mail: jhuh@calab.kaist.ac.kr.*

Manuscript received 12 Nov. 2017; revised 18 June 2018; accepted 22 June 2018. Date of publication 28 June 2018; date of current version 19 Dec. 2018. (Corresponding author: Jaehyuk Huh.)
Recommended for acceptance by P. Gratz.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2851565

0018-9340 © 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

is not in the local memory, a page fault will initiate an access to the remote memory, and the new local memory page will be allocated for the faulting VM.

In the proposed hypervisor-based design, the disaggregated memory support is directly integrated to the page management in the KVM hypervisor. Unlike the prior study [5] which supports the remote memory as a block device and uses the existing storage-based swap mechanism, the proposed integrated design can provide fine-grained adjustments of memory eviction and high scalability with the hypervisor integration.

One of the key observations from the disaggregated cloud memory is that the granularity of memory fetching from remote machines affect the frequency of remote memory accesses significantly. If the memory access patterns have high spatial locality, increasing the granularity of memory fetching (*block size*) can reduce future page faults by a prefetching effect. If the memory access pattern exhibits small random accesses, a small block can reduce the remote memory access overhead for each fault. This paper proposes a dynamic block size adjustment technique, called *elastic block*, to find the optimal block size for each VM. The proposed mechanism can assign the optimal block size not only for each VM but also different memory regions in VMs, as it tracks the spatial locality in the hypervisor-managed memory map for each VM.

Compared to the prior work, the paper has the following new contributions.

- The paper proposes an integrated hypervisor-based design for disaggregated memory. Instead of relying on the conventional swap system designed for slow disks, the disaggregated memory support is directly added to the memory management in the KVM hypervisor.
- The integrated design overhauls the memory management mechanisms and policies with a new replacement scheme turned for disaggregated memory, a latency hiding mechanism by overlapped memory reclamation and network operations, and scalability improvements.
- This study identifies the importance of selecting right memory fetching sizes from remote systems in disaggregated memory designs. The best block size may vary by many factors including access patterns, available local memory, I/O activities, and program phases.
- The design allows a fine-grained adjustment of block size to minimize the overhead of remote memory accesses, while maximizing prefetching effects. Block sizes are dynamically adjusted for each page address in a virtual machine, and thus the adjustment mechanism can find the best block size for different regions even within a virtual machine

The implementation tightly integrated to the KVM hypervisor also exploits the reduced latency provided by one-sided remote direct memory access (RDMA) operations which do not require an involvement of a processor, and the experimental results based on the implementation show that the disaggregated memory can support 6 percent performance degradation on average compared to the ideal large memory machine with the entire memory footprint stored in the local memory, even though the local memory capacity with our scheme is only 50 percent of the total memory footprint [6].

The rest of this paper is organized as follows. Section 2 presents the background and prior work on disaggregated memory. Section 3 proposes the disaggregated cloud memory architecture and Section 4 shows how to exploit the spatial locality for minimizing performance degradation. Section 5 presents the experimental results. Section 6 discusses the comparison to the swap subsystem, and the adoption of new memory. Section 7 concludes the paper.

2 Background

2.1 Memory Usage Imbalance in Clouds

The cloud systems serve heterogeneous guest applications from clusters of physical machines. Due to the heterogeneous memory usages of the user applications, available free memory may vary widely across cloud nodes [5], [7], [8], [9]. Furthermore, each node is configured to accommodate the worst-case peak memory usage, and thus the memory over-provisioning can lead to severe memory under-utilization and imbalance of memory usage across the cluster [5]. Without the flexibility of disaggregated memory, tightly provisioning bare-metal servers is challenging since the system administrator must determine a particular configuration of hardware including the size of the DRAM at the cluster installation time [10]. A study, analyzing two production clusters, showed that during 70 percent of the running time, the clusters experience severe memory utilization imbalance [5]. Samih et al. showed that the aggregated memory capacity reaches 437 TB during a typical workday in a data center cluster [11]. However, only 69 percent of its overall memory capacity is allocated. The heterogeneity of guest applications and per-node memory over-provisioning results in the available free memory spaces across the cloud cluster, although the availability can fluctuate.

The idle remote memory scattered across the cluster can constitute a logical pool of memory, accessible with high-speed cluster networks supporting RDMA functionality [6]. The logical pool can be dynamically partitioned and exposed to the VMs in the cluster. The logical memory pool provides a cost-effective way to scale and share server memory to accommodate applications requiring large memory capacities in their VMs.

2.2 Remote Memory Access Performance

The idea of using remote memory has been proposed and implemented for more than two decades. In recent years, the network speed increases dramatically and such high-speed networks have become available in common server clusters. With such readily available high bandwidth networks, using remote memory is drawing more and more attention from multiple communities than ever. The high-bandwidth low-latency interconnects constitute the basis of memory disaggregation and rack scale computing [12].

The remote memory is accessed by the RDMA controller provided by modern interconnects. The RDMA supports the zero-copy and one-sided control of data movement. The zero-copy prevents the data from being copied to/from a kernel buffer for data transmission, and the one-sided control allows that CPU involvement is not necessary in remote systems for

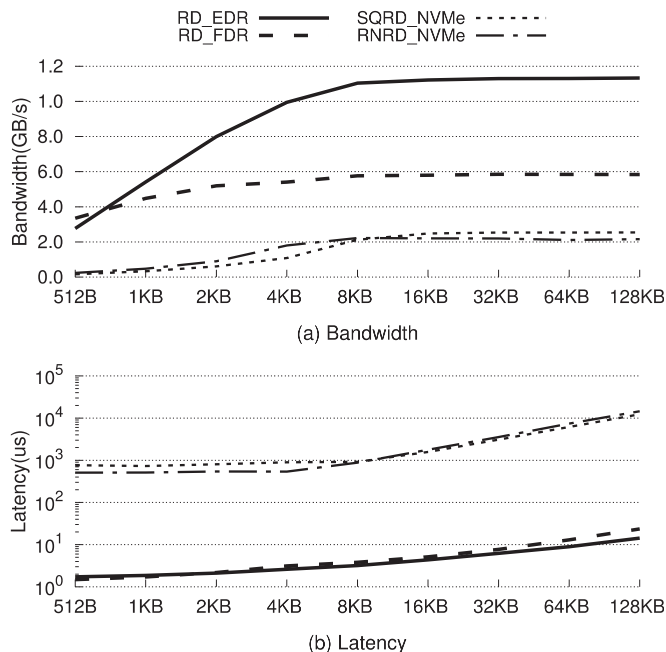

Fig. 1. Performance of RDMA operations with Mellanox ConnetX-3(IB FDR) and ConnectX-4 (IB EDR), and DMA operations with Intel P750 (NVMe SSD).

data transfers, unlike SEND/RECV model [13]. In addition to the high performance of RDMA, the one-sided control makes networked systems more robust because RDMA data connections survive the software failures including OS kernel crashes on the remote server. The separation of RDMA data route from the CPU domain provides the improved robustness.

The maximum bandwidths of InfiniBand FDR 4x, FDR 12x, and EDR 12x are 6.8 GB/s, 20.5 GB/s, and 37.5 GB/s respectively. Fig. 1 shows the bandwidth and latency of InfiniBand FDR 4x and InfiniBand EDR 4x, compared to those with a state-of-the-art NVMe SSD. In the figures, the x-axis is the sizes of message increased from 512 B to 128 KB. In the results, RD_, SQRD_, and RNRD_ prefixes notate read, sequential read, and random read respectively. For the SSD evaluation, the messages are sent either in sequential or random order, as the SSD has different bandwidth and latency characteristics depending on the access patterns. Write performance exhibits similar trends to the read operations, and thus is not shown in the figures.

In the figures, the network performance far exceeds that of the SSD, showing higher than 5 GB/s bandwidth and lower than 4us latency in case of 4 KB message size [14], [15]. In addition, InfiniBand EDR 4x supports 12 GB/s of maximum bandwidth, with similar latencies to FDR 4x. Messages larger than 4 KB can utilize the maximum bandwidth on both FDR and EDR network according to our measurement presented in Fig. 1a. As shown in Fig. 1b, although the latency increases with the message size, the latencies with 8, 16, 32, and 64 KB are increased to 1.2x, 1.64x, 2.47x, and 4.15x, respectively, compared to the latency of the 4 KB message. This observation indicates that increasing the message size can reduce per-page latencies by amortizing the transmission initiation and finalization overheads. As a comparison to the RDMA performance, the NVMe SSD can support 2,800 MB/s for 64 k sequential read/write and 460 K IOPS for 4 KB random read/write micro-benchmark in its manufacturer specification. Fig. 1 presents the SSD measurement by using the fio [16], showing 2,600 MB/s as the maximum bandwidth and much longer latencies than InfiniBand families.

RDMA is widely adopted by modern interconnects such as RDMA over Converged Ethernet (RoCE), internet Wide Area RDMA Protocol (iWARP), InfiniBand and Intel Omni-Path Architecture (OPA). Recently, Intel has announced that Xeon Scalable processor optionally integrates OPA on its package which delivers 2 * 100 Gbps bandwidth with two ports. With the broad adoption, RDMA is expected to become even more cost-effective in the near future [17].

2.3 Related Work

Swap-Based Disaggregated Memory. An evolutionary design for disaggregated memory is to rely on the current swap subsystem available in most of the operating systems. There have been several prior studies focused on providing the expansion of the memory abstraction by using the swap subsystem [8], [18], [19], [20], [21], [22]. The network RAM, based on the observation of uneven free memory in computing clusters, proposed the networked memory system exploiting faster networks than disks [22]. Nswap also proposed a network swap system for heterogeneous Linux clusters to allow any cluster node suffering from memory pressure to use the remote memory in the cluster [8]. However, the techniques require the increased utilization of processors due to additional computation for controlling network operations on each memory donation node.

To reduce the computation overheads on the memory donation node, the one-sided control provided by RDMA-capable interconnects eliminates the involvement of the processor for moving data. High performance block device (HPBD) exploited a reduced latency of networks over disks and high bandwidth of up to 10 Gbps supported the RDMA capable networks such as InfiniBand and Quadrics [23]. NBDX and NVMEoF are RDMA-based virtual block devices which use either remote memory or remote storage as their back-end storage [24], [25]. INFINISWAP implemented a block device which supports a remote memory paging-based caching system designed for RDMA networks in a decentralized manner [5].

However, the swap system has been designed to cope with the huge performance disparity between the memory and traditional disk (block devices), and its memory reclamation is focused on the pages initially used but rarely re-used [26]. As faster block devices, such as NVMe SSD, Intel Optane SSD, and Samsung Z-SSD, emerge, a patchset for making the swap subsystem more scalable [27] has been introduced. However, such improvements address only a latency optimization of swapping operation by reducing contentions on locks in swap caches and swap devices. The idle page tracking mechanism of Linux requires clearing the accessed bit of page table entry for identifying the idleness of each page [28], [29]. Recently, SPAN proposed a prefetching mechanism by improving the current swap architecture and by exploiting the parallelism of future NVM storage [30]. It is evaluated by using an emulated 3D Xpoint device. However, although modern processors have multiple cores, it did not address a scalability concern for updating shared resources such as pattern table used for prefetching.

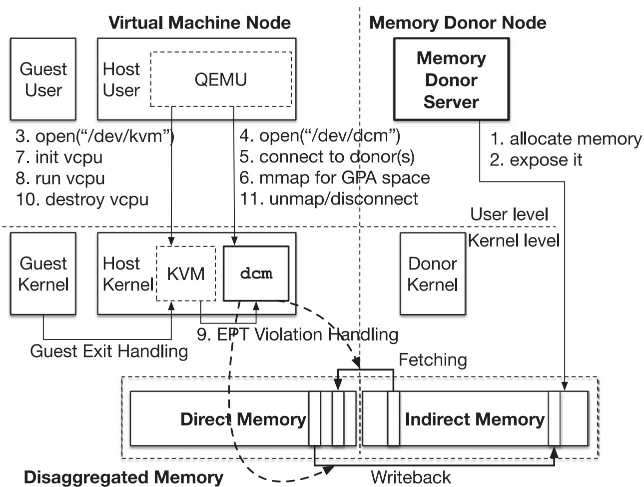

Fig. 2. Overall architecture: software components and brief execution flow.

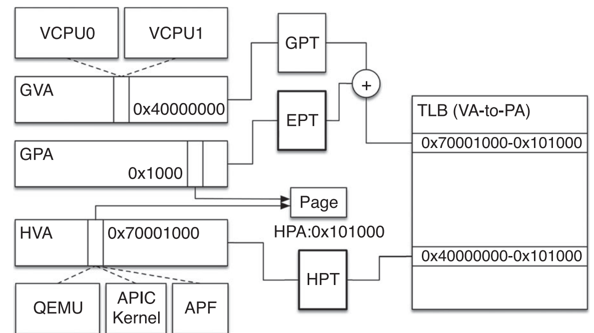

Fig. 3. Memory address translation of QEMU/KVM with EPT.

Disaggregated Memory Managed by Hypervisor. Memory management for the virtual machine has been studied since early 2000. Carl A. Waldspurger addressed several memory management mechanisms and policies, such as ballooning, idle memory tax, content-based page sharing, and hot I/O page remapping, introduced in VMware ESX Server [31]. Amit et al. studied the hypervisor swap by implementing the improved swap subsystem (VSwapper) for the KVM [32], [33]. The VSwapper proposed a guest-agnostic memory swapper for solving various types of superfluous swap operations, decayed swap file sequentiality, and ineffective prefetching decision. Ye et al. suggested a hybrid memory model motivated by the lower-cost, higher-density, and lower-power memory technologies. They prototyped the memory management architecture with a virtual machine monitor [34]. Based on the memory management of the hypervisor, MemX project studied a hypervisor-level implementation for providing cluster-wide memory as a single memory abstraction [35], [36]. Lim et al. also simulated and implemented a memory disaggregation system by using the Xen hypervisor with the content-based page sharing [4], [9]. Moreover, Rao et al. conducted a feasibility study of disaggregated memory connected to compute and storage systems using commercial network technology for an widely deployed workload, Spark SQL analytics queries [10], [37].

This study designs a hypervisor-based disaggregated memory system from scratch to overcome the limitation of the swap-based approach. Its hot-cold separation mechanism is integrated to the hypervisor memory management for fine-grained tracking of access status tracking. Remote memory accesses are fine-tuned to hide their latencies as much as possible. The overall structure is designed to provide scalability for many cores since the disaggregated memory system must process much higher rates of page reclamation than the traditional swap system. Finally, this study supports the application-aware elasticity to the block management to improve the performance by inferring the memory access pattern of the VM.

3 Memory Disaggregation Architecture

The memory disaggregation architecture provides an illusion of a big-memory VM whose memory extends beyond the physical machine boundary. Multiple physical machines connected through RDMA-capable networks donate their free memory to the VM. The hypervisor allows the big-memory VM to access the donated memory transparently. The entire guest software stacks, including guest operating systems (OS), middlewares, and applications running on the big-memory VM do not require any changes to use the disaggregated memory. In this section, the overall architecture, direct memory replacement policy, and scalability consideration are described.

3.1 Overall Architecture

The processor directly accesses the local memory, or *direct memory*, installed in the machine, but requires indirect accesses to the memory on the remote node (*indirect memory*). In contrast to the direct memory, the indirect memory access requires a page-level fetching operation: fetching a page from the remote node on demand, selecting a victim page from the direct memory to make a free direct page, and writing back the victim page to a remote node. The remotely fetched memory pages are accessible from the direct memory, until they are later evicted by the replacement policy.

We implemented the overall architecture with a kernel module, `dcm`, running on the node requiring memory extension. The kernel module is linked to Linux Kernel-based Virtual Machine (KVM) [33] and provides most of the functionality for the disaggregated memory system. In addition, each donor node runs a memory donor application, and the donor application grants their memory to the nodes in a rack through the RDMA networks. The KVM module is modified to invoke the extended page tables (EPT) violation handler of `dcm` when an EPT violation occurs during the execution of VM context. Fig. 2 illustrates the execution flow of `dcm`: it opens `dcm` device file, connects to the donors granting their memory as RDMA regions, and maps the extended memory region to the VM guest physical address (GPA) space. It handles page faults on both the host virtual address (HVA) and GPA spaces by manipulating the host page tables (HPT) and EPT [38]. It disconnects the donors when the VM is terminated.

There are two distinct address spaces mapped to the physical memory: GPA and HVA as shown in Fig. 3. GPA is used to access the memory from the VM context, and the EPT maintains its mapping to the physical address space. For a memory access from an application in a VM, the guest virtual address must be translated to the physical address via two mapping tables. For a TLB miss, the hardware page table walker traverses the guest page table (GPT) to find the

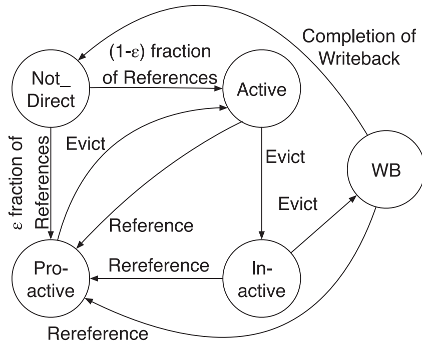

Fig. 4. State transition diagram of each block ($\epsilon = 0.1$).

corresponding GPA and then traverses EPT for the mapping between the GPA and host physical address (HPA). If no mapping is established or the access permission is not valid, an EPT violation occurs. For serving the memory accesses with a configurable block granularity, `dcm` manipulates the EPT. The HVA space is accessed from the QEMU process context to emulate I/O devices, the advanced programmable interrupt controller (APIC) and so forth. The HVA to HPA mapping is maintained with the host page table.

Page Descriptor. The states of each page is recorded and tracked by the page descriptor. The page descriptor is indexed by the guest physical page number. Each descriptor entry contains the state which will be discussed in the next section, and its location in the direct memory or indirect memory. In addition, the entry contains links used for the state queues and writeback operations.

Memory Block. Although a 4 KB page is the minimum unit of management in `dcm`, the actual management unit is a *block* whose size changes by the elastic block mechanism which will be discussed in Section 4. A block is a set of adjacent pages, and its size is always a power of two of the minimum 4 KB page size. In the page descriptor, the block size is encoded to identify the block each page belongs to, and the page states in the same block are always in the same state. In the rest of this paper, *block* is used as the management unit in `dcm`.

3.2 Direct Memory Management

To reduce costly remote memory accesses, memory blocks stored in the direct memory must be managed to retain the blocks which will be re-referenced in near future. In `dcm`, a modified LRU-3 replacement policy is adopted for managing the direct memory [39]. The direct memory is managed by using three different FIFO queues: proactive, active, and inactive queues. A victim from the proactive queue is evicted to the active queue, and a victim from the active queue is evicted to the inactive queue. The blocks in the inactive queue are still in the direct memory, but their mappings are removed from the EPT of the VM. In each queue, the head of the queue becomes the next victim, and a new block is inserted to the tail. When a block is evicted from the inactive region, it is written back to the remote memory. Note that accesses to the memory blocks in the proactive and active queues are not traced by `dcm`, as they do not incur any exception. Once blocks are evicted to the inactive queue, any references to the blocks can be detected, and the re-referenced blocks are promoted to the proactive queue.

With the three queues in the direct memory, as shown in Fig. 4, each block is in one of the following states:

NotDirect: The block is not touched yet or located in the remote donor node. The remote location of the block is maintained by `dcm` in the page descriptor.

Active: The block is either fetched from the indirect memory or newly touched on demand, and it is placed on the direct memory and mapped to either HVA, GPA, or both. A block transitioning to *Active* is moved to *Proactive* by a small random probability of ϵ. The random insertion to the proactive queue avoids a low utilization of the direct memory in case of sequential memory accesses.

Proactive: When a block is re-referenced while it is in *Inactive* or *Writeback*, the block is inserted into the proactive queue. The proactive queue contains the blocks with a known re-reference history, or by a random promotion as described in the aforementioned *Active* transition.

Inactive: The block still exists in the direct memory, but its mapping to HVA or GPA is disconnected. If the block is re-accessed, the mapping is restored with the local copy. Note that `dcm` detects memory accesses only through fault exceptions. The fault handling checks whether the block exists in the inactive region.

Writeback: The block is being transferred to an allocated remote block in a donor node. It is not in any of the three queues, but linked to a writeback descriptor. The fault handler will confirm the completion of network transfer at the next fault of the same vCPU.

`Dcm` does not use the periodic checking of access bits for maintaining the LRU chain of each queue [39]. Unlike the traditional swap system which infrequently demotes unaccessed pages to the slow swap disk, the disaggregated memory system moves significantly more pages back and forth between the direct and indirect memory repeatedly, to address the deficiency of local memory capacity all the time.

3.3 Latency Hiding with Overlapped Execution

To reduce the page reclamation overheads of the swap system, conventional mechanisms employ either batching schemes or dedicated servers. The batching increases the throughput of the page reclaimer by batch processing multiple pages at once. The dedicated server thread receives "memory pressure" signals from the memory requester thread, and then the server evicts multiple pages by speculating about the future demands. In common, the batching and dedicated server thread increases their reclamation efficiency by amortizing the cost by swapping out multiple pages. However, it is essential to keep the speculation balanced. Over-speculation hurts the performance of applications by reclaiming potentially useful pages early, and under-speculation increases the demand paging latencies by delaying the preparation of free pages.

In `dcm`, fine-grained memory reclamation is used to avoid the risk of bulk reclamation of the batching and dedicated server. However, to reduce the cost of fine-grained reclamation, `dcm` overlaps the page reclamation process with network transfers by leveraging the one-sided asynchronous transfer of RDMA operations. The writeback latency overlaps with the execution of the virtual machine

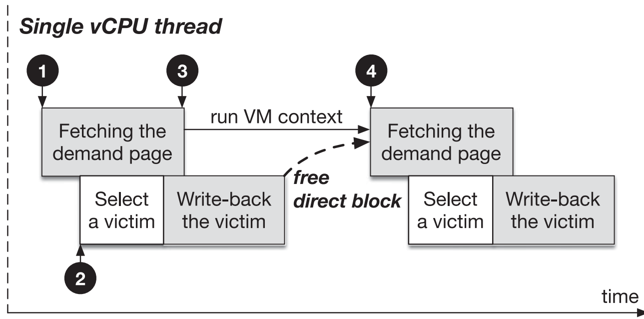

Fig. 5. Overlapped page reclaim and transfer: Grey rectangles are processed by the networks. The first fetching is initiated (circle 1), and the fetched page will be stored in a free page deallocated in the previous fault handling. When the fetch operation is completed (circle 3), the VM can run its context without waiting for the completion of writeback operations. While the first fetch operation is in processing, the CPU selects a victim and initiates a writeback (circle 2). The freed direct memory page by the writeback is used for the second fetching (circle 4).

context, and the fetching latency overlaps with the page reclamation as described in Fig. 5. This overlapping not only hides the writeback network latency but also improves vCPU utilization by performing victim selection while the vCPU waits for the completion of fetching. It does not need to reclaim memory pages speculatively, because it reclaims as many victim pages as the VM needs. The reduced overhead enables to apply a more sophisticated algorithm for victim selections, as the victim selection itself can be overlapped with a page fetching.

The reliable connection of RDMA guarantees the order of the writeback transfer operations to the indirect memory, eliminating the need for waiting its completion. For example, Fig. 6 shows how such ordering guarantee simplifies the overlapped operations. In the figure, vCPU0 selects page number 10 from the direct memory (*direct-10*) as a victim page and writes back the page to page number 200 in the indirect memory (*indirect-200*). During the transfer, vCPU1 incurs a demand page fault to the same direct memory page 10, which reuses the data directly from the page in the writeback state. As *direct-10* is reused during the writeback and exists in the direct memory, indirect-200 is no longer necessary. The next victim selection by vCPU2 uses the freed indirect memory page 200 to write back another page (*direct-11*). In the example, even though vCPU2 reuses the

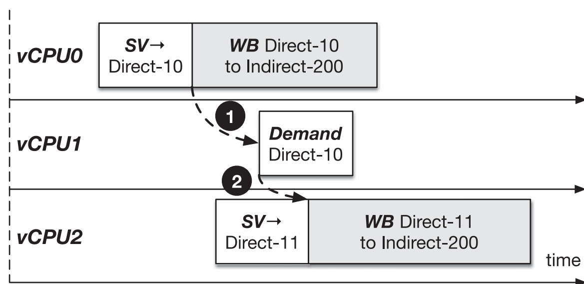

Fig. 6. Writeback ordering: SV, WB, and Demand denote *select victim*, *writeback*, and *demand* operations of the `dcm` respectively. A demand for a block (direct 10) occurs during the writeback of the block (circle 1), and the corresponding indirect block (indirect-200) is no longer necessary, even though the writeback is already initiated. Right after the release, the same indirect block is re-allocated for vCPU2 to write back another page (Direct-11 at circle 2). With the ordering support, the second writeback does not need to wait for the completion of the first writeback, even though the second one re-uses the indirect page 200.

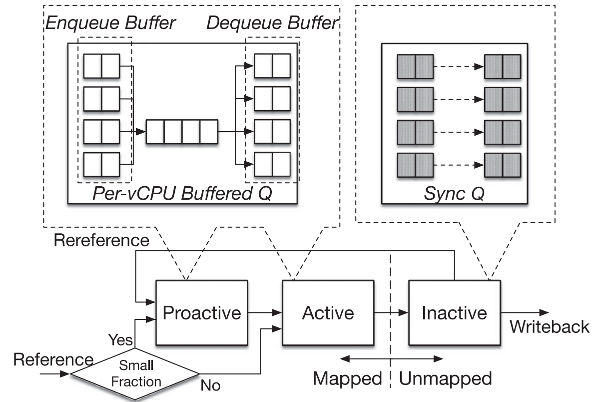

Fig. 7. Direct memory management for scalability by reducing contentions of the shared queues.

indirect memory 200, it does not check and wait for the completion of the prior writeback by vCPU0. The write ordering of RDMA guarantees that the second writeback to *indirect-200* occurs only after the completion of the first writeback, allowing the second writeback to safely overwrite the page.

3.4 Scalability Consideration

`Dcm` must be scalable enough to support multiple large scale virtual machines with many vCPUs in a system. To mitigate scalability bottlenecks, the internal data structures of `dcm` are designed to reduce conflicting accesses to the critical shared data. To minimize the contention on the critical shared data, the internal organization extensively employs per-vCPU data structures.

Scalable Memory State Management. The queue-based block state management discussed in Section 3.2 is implemented for scalability improvement. Fig. 7 describes the scalable implementation of the proactive, active, and inactive queues. For the proactive and active queues, per-vCPU buffers are used for enqueue and dequeue operations, which relieve the contention on the head and tail of the two shared queues. Between the per-vCPU buffers and shared queue, the insertion to and deletion from the queue are batch processed to amortize the locking overhead. Unlike the proactive and active queues which serve multiple vCPUs at the head and tail of the queues, any entries in the inactive queue can be updated by multiple vCPUs to serve page faults. To avoid the contention on a single global inactive queue, the inactive queue is organized as multiple per-vCPU synchronized queues. With the multi-queue organization, the inactive queue is not a global FIFO queue, but the approximate approach is good enough for the inactive state management.

Lockless Writeback Descriptor. The writeback implementation is also designed to avoid a global data structure. When a writeback operation is initiated by a vCPU by selecting a victim from the inactive sub-queue of the vCPU, a writeback descriptor is created and attached to the per-vCPU writeback (WB) head, as shown in Fig. 8. Each writeback descriptor has a link to the page descriptor associated to the writeback operation. Note that the direct memory freed by a writeback operation is used to serve future page faults, not the currently pending one, to avoid serializing writeback and fetching operations. The current pending fetch operation uses the direct memory block freed by a prior writeback operation.

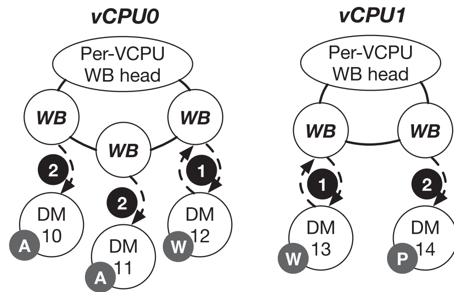

Fig. 8. Lockless writeback descriptor: Dark circles 1 and 2 represent a normal ownership and a canceled ownership respectively. Grey circle A, P, and W represent the active state, proactive state, and writeback state respectively. The direct memory (DM), whose state are not the writeback state, represents a canceled relationship because of re-reference during their writeback processes.

A block must be transitioned to the proactive state if it is re-referenced by another vCPU thread during the writeback operation. To allow such state transition for an in-flight writeback operation without acquiring a lock, `dcm` checks the ownership with pairwise double links between a writeback descriptor and page descriptor. Fig. 8 shows that the descriptors of writeback operations are linked to a per-vCPU structure, and each direct memory descriptor is doubly linked to the corresponding writeback descriptor.

If a block is re-referenced before the completion of the writeback operation by another vCPU thread, the page descriptor is updated to represent the canceled ownership by removing a link from the direct memory descriptor to a writeback descriptor. Fig. 8 shows the link removal in circle 2. The link from the writeback descriptor is preserved to mark the reuse status and prevent a data race. Subsequently, when the vCPU checks the writeback descriptor during the next fault processing, the writeback descriptor is simply discarded.

3.5 Fault Tolerance

Using remote memory donors makes the system vulnerable to failures in the remote nodes, and `dcm` shares the fault-tolerance issue of the disaggregated memory with the prior approaches. The conventional fault tolerance mechanism maintains redundant pages over several remote donors, as discussed by the prior approaches[5], [40], [41], [42]. Such redundancy provides a transparent fault handling to guest operating systems and applications, at the cost of additional indirect memory usage. To reduce the memory capacity overhead of remote nodes by redundancy, local storages can be used to asynchronously back up the memory content, as adopted by INFINISWAP [5].

Although `dcm` does not employ the redundancy-based fault tolerance in the current implementation, it improves the reliability by using the robustness of RDMA. `Dcm` uses a heartbeat-based fault detecting and migration policy in response to a software failure. RDMA data connections survive the software failures such as kernel crashes on the memory donor nodes. A failure does not affect the established RDMA data connection as long as the memory donor does not disconnect its connection. `Dcm` module periodically communicates to the entire memory donors and migrates the indirect blocks to another memory donors when a software crash of the memory donor system is detected. Investigating the incorporation of the additional redundancy-based mechanism is our future work.

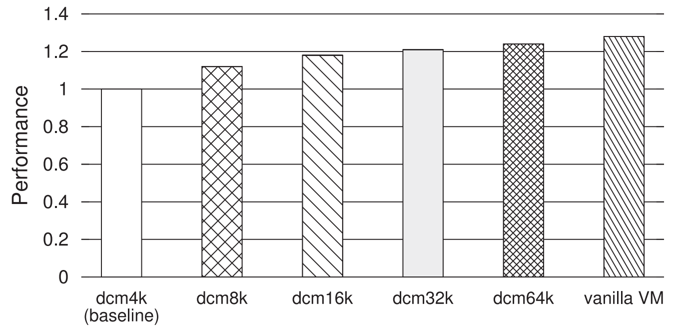

Fig. 9. Performance of a microbenchmark with sequential memory accesses normalized to dcm4k: dcm$X$k (x-axis) is a VM with 4GB direct memory and XKB block size.

4 ELASTIC BLOCK

As discussed in Section 2.2, fetching multiple memory pages in a single message can reduce the cost of remote data transfer in RDMA. However, without spatial locality, fetching a large chunk of memory pages for a single fault to the direct memory not only wastes the network bandwidth, but also pollutes the direct memory with unused pages. In this section, we discuss how to select the optimal migration granularity (*block size*) by tracing the spatial locality in different memory regions in VMs. The proposed hypervisor-integrated disaggregated memory facilitates the identification of spatial locality by maintaining the page fault records for each VM memory page.

A complicating factor in the hypervisor-based approach is that the hypervisor can trace the spatial locality only in the guest physical address space. The virtual-to-GPA mapping controlled by the guest operating system can potentially break applications' inter-page spatial locality when it is observed through GPA by the hypervisor. However, the application-level spatial locality is commonly preserved in the GPA for modest ranges for two reasons. First, the common use of the transparent huge page enabled by defaults from Linux 2.6.38 provides the contiguous mapping within the 2 MB large page. Second, the buddy memory allocation by the guest OS often maps contiguous guest physical pages to virtual pages [43], [44]. With the two factors, the application-level spatial locality up-to 64 KB block size, if it exists, is commonly preserved in the GPA.

4.1 The Effect of Block Size

To investigate the performance impact of the block size, we first evaluate a simple micro-benchmark. Fig. 9 shows the performance of a VM application which sequentially accesses 16 GB memory, when the block sizes are varied. In the figure, The dcm$X$k (x-axis) denotes a VM with 4 GB direct memory and XKB block size. The baseline is the dcm4k which uses a VM with 4 GB direct memory and 4 KB block size, and the rest of the memory is located in a remote node. A performance gap between the baseline and a vanilla VM, which uses only the direct memory as an ideal configuration, is large with 27.8 percent difference. The performance of `dcm` for the sequential access patterns improves significantly, as the block size

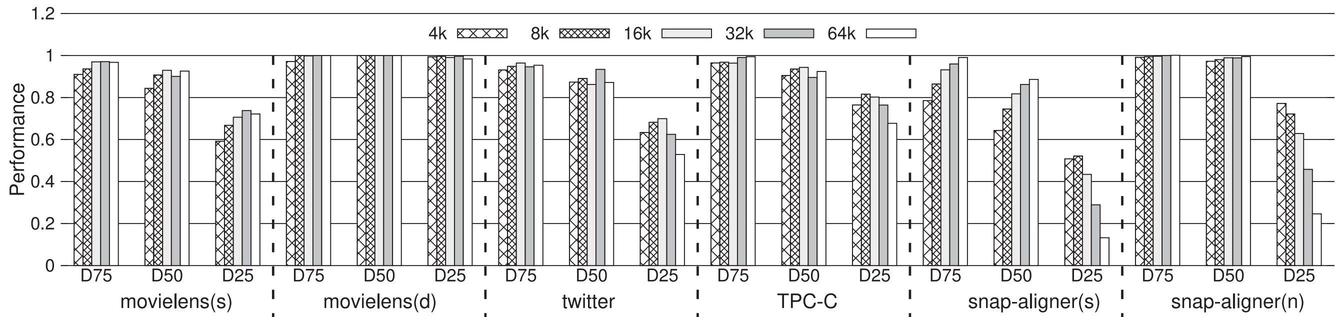

Fig. 10. Normalized performance of target workloads to the baseline with 4 KB-64 KB block sizes: DX denotes that $X/100$ fraction of the Resident Set Size (RSS) of each VM is located in the direct memory.

increases; the performance of dcm64k increases by 24 percent and shows only 3 percent slower performance than the ideal vanilla VM because the sequential memory access has strong spatial locality.

Fig. 10 shows the performance of workloads with five different block sizes and three direct memory ratios, which is normalized to the performance of the ideal vanilla VM. The experimental setup of the figure is described in Section 5.1. DX denotes $X/100$ fraction of the Resident Set Size (RSS) of each VM is allocated in the direct memory. The figure indicates that the performance of workloads is highly affected by the block size and the ratio of the direct memory. It discloses various characteristics depending on both the block size and the degree of pressure on the direct memory. `movielens_small`, `GraphX with twitter`, and `TPC-C on VoltDB` prefer 32 KB block, 8 KB block, and 8 KB block under D25, respectively. `movielens_large` does not show any preference. In addition, with different direct memory ratios, the best block size changes. For example, `snap-aligner(s)` prefers large block sizes under D75 as presented in Fig. 10. However, increasing the block size to 32 or 64 KB produces a negative effect on D25 configuration, which has only 25 percent direct memory compared to the total memory footprint. For the workload, block sizes larger than 8 KB under D50 exhibit better performance than 4 KB block size under D75, although D75 has 16 GB bigger direct memory than D50. The result indicates the importance of using a good block size, as the block size can be more important than the direct memory capacity for certain scenarios. `snap-aligner(n)` generally prefers a small block size.

However, the analysis with Fig. 10 does not account the access characteristics in different times and locations in the memory, as it fixes the block size for each run. Applications can have different best block sizes for different phases. Among the benchmark applications, `snap-aligner(s)` consists of two phase executions. The first stage loads the reference data from a large file and the second stage aligns the target sequence data with the reference data loaded in the first stage. Fig. 11 shows the different block size preference for the two phases of `snap-aligner(s)`. Two phases have opposite characteristics in the preference on the block size: the loading stage prefers large blocks, but the alignment stage prefers small blocks.

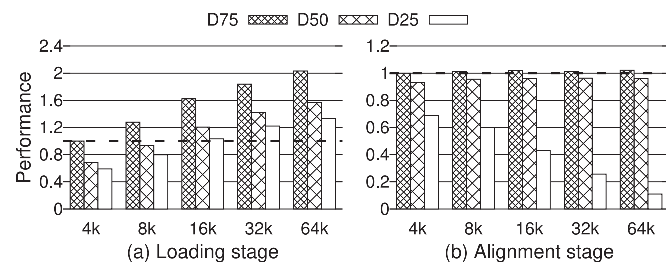

Fig. 11. Performance of two stages with snap-aligner(s): the loading stage prefers the 64KB block size, while the alignment stage prefers 4 KB when the direct memory is small (D25).

Based on the analysis in this section, the optimal block size depends on multiple factors, access patterns which can vary by execution phases or by memory regions, and available direct memory capacity. To accommodate such a dynamic nature of block size selection, a dynamic block adjustment mechanism is required to achieve the best performance improvement.

4.2 Dynamic Block Adjustment

It is challenging to decide the appropriate block size because the best block size depends on various conditions of each workload, access patterns affected by computation stages and memory regions, and the degree of memory pressure. Smaller block sizes than the best one may not extract potential performance of the workloads with high spatial locality. When the spatial locality is high, large block sizes can provide the prefetching effect as they can bring the nearby pages even before demand accesses to the pages occur. However, larger block sizes can waste network bandwidth and pollute the direct memory for applications with random memory access patterns.

To find the best block size, we define an elastic block that consists of multiple pages and dynamically changes its size of $2^{12+\beta}$KB, $0 \leq \beta \leq 4$, in this study. The first page of a block is the head page, and the rest of pages are members. Each page in a block is associated with its page descriptor along with the block size for the page. The descriptor also has a lock variable to control concurrent accesses to a block. We designed the block management using the same buddy algorithm as the buddy page in the memory management of operating systems. With the design, the elastic block mechanism selects an appropriate block size dynamically for each memory page, and thus even for a single VM, different regions of memory can have distinct block sizes.

The elastic block introduces two maintenance operations: Stretch and Reduce through which it is adapting to the memory access patterns of the workloads. The Stretch

operation is triggered when a related fault is raised, and the REDUCE operation is triggered just after completing a write-back of a block. Its management employs a conservatively stretch and aggressively reduce (CSAR) policy to cope with iterative sequential memory accesses and abrupt random memory accesses.

Algorithm 1. STRETCH Operation

```
1:  procedure STRETCH(GPA)                    ▷ guest physical address
2:      chead:= the head page descriptor of the block for a given
        GPA
3:      bhead:= the head page descriptor of the buddy block
4:      mhead:= the head page descriptor of the merged block
5:      if block size of chead = block size of bhead
6:  and bhead block exists in direct memory then
7:          acquire the lock of buddy block
8:          if buddy block is included in active list then
9:              move to the MRU (tail) position of active list
10:         end if
11:         copy the state of the buddy block into mhead
12:         for all member descriptors do
13:             increase the block size
14:         end for
15:         release the buddy block's lock
16:         return stretched
17:     end if
18:     return not_stretched
19: end procedure
```

STRETCH: As shown in Algorithm 1, a block is stretched when a page fault is handled, fetching pages from the indirect memory. If an adjacent block (*buddy block*) already exists in the direct memory and a new block neighboring with it enters the direct memory by the page fault, the two blocks becomes a candidate for stretch. To maintain the block size constraint of a power of two pages, a stretch requires that the buddy block has the same size as the new block. Since the new block is added to the direct memory, the entire stretched block is moved to the MRU (tail) position of the active list, if the buddy block was in the active list. During the stretch operation, the lock variables of the fault block and buddy block are acquired for synchronization before the block size information of each descriptor of block candidates is updated. As a conservative stretch policy, the stretch operation is performed only once for each fault handling.

REDUCE: As presented in Algorithm 2, a block is reduced completely to the minimum 4 KB page size during its eviction from the direct memory, if less than a half of pages in the block are referenced. The aggressive reduce quickly responds to the changes to random access patterns. The reference information, collected by checking the access bits of HPT and EPT whenever the pages are unmapped, is maintained, while the block exists in the direct memory. A reduce operation requires the lock variables of all member pages to update the block size.

The elastic block allows the system to choose an appropriate size of each block for the entire memory regions dynamically. It changes the block size according to the characteristics of memory regions in different time periods.

4.3 Synchronization of Elastic Operation

The stretch operation enlarges the block by merging the adjacent blocks, and the reduce operation reduces the coverage of the block. However, the two elastic operations for a block have a chance to be concurrently executed with fault handling for the target block of the operations. For example, the reduce operation is triggered when a block is evicted from the direct memory due to its insufficient spatial locality. At the same time, a demand for a member page may occur. The fault handler must identify the changing coverage of the target block and use a valid synchronization variable.

Algorithm 2. REDUCE Operation

```
1:  procedure REDUCE(block)                ▷ an elastic block
2:      if more than a half of pages in the block are referenced
        then
3:          return not_reduced
4:      end if
5:      acquire the locks of all the member descriptors
6:      for all member descriptors do
7:          reset the block size of member descriptor
8:          copy the state of the head to each member descriptor
9:      end for
10:     release the locks of all the member descriptors
11:     return reduced
12: end procedure
```

Algorithm 3. Locking Protocol

```
1:  procedure LOCK_BLOCK(block)                ▷ an elastic block
2:      loop
3:          bhead := the head descriptor of the block
4:          block_size := block_size of the block
5:          acquire the lock of bhead
6:          if block_size == block_size of the block then
7:              break                    ▷ lock of the block is acquired
8:          end if
9:          release the lock of bhead
10:         continue        ▷ the block is changed during spinning
11:     end loop
12: end procedure
13:
14: procedure UNLOCK_BLOCK(Block)              ▷ an elastic block
15:     bhead := the head descriptor of the block
16:     release the lock of bhead
17: end procedure
```

Fig. 12 shows an example of a synchronization between the reduce operation and fault handling for GPA 0×11000. vCPU1 decides to reduce the block with 16 KB size as marked by circle 1. Concurrently, vCPU0 accesses a member of the block at GPA 0×11000 during the reducing process, and waits for the completion of the operation by referencing the head descriptor at 0×10000 and storing the block size locally as represented by the circle 2. After the completion, vCPU0 notices the changed block size of the demand block by comparing the copied block size and the block size of the descriptor for the GPA 0×11000 as shown in the circle 2′. Upon detecting the change, vCPU0 uses a new head descriptor and proceeds the fault handling for GPA 0×11000 by the lock for GPA 0×11000. Algorithm 3

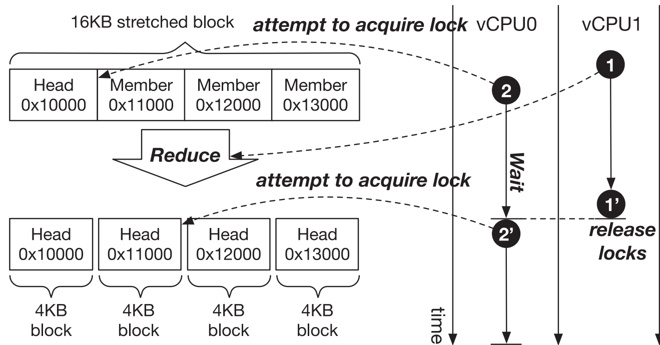

Fig. 12. Synchronization between REDUCE operation and demand fault handling: circle 1 is the reduce operation on a block at 0x10000, and circle 1' indicates lock release. Circle 2 and 2' are repeated lock acquire attempts for handling demand faults at 0x11000.

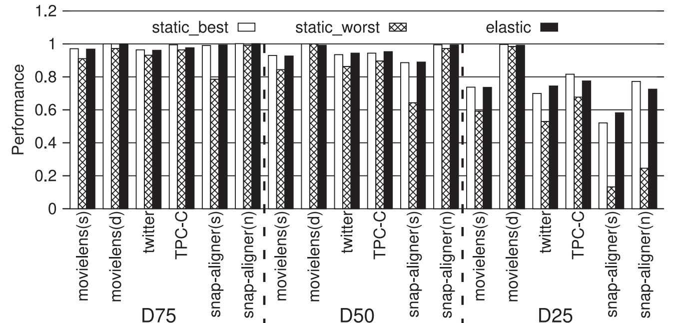

Fig. 13. dcm Performance with three direct memory configurations. Performance is normalized to the baseline. The static_best and static_worst are the best result and the worst result of the static blocks respectively.

shows the locking protocol as stated above. The LOCK_BLOCK acquires a lock variable of its head descriptor after confirming the validity of the block size, and the UNLOCK_BLOCK releases a lock variable of its head descriptor.

5 EXPERIMENTAL RESULTS

5.1 Methodology

The evaluation system consists of five physical nodes. A node hosting a big memory VM is directly connected to four donor nodes via InfiniBand FDR x4. The big memory VM node is equipped with two Intel Xeon E5-2670v3@2.3 GHz, DDR4 96 GB memory, and two Mellanox ConnectX-3 FDR dual-port host channel adapters (HCA). The Ubuntu 15.04, Linux Kernel 3.19, and Mellanox OpenFabrics Enterprise Distribution (OFED) 3.2 are installed to build an environment for the proposed dcm module. The donor nodes are equipped with two Intel Xeon E5-2650@2 GHz, 160 GB memory and Mellanox ConnectX-3 FDR single-port HCA. Ubuntu 17.04, Linux Kernel 4.10, and Mellanox OFED 4.0 are installed. Each memory exporter running on a donor node covers a consecutive 4 GB memory region, and each donor node exports 64 GB memory with 16 exporters. The big-memory VM is configured to have 16 vCPUs and 256 GB memory, and Ubuntu 16.04 is used as the guest OS. The transparent huge page (THP) of the guest OS is not disabled.

Table 1 shows the application workloads with their execution time/performance and memory requirements in our experiments. The resident set size is a physically resident memory also known as memory footprint of a VM for executing the target workload. We measure either the execution time or the performance of workloads running on a VM where the transparent huge page of the host operating system is turned off to support fine-grained page migration between the direct and indirect memory, and to isolate the performance effect and variance by huge pages. The current version of the Linux Kernel also splits the large page into small pages to swap out.

TABLE 1
Workloads: Performance and Resident Set Size (RSS)

Workload	Exec. time/Performance on vanilla VM	RSS of VM
movielens (small)	26.9 s	7,556 MB
movielens (default)	1,226.4 s	22,321 MB
twitter	355.9 s	17,334 MB
TPC-C	39749.1TPS	27,018 MB
snap-aligner (small)	215.3 s	63,556 MB
snap-aligner (normal)	1557.6 s	81,493 MB

To evaluate the proposed memory disaggregation system for the cloud computing, we chose four types of workloads to represent each area in the in-memory computing: in-memory analytics (movielens), graph analytics (Apache Spark-based graph analytics with twitter), in-memory database (TPC-C on VoltDB), and bioinformatics (snap-aligner) [45], [46], [47], [48], [49]. In addition to the basic four types, we complement movielens (small) and snap-aligner (small) analyzing a small size of input data. The movielen with Apache Spark runs a collaborative filtering algorithm in-memory on a dataset of user-movie ratings. The twitter running on Spark, is a graph analytics benchmark with large-scale datasets. Movielens and twitter are part of CloudSuite, a benchmark suites for cloud services[45], and they are configured as a single-node deployment. VoltDB is an in-memory database system for running the TPC-C workloads, and the snap-aligner is a genome sequence aligner.

We evaluate three scenarios with different direct memory ratios compared to the total RSS of the benchmark VM. The three direct memory ratios are 75, 50, and 25 percent of the ResidentSetSize of the VM, and they are denoted by D75, D50, and D25, respectively in this section.

5.2 Performance Evaluation

To evaluating the performance of the elastic block support, we compare the performance of each workload running on the elastic block-enabled VM to the best and the worst result of the static block size selection. Fig. 13 shows the result of the experiments and the y-axis value is normalized to the performance of the baseline which uses only the direct memory.

Most of the workloads with the D75 configuration result in the near direct-only performance because of the low pressure of the direct memory. The static_best and elastic block results show 98.5 and 98.4 percent of the performance of the baseline, and even the static_worst also shows 94.4 percent of the performance of the baseline. However, a performance degradation between the D75 and D50 is not significant because the guest OS keeps the page caches in its memory and the effect on the performance is not noticeable yet. In the D50 results, although only 50 percent of RSS is allocated in the direct memory for each workload, the performance

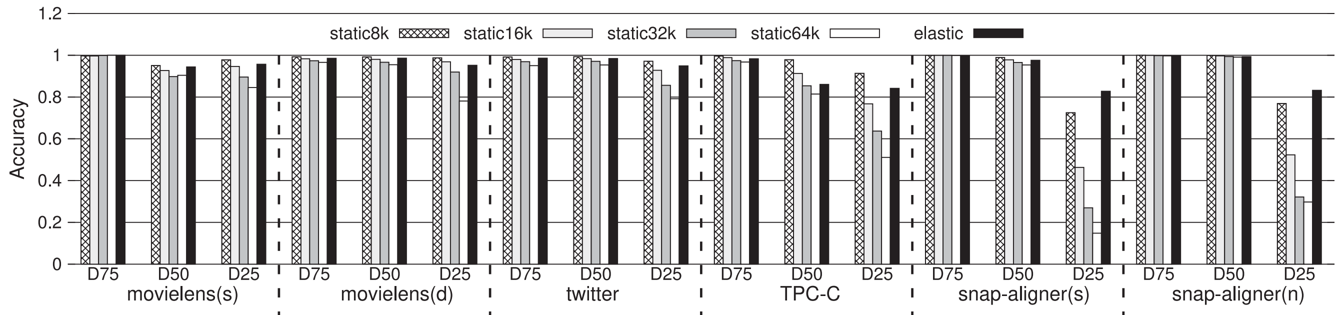

Fig. 14. Accuracy of the prefetched pages within each block.

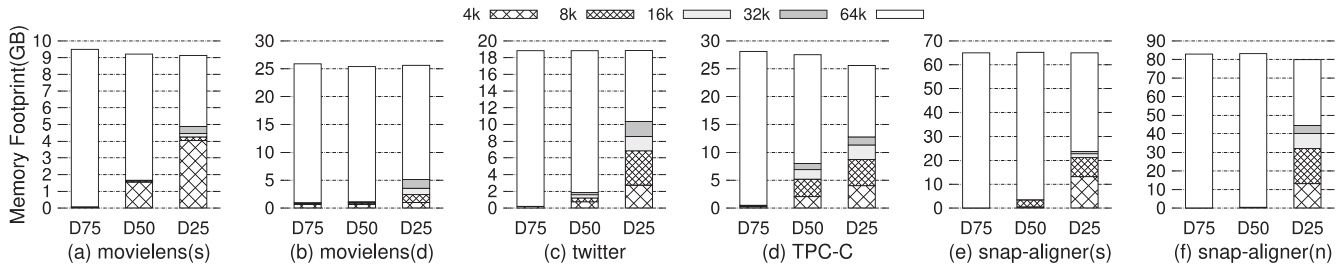

Fig. 15. Block composition of workloads with elastic block after an execution of the workloads.

degradation is within 10 percent, showing the potential of disaggregated memory. As the pressure on the direct memory increases significantly with 25 percent, the performance degradation is increased.

Although the performance gap between the static_best and the static_worst increases up to 50 percentage point under the D25, the performance gap between the static_best and the elastic block is less than 5 percentage point for all the cases. It proves that the proposed elastic block mechanism can effectively select the block size dynamically during the execution of various workloads.

The performance of the elastic block is directly related to the accuracy of the prefetching as shown in Fig. 14. The accuracy of the prefetching is defined as the ratio of the number of prefetched pages to the number of used pages. The overall accuracy of prefetching with the elastic block, under the D75 and D50, is 0.93, although the accuracy of 8, 16, 32, and 64 KB block is 0.94, 0.86, 0.80, and 0.75 respectively. These results show that the elastic block effectively exploits the spatial locality and provides a chance for improving the performance. In addition, twitter and snap-aligner with small input dataset outperform static blocks thanks to the effectiveness of the variable block size supported by the elastic block. The composition of the variable block of each configuration is shown in Fig. 15. Under the D75, the elastic block does not reduce most of the large pages, but a block size adapts to the pattern of memory access as the capacity of the direct memory decreases.

6 Discussion

Comparison to Conventional Swap with RAMdisk. To evaluate the effectiveness of dcm against the current swap component, we compare the performance of dcm against the Linux swap system with RAMdisk. A RAMdisk block device provides an ideal block device for the swap device, as it does not incur any I/O operations for swapping the memory pages from/to the swap device. With this RAMdisk setup, the performance depends on the efficiency and scalability of victim selection and page migration, as the indirect memory also resides in the same local memory. dcm is compared to the ideal memory-based swap system with multiple vCPUs for evaluating the scalability of dcm. Note that the indirect memory of dcm is still the remote memory in other donor nodes transferred through the networks, unlike the RAMdisk used for swap.

Fig. 16 shows that the execution time of snap-aligner(s) with various local memory capacities and numbers of vCPUs. In the x-axis, the configurations of local memory are 32, 24, 16, and 8 GB denoted as L32, L24, L16, and L8 respectively for the RAMdisk swap and dcm. For the swap runs, the direct memory capacity is controlled by using the cgroup of Linux. For each memory configuration, the number of vCPUs is varied from 8 vCPUs to 24 vCPUs. The leftmost vanilla configuration shows the execution time of snap-aligner(s) running in a VM entirely with only the direct memory. The y-axis shows their execution time of snap-aligner(s).

The figure shows that more vCPUs are allocated to the VM, the execution time decreases significantly for the workload when it runs in the vanilla VM. However, the memory-based swap does not support such scalability. As the local memory size decreases from L32 to L8, the swap system suffers from

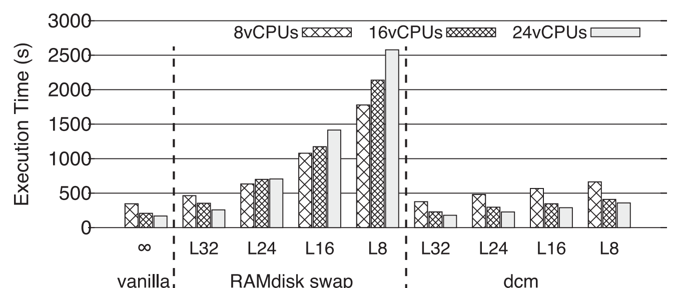

Fig. 16. RAMdisk swap versus dcm for snap-aligner(s): LX denotes a VM with XGB direct memory.

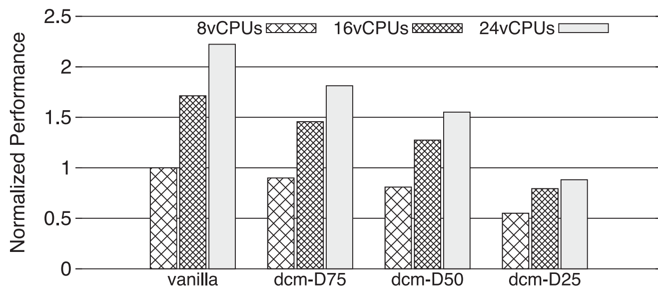

Fig. 17. Normalized performance of snap-aligner(s) running in dcm with Intel Optane SSD and 4 KB block.

significant increases of execution times. In addition, even if the number of vCPUs increases, the execution times are increased in L16 and L8. The results show that the current swap system cannot properly handle the frequent memory reclaim and fetch operations required for the disaggregated memory. However, by the scalable design of dcm, such as the per-vCPU data structures, scalable memory state management, and lockless writeback descriptor, dcm can provide relatively minor increases of execution times compared to the ideal vanilla VM. In addition, even if the direct memory size is the smallest 8GB, increasing the number of vCPUs still reduces the execution time significantly.

Emerging New Memory Technology. Recent advancements of non-volatile memory technologies can potentially extend the disaggregated memory to use the new memory as indirect memory, as the new memory technologies have significantly increased the bandwidth of the storage devices. The recent Intel Optane SSD supports 2,500 MB/s and 2,200 MB/s bandwidth for sequential reads and writes [50]. High-performance SSDs such as Samsung Z-SSD provide the bandwidth of the sequential read/write up to 3.2 GB/s [51]. To expand the memory capacity with the new technologies, Intel Memory Drive Technology exploits the high bandwidth Optane SSD as the second-level memory [52].

In this section, we discuss how dcm can be extended to use the new memory technologies. Fig. 17 shows the performance of snap-aligner(s) running in dcm with Intel Optane SSD DC P4800X instead of the remote memory. The performance is normalized to the performance of the vanilla VM with 8 vCPUs. Due to the reduced bandwidth of the SSD, the performance of dcm is modestly reduced. For D25 when the direct memory is insufficient, the performance degradation is 12%P compared to the performance with the prior remote memory, since the SSD bandwidth is still lower than the InfiniBand FDR which supports at most 56 Gbps. The elastic block management may reduce the performance degradation, but further optimizations are necessary. However, by exploiting the proposed scalable design, dcm can provide the scalability with more vCPUs for the snap-aligner(s) across different direct memory capacities, unlike the scalability problem of the memory-based swap presented in Fig. 16.

Application-Level Memory Sharing. Memory sharing at the application-level has been used to allow accesses to the remote memory in data processing framework such as Apache Spark. For example, Apache Ignite supports the sharing of data across multiple nodes in a Spark cluster [53]. Apache Ignite tightly coupled to Spark, stores data in the form of key-value pairs. On the other hand, the proposed system provides the memory of the VM as a more generic way than Ignite by managing the memory hierarchy by the hypervisor instead of the middleware. Dcm maintains the conventional memory abstraction for hierarchically composed memory architectures, and it transparently exploits temporal and spatial locality of applications unlike the middleware approach designed for a specific application.

7 Conclusion

This paper proposed a new memory disaggregation system backed by RDMA-supported high bandwidth networks. The proposed hypervisor-based design for disaggregated memory provides memory extension to the remote memory transparently to guest operating systems and applications. Its new design proposed a new replacement scheme, overlapped memory reclaim and network transfer, and scalability supports by per-vCPU data structures and lockless writeback operations. In addition, the elastic block maximizes the performance benefit of exploiting the spatial locality, as it dynamically adapts to changing access patterns. The experimental results showed that the disaggregated memory can provide on average 6 percent performance degradation compared to the ideal local-memory only machine, even though the direct memory capacity is only 50 percent of the total memory footprint. In addition, the proposed design provides scalable performance with increasing numbers of vCPUs. With the advent of high bandwidth non-volatile memory technologies, the proposed disaggregated memory will be able to expand its support for general hierarchical memory systems, such as conventional DRAM and new non-volatile memory. To prove its design flexibility, the paper also showed the preliminary performance evaluation with the new Optane SSD as the indirect memory.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) (No.2017-0-00051) and by the R&D program of NST (grant B551179-12-04-00) and ETRI R&D program (grant 18ZS1220). The grants were funded by the Ministry of Science, and ICT (MSIT), Korea. Jaehyuk Huh was supported by IITP (No.2017-0-00466) funded by MSIT, Korea.

References

[1] Google, "Google CloudPlatform machine types," [Online]. Available: https://cloud.google.com/compute/docs/machine-types/, 2018.

[2] Jeff Barr, "AWS news blog: Now available EC2 instances with 4 TB of memory," [Online]. Available: https://aws.amazon.com/ko/blogs/aws/now-available-ec2-instances-with-4-tb-of-memory/, 2017.

[3] Jeff Barr, "AWS news blog: EC2 in-memory processing update: Instances with 4 to 16 TB of memory + scale-out SAP HANA to 34 TB," [Online]. Available: https://aws.amazon.com/ko/blogs/aws/ec2-in-memory-processing-update-instances-with-4-to-16-tb-of-memory-scale-out-sap-hana-to-34-tb/, 2017.

[4] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *Proc. IEEE 18th Int. Symp. High-Perform. Comput. Archit.*, 2012, pp. 1–12.

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

[6] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," IETF RFC 5040, Oct. 2007.

[7] A. Acharya and S. Setia, "Availability and utility of idle memory in workstation clusters," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 1999, pp. 35–46.
[8] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel, "Nswap: A network swapping module for linux clusters," in *Proc. Eur. Conf. Parallel Process.*, 2003, pp. 1160–1169.
[9] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 267–278.
[10] P. S. Rao and G. Porter, "Is memory disaggregation feasible?: A case study with spark SQL," in *Proc. Symp. Archit. Netw. Commun. Syst.*, 2016, pp. 75–80.
[11] A. Samih, R. Wang, C. Maciocco, M. Kharbutli, and Y. Solihin, "Collaborative memories in clusters: Opportunities and challenges," in *Transactions on Computational Science XXII*, Berlin, Germany: Springer, 2014, pp. 17–41.
[12] A. Daglis, S. Novaković, E. Bugnion, B. Falsafi, and B. Grot, "Manycore network Interfaces for In-memory Rack-scale Computing," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 567–579.
[13] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 103–114.
[14] Mellanox, "ConnectX-3 single/dual-port adapter with VPI," [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family-=119, 2013.
[15] Mellanox, "ConnectX-4 single/dual-port adapter supporting 100Gb/s with VPI," [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family-=201, 2014.
[16] Jens Axboe, "Flexible I/O tester," [Online]. Available: https://github.com/axboe/fio, 2016.
[17] Intel, "Intel xeon scalable processors," [Online]. Available: https://newsroom.intel.com/press-kits/next-generation-xeon-processor-family/, Last Accessed on: Jun. 11, 2018.
[18] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets, "Cashmere-VLM: Remote memory paging for software distributed shared memory," in *Proc. 13th Int. Parallel Process. Symp. 10th Symp. Parallel Distrib. Process.*, Apr. 1999, pp. 153–159.
[19] E. P. Markatos and G. Dramitinos, "Implementation of a reliable remote memory pager," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 1996, pp. 15–15.
[20] G. Dramitinos and E. P. Markatos,"Adaptive and reliable paging to remote main memory," *J. Parallel Distrib. Comput.*, vol. 58, no. 3, pp. 357–388, 1999.
[21] G. Bernard and S. Hamma, "Remote memory paging in networks of workstations," in *Proc. SUUG Int. Conf. Open Syst.: Solutions Open Word*, 1994.
[22] E. A. Anderson and J. M. Neefe, "An exploration of network RAM," EECS Department, Univ. California, Berkeley, Tech. Rep. UCB/CSD-98-1000, Dec. 1994.
[23] S. Liang, R. Noronha, and D. K. Panda, "Exploiting remote memory in InfiniBand Clusters using a High Performance Network Block Device (HPBD)," The Ohio State Univ., Columbus, OH, Tech. Rep. OSU-CISRC-5/05-TR36, 2005.
[24] Accelio, "Accelio based network block device," [Online]. Available: https://github.com/accelio/NBDX, 2014.
[25] D. Minturn and J. Metz, "Under the hood with NVMe over Fabrics," [Online]. Available: https://www.snia.org/sites/default/files/ESF/NVMe_Under_Hood_12_15_Final2.pdf, 2015.
[26] G. Sims, "All about Linux swap space," [Online]. Available: https://www.linux.com/news/all-about-linux-swap-space, 2007.
[27] J. Corbet, "Making swapping scalable," [Online]. Available: https://lwn.net/Articles/704478/, 2016.
[28] M. Lespinasse, "V2: Idle page tracking / working set estimation," [Online]. Available: https://lwn.net/Articles/460762, 2011.
[29] V. Davydov, "idle memory tracking," [Online]. Available: https://lwn.net/Articles/643578/, 2015.
[30] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy, "Speculative paging for future NVM storage," in *Proc. Int. Symp. Memory Syst.*, 2017, pp. 399–410.
[31] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Operating Syst. Rev.*, vol. 36, pp. 181–194, Dec. 2002.
[32] N. Amit, D. Tsafrir, and A. Schuster, "VSwapper: A memory swapper for virtualized environments," in *Proc. 19th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 349–366.
[33] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 225–230.
[34] D. Ye, A. Pavuluri, C. A. Waldspurger, B. Tsang, B. Rychlik, and S. Woo, "Prototyping a hybrid main memory using a virtual machine monitor," in *Proc. IEEE Int. Conf. Comput. Des.*, Oct. 2008, pp. 272–279.
[35] M. R. Hines and K. Gopalan, "MemX: Supporting large memory workloads in xen virtual machines," in *Proc. 2nd Int. Workshop Virtualization Technol. Distrib. Comput.*, 2007, pp. 2:1–2:8.
[36] U. Deshpande, B. Wang, S. Haque, M. Hines, and K. Gopalan, "MemX: Virtualization of cluster-wide memory," in *Proc. 39th Int. Conf. Parallel Process.*, Sep. 2010, pp. 663–672.
[37] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.
[38] Intel, "Intel64 and IA-32 architectures software developers manuals," [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm, 2016.
[39] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *SIGMOD Rec.*, vol. 22, no. 2, pp. 297–306, Jun. 1993.
[40] M. Stumm and S. Zhou, "Fault tolerant distributed shared memory algorithms," in *Proc. 2nd IEEE Symp. Parallel Distrib. Process.*, Dec. 1990, pp. 719–724.
[41] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote memory in the age of fast networks," in *Proc. Symp. Cloud Comput.*, 2017, pp. 121–127.
[42] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 29–43.
[43] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, Feb. 2014, pp. 558–567.
[44] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2012, pp. 258–269.
[45] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Notices*, vol. 47, no. 4, pp. 37–48, Mar. 2012.
[46] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2016, pp. 122–132.
[47] M. Stonebraker and A. Weisberg, "The VoltDB Main Memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, Jun. 2013.
[48] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler, "Faster and more accurate sequence alignment with SNAP," *CoRR*, vol. abs/1111.5572, 2011.
[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.
[50] Intel, "INTEL OPTANE SSD DC P4800X SERIES," [Online]. Available: https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html, 2017.
[51] Samsung, "Ultra-low latency with samsung Z-NAND SSD," [Online]. Available: https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf, 2017.
[52] Intel, "Intel memory drive technology," [Online]. Available: https://www.intel.com/content/www/us/en/software/intel-memory-drive-technology.html, 2018.
[53] Apache, "Apache ignite - The apache software foundation!" [Online]. Available: https://ignite.apache.org, 2015.

Kwangwon Koh received the master's degree in computer science from Yonsei University. He has worked on several projects such as virtual machine monitor for ARM architecture, super-computing system for genome processing, and scalable operating system for many-core processors. He currently focuses on the software stack of a multi-tier memory system. His research interests include system software for parallel computing, virtualization, and multi-tier memory system.

Kangho Kim received the MS degree from Kyungpook University. He currently focuses on the software stack of a multi-tier memory system. His research interests include system software for parallel computing, virtualization, and multi-tier memory system.

Seunghyub Jeon received the MS degree from Korea University. He currently focuses on the operating system scalability in manycore system. His research interests include system software for parallel computing, virtualization, and multi-tier memory system.

Jaehyuk Huh received the BS degree in computer science from Seoul National University, and the MS and PhD degrees in computer science from the University of Texas, Austin. He is an associate professor of computer science at KAIST. His research interests include computer architecture, parallel computing, virtualization and system security. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.