2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

Memory Disaggregation: Research Problems and Opportunities

Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang+, Juhyun Bae, Yanzhao Wu

Geoegia Institute of Technology, Atlanta, GA30329, USA
+IBM Thomas J. Watson Research, New York, USA

Abstract—Memory usage imbalance has been consistently observed in many virtualized Clouds and production datacenters. Such temporal memory utilization variance is a major root cause for excessive paging and thrashing on virtual servers even though there are sufficient idle memory on the same node or in the Cloud cluster. Memory disaggregation is an emerging research and development endeavor towards addressing these memory usage imbalance problems. This paper first defines and characterizes the concept of memory disaggregation, and discusses the demands and challenges of efficient memory disaggregation in cloud datacenters. It then examines some promising research issues, design choices and directions to overcome some of the challenges posed by memory disaggregation. Specifically, it proposes two major new research challenges and solution directions for enabling elastic, on-demand disaggregated memory orchestration: (1) *virtual server memory and node level memory co-design* and (2) *local memory and remote memory co-design*. A brief description of two ongoing research projects is provided for both solution directions. The paper ends with a brief discussion of other advanced and emerging memory and storage technologies and potential opportunities for memory disaggregation.

Keywords—Cloud computing, disaggregated memory, shared memory, remote memory, virtualization

I. Introduction

Big data systems and analytics continue to fuel the in-memory data processing research and development. On one hand, to meet the growing demand of in-memory data processing and increase the competitive edge in service provisioning, Cloud providers have started to plan and upgrade their Infrastructure as a Service (IaaS) to include large memory virtual machines with terabytes (TBs) of memory in their high-end offerings [1,2]. On the other hand, memory utilization imbalance and temporal memory usage variations are frequently observed and reported in virtualized clouds [3-11] and production datacenters [12-16]. The memory upgrade trend further exaggerates the memory utilization imbalance problems.

Memory usage imbalance: opportunity and challenge. Two primary factors may cause large temporal memory usage variance at node level and severe memory utilization imbalance across nodes in a cluster. First, Cloud systems typically serve heterogeneous guest application workloads in their clusters of physical nodes (machines). Second, many guest applications have heterogeneous data access patterns during runtime, represented by the big data powered iterative machine learning (ML) workloads. Third, virtual machines (VMs), containers or Java Virtual Machine (JVM) executors are the three types of *virtual servers* popularly used in many Cloud application deployment models. Each virtual server (VM/container/JVM executor) is configured to accommodate the peek memory usage (worst-case demand) either at the initialization of the cluster or the initialization of each VM/container/executor. The accurate estimation and provisioning of server memory resources has been a long-standing technical challenge since the working set size is not always easy to predict and the DRAM allocation for each VM/container/executor has to be determined at the compute cluster installation time [14-16]. It is reported [11-12] that the clusters experience severe memory utilization imbalance with an average of 30% idle memory during 70% of the running time, and of the 80% memory allocated, only 50% on average is used. [17] shows that the aggregated memory capacity of a data center cluster reached 437 TB on a typical workday, which is only 69% of its overall allocated memory capacity. Clearly, the heterogeneity of guest applications and the peek-usage-estimation based allocation of per-node memory will result in a pool of idle (unused) memory spaces across the cluster, which presents substantial *opportunities*. At the same time, such availability of free memory may fluctuate in an unpredictable manner over the lifetime of the cluster or the lifetime of per virtual server, which also presents important *challenges* for disaggregated memory orchestration.

Impact of memory utilization on server performance. The memory resource is a vital performance bottleneck in data-intensive computing system. Datacenters progressively leverage virtualization to increase the resource utilization efficiency. Data intensive, latency-demanding applications enjoy high throughput and low latency if they are served entirely from memory. However, actual estimation and memory allocation are difficult [18]. When these applications cannot fully fit their working sets in real memory of their VMs/containers/executors, they suffer hefty performance loss due to excess page faults. Even when idle memory is present in other VMs, containers or JVM executors on the same node or remote nodes in the same cluster, these applications are unable to share those unused host and remote memory.

In this paper, we first describe the existing efforts for addressing large memory workloads. Then we define and characterize the concept of memory disaggregation, as well as the demands and challenges of efficient memory disaggregation in cloud datacenters. We next examine some promising research issues, design choices and directions to overcome challenges posed by memory disaggregation at either node-level or cluster-level. Specifically, we describe

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00165

two major new research challenges and solution directions for enabling elastic, on-demand disaggregated memory orchestration: (1) *virtual server memory and node level memory co-design* and (2) *local and remote memory co-design*. We briefly describe our ongoing research results in both directions. Finally, we discuss other advanced and emerging memory and storage technologies (e.g., NVM, SSD) and potential opportunities.

II. Related Work

Existing effort for handling large memory workloads can be broadly classified into three categories: (i) Estimation of working set size for peak usage memory resource allocation, (ii) memory-resident databases and key-value caching, and (iii) increasing effective memory capacity.

A. Estimation for Peak Usage Memory Allocation

There are two inherent problems with the techniques for estimating VM/application's working set size. First, accurate memory allocation is hard as peak memory variations happen under different application types, workload inputs, data characteristics and traffic patterns. Second, applications often over-estimate their requirements or attempt to allocate for peak memory usage [3,7,8,12], resulting in severely imbalanced memory usage across virtual servers (e.g., VMs, containers or executors), and underutilization on the local host node and remote nodes across the cluster [11,18].

B. Memory-resident databases and caching

To overcome the disk I/O bottleneck, memory resident databases and caching techniques have been proposed. Facebook [19] caches the results of frequent database queries using Memcached. This line of efforts by design embraces the vision [20] that main memory will be viewed as secondary storage and secondary caches to processors. However, most of the in-memory systems to date rely on OS virtual memory swap facility to keep the memory footprint of those data that cannot fit in memory on swap disk provisioned by OS. Thus, when an application or VM cannot fit their working set in memory, they suffer from slow disk I/O for OS paging events, as shown in Figure1(a). Several technologies are designed to allow the NIC to send data directly to end-host applications, bypassing the OS, such as Intel DPDK, Open vSwitch, VMware's vNetwork distributed switch. These technologies remove the inherent overheads due to traditional interrupt driven OS-level packet processing. With DPDK and Single Root I/O Virtualization (SR-IOV), multiple VMs can access the NIC via a virtual function, namely a per-VM virtual PCI-based NIC.

C. Increasing effective memory capacity

In recent years, proposals for increasing effective memory capacity have been put forward to promote the allocation of global memory resource shared by all servers to increase their effective memory capacities. These proposals promote new architectures and new hardware design for memory disaggregation [17], or new programming models [25]. But they lack of desired transparency at OS, network stack, or application level, hindering their practical applicability. Recent efforts represented by Accelio, NBDX and Infiniswap [26] exploit the Remote Direct Memory Access (RDMA) networks to leverage the disk-network latency gap. Examples are mainly using remote memory for certain workloads, such as remote storage for key-value stores, swap pages [34-37], RDD caching [38-40]. Most of these efforts lack of desired transparency and none addresses the opportunity and benefit of unused host memory sharing.

From developers' perspective, domain-specific *out-of-core* computation techniques [28] have been implemented to juggle I/O and computation. Moreover, many large-scale web applications [19] require access to disparate parts of massive datasets for serving each user request while exhibiting little spatial locality. Amazon produces a single Web page by processing hundreds of internal requests, such multi-tier software architecture for application serving accumulate and aggregate the I/O latency at each tier. Given that the cost-per-GB of DRAM increases non-linearly, it may not be viable for many to simply buy or upgrade to specialized, large-memory machines [1,2,29], which are too expensive to acquire and manage in a cost-effective manner due to the hard problem of accurate estimation of VM's working set size.

III. Memory Disaggregation: Characterization

Memory disaggregation decouples physical memory allocated to virtual servers (e.g., VMs/containers/executors) at their initialization time from the runtime management of the memory. The decoupling aims at allowing the server under high memory pressure to use the idle memory either from other servers hosted on the same physical node (node level memory disaggregation) or from remote nodes in the same cluster (cluster level memory disaggregation).

Node-level v.s. Cluster-level Memory Disaggregation. In the context of *node-level memory disaggregation*, a portion of the memory of all the servers in a computer node (VM/container/JVM executor) is transparently exposed as a single shared memory pool to all the applications running on the physical node (host). Similarly, the *cluster-level memory disaggregation* refers to the capability that the memory of all the nodes in a computer cluster is transparently exposed as a single cluster-level shared memory pool for the discretion by all the applications running on the same compute cluster.

By the principle of virtualization, VMs or containers or JVM executors hosted on the same node are viewed by their applications as independent virtual servers. Thus, disaggregated memory at either node-level or cluster level can be viewed as remote idle memory. However, memory shared among virtual servers hosted on the same physical node can be accessed at the DRAM speed if we can leverage shared memory mechanisms to implement node level memory disaggregation through the coordination between the node shared memory manager and its client agent for each of

the virtual servers. This allows us to enable access to node level disaggregated memory at the DRAM speed instead of the network I/O speed. As long as the gap between DRAM access speed and network I/O speed exists and such performance gap is non-negligible, we argue that it is significantly beneficial to optimize the implementation of disaggregated memory at node level using node-coordinated shared memory mechanisms, and implement disaggregated memory at cluster level using the high throughput and low latency interconnection of servers, e.g., RDMA networks.

Full v.s. Partial Memory Disaggregation. We can further categorize memory disaggregation capabilities into full memory disaggregation and partial memory disaggregation. Technologically speaking, *full memory disaggregation* is not yet feasible today for a number of reasons. First, DRAM memory had two main attributes that affect server performance: memory capacity and memory speed. Second, computer processors requires extremely fast access to memory. Based on the state of art networking technologies, local memory speed remains much faster than the network I/O. Third, if a system is running slowly due to the lack of local DRAM memory, and the processor can read data from local memory or remote memory much faster than from an external hard drive, then adding more memory or using memory disaggregation presents an opportunistic solution. This is because when a system runs short of DRAM memory allocation, it must swap the overflowed data to the hard drive, which can significantly slow down the system performance. Thus, full memory disaggregation at cluster level will be feasible when remote memory access speed is comparable to local memory speed.

Partial memory disaggregation refers to the capability of supporting memory disaggregation only for a selection of memory utilities. This applies to both local memory at node-level and remote memory at cluster level. For example, there have been several research projects on disaggregated memory over the last decade, from computer architecture [21-23], computer networks [24-27], key-value systems. Most of these efforts use disaggregated remote memory when a server experiences shortage of its allocated memory.

Partial memory disaggregation at cluster level refers to the capability of using idle memory on the remote nodes in the same cluster. The rapid advance in high speed networks of 10/40/100 Gbps, compound with the Remote Direct Memory Access (RDMA) technologies, such as InfiniBand, RoCE (RDMA over Converged Ethernet), is the key enabling technologies for efficient remote memory disaggregation, thanks to high bandwidth and low latency interconnection of servers in a RDMA cluster. By partial, we mean that if the processor exceeds its memory capacity, and if it can write data to (and read data from) remote memory much faster than an external hard drive, then using cluster-level memory disaggregation presents a viable opportunity for boosting application performance. Existing research has shown that memory swapping and key-value based memory caching are the two killer applications for partial memory disaggregation.

Partial memory disaggregation at node level refers to the capability of allowing a server (VM, container or JVM executor) to meet its transient high memory pressure by utilizing the idle memory from other VMs, containers, or JVM executors on the same physical node (host). For example, when a server can no longer fit its full working set in memory, it must swap out some memory page to the hard drive, which can cause applications to experience significant performance degradation, even when there is idle memory on other servers co-hosted on the same node. One example of using partial memory disaggregation at node level is to provide shared memory capability to allow swapping memory pages to node-level coordinated shared memory regions first before swapping to the hard drive.

IV. Disaggregated Memory System

In this section, we will discuss the system design objectives, the general architecture for disaggregated memory systems, including the interaction between memory disaggregation at node level and memory disaggregation at cluster level, as well as the alternative design decision choices.

A. System Design Objectives

In all general computing systems, the computing resources are managed and tuned for all applications internally at either the operating system kernel level or at the middleware level, such as Spark, Hadoop, NoSQL systems. We argue that a disaggregated memory system should be design to meet the following objectives. First, the memory disaggregation infrastructure should be made available without requiring applications to be aware of when and where the disaggregated memory is being used. Put differently, the use of disaggregated memory should be made transparent to all the applications running on any node of the computing cluster, regardless whether they are running on a VM, a container or a JVM executor. Applications do not need to know the exact location of the disaggregated memory. Second, memory disaggregation should be supported at either the OS level or the middleware level, and it should not require any involvement of applications. Third, the OS and the middleware should shield applications from the complexity of memory disaggregation at both node level and cluster level. Fourth, the use of disaggregated memory should be made transparent to both the applications and their runtime environment such as the middleware and the guest OS. In other words, VM/container/executor that uses disaggregated memory should be able to run on unmodified applications, unmodified middleware and unmodified guest OS.

B. General System Architecture

A disaggregated memory system is a distributed system composed by node level disaggregated memory facilities and

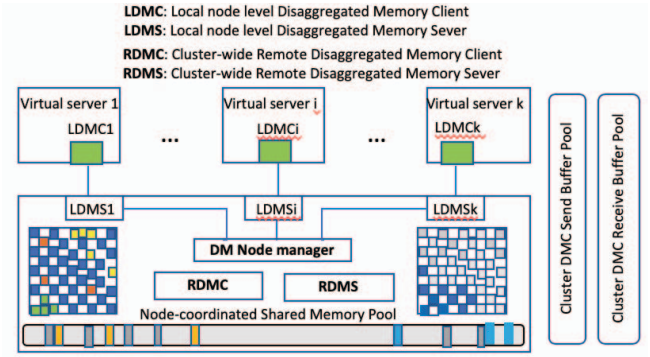

Fig. 1. Functional components of disaggregated memory system (per node view)

cluster level disaggregated memory facilities. Given a compute cluster of N virtual servers, we will have n physical nodes with $n \leq N$, hosting these N virtual servers. Each node can participate in the disaggregated memory system in two roles at any given time: as a server by donating its idle memory and as a client by using idle memory from other node. Each node maintains three types disaggregated memory pools: the shared memory pool coordinated by the node manager and two cluster wide disaggregated memory buffer pools: send buffer pool and receive buffer pool. Figure 1 shows the main functional components provided at each node for supporting disaggregated memory at node level and cluster level.

Memory disaggregation at node level. A typical node in a Cloud cluster may host multiple virtual servers (VMs, containers, or JVM executors). One widely adopted approach to memory resource management is to allocate equal amount of memory capacity to all virtual servers on a single node, based on the estimated peak time demand at the cluster initialization time. Given that most of applications are running on a cluster of virtual servers, we assume without loss of generality that applications can rent a disaggregated memory enabled cluster. Thus, N virtual servers of this cluster and their corresponding physical nodes will be the participants of a disaggregated memory system. Thus, each physical node in the cluster agrees to donate some portion of its physical DRAM by registering some memory reserved for RDMA network. This registered memory region will be used for maintaining two types of disaggregated memory pools at each node: the send buffer pool and the receive buffer pool. Also, the N virtual servers agree to donate a system-defined x% of their allocated memory as the shared memory resource, which can be used on demand by any of these N virtual servers. Note that it is possible that some virtual servers hosted on the same physical node may not belong to this cluster and thus will not be the participants in the disaggregated memory system.

When a virtual server needs to use additional memory from the disaggregated memory pool, such as experiencing page fault [34], or performing in-memory caching of RDD in Spark system or caching of key-value data in Memcached, the local disaggregated memory client (LDMC) running on the virtual server will send a put request to its corresponding node level disaggregated memory sever (LDMS), which will check if it has sufficient space in the node coordinated shared memory pool to serve this put request. If not, this LDMS will interact with the node manager to obtain additional free memory slabs in the shared memory. When the put operation is completed, the disaggregated memory page table maintained by the node manager together with LDMS is also updated. If there is insufficient free memory in the shared memory pool, the node manager will communicate with the remote disaggregated memory client (RDMC), which select the remote node(s) in the same group for memory disaggregation. The put request from the virtual server will now be served by the chosen remote node through the RDMC module running on the local node, as shown in Figure 1.

Memory disaggregation at cluster level. Consider a simplified scenario: Let node A and node B are any two nodes in the cluster. When a virtual server on node A needs to expand its local memory to the node-coordinated disaggregated memory, if node A decides to select node B as the remote disaggregated memory to meet the memory demand of this virtual server, then upon receiving the data entry from the virtual server, node A will place the data in its cluster-wide disaggregated memory (DM) send buffer pool. In this case, node A serves as the cluster-wide disaggregated memory client (RDMC) and node B will serve as its cluster-level remote disaggregated memory server (RDMS). Node A sends the data entry via RDMA write operation to the node B's remote disaggregated memory receive buffer pool. When the virtual server later needs to read the data parked on node B, it will send a read request with the data entry ID to its local disaggregated memory server (LDMS) on node A, using its disaggregated memory map, node A knows the data entry is residing at node B, and thus issues an RDMA read request to node B. Node B will locate and get the data to the disaggregated memory receive buffer on node A. Figure 2 illustrates how node A uses the disaggregated memory donated by node B.

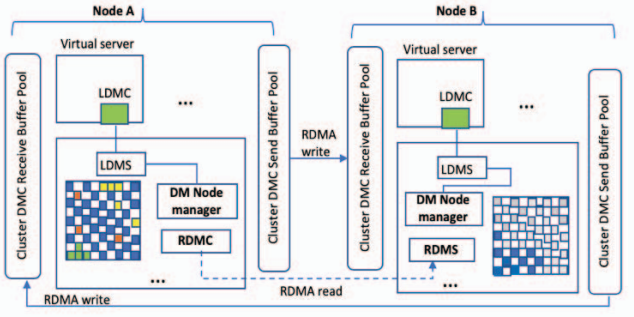

Fig. 2. Accessing disaggregated memory between two nodes in a cluster

There are a number of important design decisions that need to be made carefully to preserve the commonly desired

system properties, including the performance, scalability, reliability, memory balancing, memory registration and eviction management, connect, correctness and consistency management of the disaggregated memory system. We will discuss these issues and the alternative design choices in the subsequent sections.

C. Scalability Design Choices

A fundamental challenge for providing cluster-wide memory disaggregation is to scale the system to terabytes of collective memory in a cluster. However, to track where each data entry (such as the swap-out page, the cache partition) is located in the cluster, both the local and remote disaggregated memory server modules (LDMS and RDMS) corresponding to a virtual server (e.g., VM, container or JVM executor) needs to maintain the metadata such as the virtual server ID, the remote node ID and offset in the disaggregated memory map for this data entry, indicating whether the data entry is parked in the node-coordinated shared memory, the local RDMA registered memory buffer pool, or on the remote node. Each virtual server on the node needs to maintain such a memory map, which is used to store the location of each of its data entries in the cluster. This memory map is critical for the virtual server to locate and track the whereabout of its data entries in the entire cluster. However, using a single memory map as such does not scale well for large-size clusters.

Consider a simple in-memory hash table to implement our disaggregated memory map, and assuming that each entry is 4KB in size, and each location identifier metadata is 8 bytes, then we will need 5 GB at each node to maintain the hash table for the cluster-wide disaggregated memory of 2 TB, and 25 GB hash table per node for sharing 10 TB disaggregated memory. Maintaining a hash table of such size for each virtual server will incur prohibitively high cost to the local memory of each node in the cluster. For the cluster with large number of nodes, even though the cluster-wide disaggregated memory is increasing, but this solution approach fails to scale up.

One common approach to address the scalability challenge of large size clusters is to adopt a group-based coordination for disaggregated memory sharing. For example, the hierarchical group sharing model allows the partitioning of all nodes in a cluster into groups of smaller sizes and each with similar number of nodes. Nodes within each group can form a disaggregated memory sharing community. Nodes from one group cannot share the disaggregated memory pools of another group directly. One way to extend the flat structure of the group based sharing model is to introduce two or more tiers of hierarchical grouping algorithms. Each group in each tier will elect a group leader to synchronize the coordination among the server nodes in the group. A leader election protocol [30] periodically elects the one that meets certain constraints as the leader of a group, such as the one with the maximum available memory. If the leader node crashes (handshake time-out), a new leader election process will be triggered. Also, a leader can request dynamic re-grouping when its group experiences shortage of disaggregated memory.

D. Supporting Fault Tolerance

When the local disaggregated memory client (LDMC) on a virtual server is sending a write or read request to its corresponding local disaggregated memory sever (LDMS), it may encounter a number of failure scenarios.

*Local node or virtual server failure***.** If the corresponding virtual server or the host machine fails, by design the disaggregated memory system should provide the same failure semantics as the situation in which no disaggregated memory is supported. For instance, consider the virtual server is a VM, if we use the disaggregated memory system for providing the fast swapping service to the applications running on this VM, the disaggregated memory system should provide the same failure resilience capability as the guest OS swap facility today. Similarly, if the virtual server is a JVM executor on a Spark cluster, then upon the node failure or the executor failure, the disaggregated memory system for RDD caching should provide the same failure resilience semantics as the vanilla Spark.

Network connection, remote node or virtual server failure. Similarly, in addition to the node manager, each node also has the remote disaggregated memory client (RDMC) and the remote disaggregated memory server (RDMS). The former is sending a write or read request to a remote node, and the latter is receiving a read or write request from a remote node. It is important to handle unexpected failure scenarios due to network connection (link) failure, or remote virtual server failure, or remote node failures.

One can use centralized coordination or decentralized coordination for managing network connection failure, remote node or virtual server failures. The advantage of using decentralized approach is obvious. By not requiring central coordination, we can avoid single point of failure and frequent message synchronization. In addition, if we replicate each remote data write or read operation to or from more than one remote nodes, then we can significantly reduce the inconsistency caused by connection failure or node failure. For example, triple replica modularity has been provided for maintaining high reliability for Hadoop File System (HDFS). We can offer the same degree of fault tolerance by enforcing triple replica modularity for all remote read and write operations. Finally, each remote write or read operation is treated as an atomic transaction, all or nothing, and it is recorded in the corresponding entry of the disaggregated memory map maintained by the LDMS on the node, thus removing the inconsistency due to remote connection failure and unreachable server induced failure.

E. Memory Balancing and Eviction Management

Given a set of remote nodes that offer memory disaggregation services, when a write service request is sent by a node (say A), the node manager of A will need to select one primary node and two other replica nodes to serve this request. By

consulting with the leader node, one can identify a subset of remote nodes that are candidates for such selection. Several algorithms can be employed to minimize memory imbalance across nodes in a cluster (or a group), such as random, round robin (RR), weighted RR, or power of two choices [31].

F. Disaggregated Memory Registration and Eviction

To support disaggregated memory at node level and cluster level, we need to reserve certain amount of memory based on the assumption that comparing with the option of allocating all the available memory on each node to all the virtual servers hosted on the node, reserving some memory resource for supporting disaggregated memory can greatly improve the performance of applications running on each node and the cluster.

The shared memory pool maintained at each node is composed of a fraction of allocated memory from each virtual server. It could be 10% initially and proactively increase to 40% or reduce to zero. It is a configurable system parameter defined by each user of the disaggregated memory system. Similarly, we can also proactively allocate memory slabs of a given size and registers them as memory regions for RDMA operations on remote servers.

Remote idle memory is monitored and when it drops below certain threshold, remote memory slabs will be deregistered preemptively through the remote slab eviction handler, and updates the respective disaggregated memory maps maintained on the nodes corresponding to the deregistered slabs. At the same time, new remote memory servers will be selected to host the evicted pages in order to maintain the triple replica of the data entries hosted in the remote disaggregated memory.

To best serve the applications running on a virtual server, the following memory management policies are recommended:

(1) If there are frequent requests to remote disaggregated memory in the cluster, then it indicates that the node does not have sufficient memory resource to serve its active virtual servers. In this case, it is recommended to evict some memory slabs from the RDMA receive buffer pool, which reduces the proportion of the physical DRAM on this node being used for serving as remote disaggregated memory.

(2) If a virtual server on some physical node is observed to request disaggregated memory at node level or cluster level frequently over a period of certain threshold, then it is recommended to balloon more DRAM memory to this virtual server by evicting some memory slab(s) from the node coordinated shared memory pool or the RDMA send buffer pool. This will consequently reduce the data overflow to the disaggregated memory. Also based on our experiences with using disaggregated memory system for memory paging [34] and for RDD caching [38], maximizing the shared memory pool will provide higher throughput and lower latency for both big data and machine learning workloads. We share some of these results in Section V.

G. Connection and Consistency Management

RDMA is the most preferred networking technology for disaggregated memory systems to date. RDMA is characterized by high throughput, low latency, high messaging rate, low CPU utilization, low memory bus contention, message boundaries preservation and asynchronous operation. Three concrete implementations of RDMA technologies are (i) InfiniBand (used by about 50% of top 500 supercomputers), ranging from SDR 4x – 8Gbps, DDR 4x – 16 Gbps, QDR 4x – 32 Gbps, FDR 4x – 54 Gbps; (ii) RoCE – RDMA over converged Ethernet with 10 Gbps ~ 40 Gbps; and (iii) iWrap (Internet Wide Area RDMA protocol) with 10 Gbps. InfiniBand specification is written in terms of verbs (syntax for functions, structures and types, etc.). Open Fabrics Alliance (OFA) Verbs API is an effort to unify InfiniBand market with implementations of OFA Verbs for Linux, FreeBSD, Window. Although TCP and RDMA both use the client-server model and require a connection for reliable transport, TCP provides a reliable in-order sequence of bytes and RDMA provides a reliable in-order sequence of messages. Unlike TCP/IP, RDMA offers **zero copy** (data transferred directly from virtual memory on one node to virtual memory on another node), **kernel by pass** (no OS involvement during data transfer), and **asynchronous operation** (threads not blocked during I/O transfers). In summary, the RDMA access model provides (1) messages, preserving user's message boundaries; (2) Asychronous, no blocking during a transfer; (3) 1-sided (unpaired) and 2-sided (paired) transfers; and (4) no data copying into system buffers, as it ensures that order and timing of send() and receive() are relevant and memory involved in transfer is untouchable between start and completion of transfer.

Connection Management. To implement a disaggregated memory system, we can use one-sided RDMA write/read operations for data plane activities and RDMA send/receive operations for control plane activities. For each individual connection, two types of channels are established: RDMA channel for maintaining the network connection and data transfer, and the disaggregated memory system channel for interacting with the remote node agent, maintaining the system status, and performing placement and eviction algorithms.

Consistency. For each virtual server (e.g., VM, container, or JVM executor), the disaggregated memory system should maintain a memory map which serves as a log table to track of where a data entry is in the disaggregated memory system. It can be at the node coordinated shared memory, or remote memory of three remote nodes in the cluster, or an external secondary storage (when no sufficient remote memory is available to host the entire data entry). For each RDMA channel, we can configure RC QP to guarantee that messages are delivered from a requester to a responder at most once as well as in order without correction, which avoids data

corruption during transfer.

H. Optimizations

We discuss three types of optimizations that are helpful in our disaggregated memory systems for in-memory swapping and for in-memory caching Spark RDD partitions.

Memory page compression. In FastSwrap [34-37], four granularity of page compressions are used to compress 4KB pages: 512 B, 1 KB, 2 KB, and 4 KB. Figure 3 shows the compression ratio for 10 machine learning (ML) workloads using 2 compression granularity (2 page sizes) and using 4 compression granularity (4 different page sizes) in FastSwap compared to Zswap [32], a compressed RAM cache for disk based swap devices.

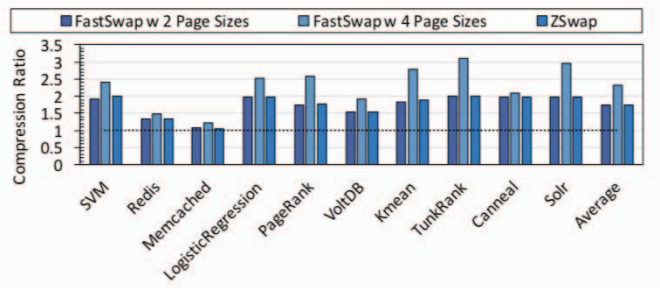

Fig. 3. Compression Ratio for 10 ML Workloads in FastSwap

Figure 4 shows the impact of 4 memory compressibility ratios on application performance for logistic regression workload in terms of completion time when 50% of the working set can fit into memory. Figure 4(a) and Figure 4(b) show the impact of compression when swapping-out least recent pages to the remote memory v.s. to the disk respectively when the shared memory pool is full on the local node.

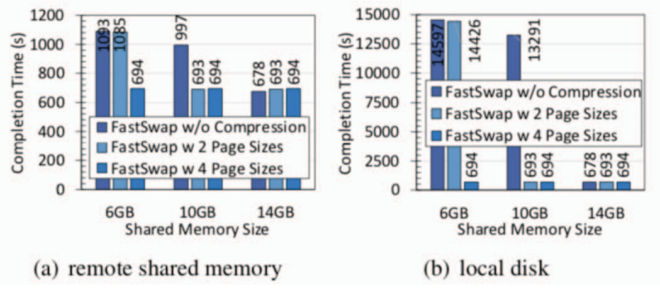

Fig. 4. Effect of compression ratio on remote memory and local disk

Figure 5 shows the impact of disaggregated memory compression on application performance. These experiments clearly show the advantage of memory page compression in a disaggregated memory system.

Window based batch access to disaggregated memory. Window based batchng can be effective for efficient access to disaggregated memory at node level (i.e., the shared memory buffer pool) and disaggregated memory at cluster level (i.e., the remote RDMA send buffer pool and receive buffer pool). For example, we implement a disaggregated memory system for in-memory RDD partition caching, called DAHI [37-38], on top of Accelio, an open source RPC library on top of RDMA. In the first prototype of DAHI, we use the default message size of 8 KB in Accelio for RDMA read and write operations between two nodes by using a window size d for batching d number of 8 KB messages in each RDMA transfer. We found that such batch is much more effective than per 8 KB messaging, when we need to cache large size of RDD partitions, ranging from 1 MB, 10 MB to 1 GB.

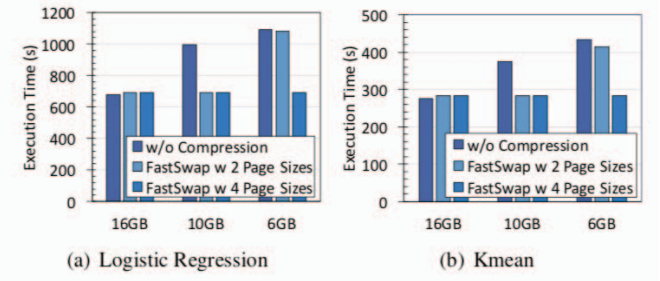

Fig. 5. Disaggregated memory compression on application performance

Similarly, we implement our disaggregated memory system for in-memory swapping in FastSwap [34-35]. By providing batching in both swapping out and swapping in, we show the performance improvement of our disaggregated memory system compared with no batching optimization. Figure 6 shows the completion time measurement comparison on four systems for 4 sizes of disaggregated memory workloads: FastSwap with proactive batch swap-in (PBS), FastSwap without PBS, Infiniswap, a RDMA based remote memory paging system [26], and Linux disk swapping. This experiment shows the obvious advantage of using batch swap-in.

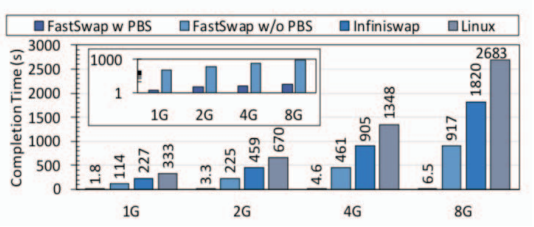

Fig. 6. Effect of compression ratio on remote memory and local disk

The current implementation of FastSwap is on top of NBDX, which is a block I/O device with 4 KB default message size. Based on our experience of implementing the disaggregated memory system for in-memory RDD caching directly on top of Accelio RPC library, which provides default message size of 8 KB and the max message size up to 1 MB, it is worth to experiment window based message batching with both different window size *d* and different message size *m*.

V. Experiments and Observations

In this section, ten popular memory-intensive applications are used for evaluation (see Table 1). The working sets for the

ten applications range from 25GB to 30GB and their input dataset sizes range from 12GB to 20GB per virtual server. Experiments are performed on a 32-machine, 56 Gbps Infiniband cluster. Each machine has 32 core E5-2650v2 CPU, 64 GB memory, 2TB SATA 7.2K rpm hard drives, and running KVM 1.2.0 with QEMU 2.0.0 as virtualization platform. We use Linux 4.1.0 and Ubuntu 14.04 for both the guest and host system. For most of the experiments unless otherwise stated, we run 80 VMs on a 32-machine RDMA cluster and created an equal number of VMs for each application workload.

Workload	Suite/Application	Dataset
PageRank	Spark GraphX	1 million pages
LogisticRegression	Spark Mllib	7.5 million samples
TunkRank	PowerGraph	30 million vertices
Kmean	PowerGraph	7 million samples
SVM	Liblinear	45 thousand samples
YCSB-Memcached	Memcached	20 million records
YCSB-Redis	Redis	20 million records
YCSB-VoltDB	VoltDB	20 million records
Canneal	PARSEC	2.5 million records
Apache Solr	Cloudsuite	12GB index

Table 3. Applications used in Experiments

A. Disaggregated Memory System for In-memory swapping

We use FastSwap [35], our in-memory swapping system for virtual machines, as the hybrid disaggregated memory system (both node-level and cluster-level). We compare it with Linux disk swapping and Infiniswap, a remote memory swapping system built on top of NBDX (a RDMA based block I/O device). We first compare the three systems in terms of completion time. Figure 7 shows the comparison results. The experiment measures the FastSwap performance on PageRank, LogisticRegression, TunkRank, Kmean, and SVM and compares them to those using Infiniswap and Linux. For 75% configuration (75% working set fits in memory), FastSwap improves application completion time by 24x on average and up to 83x over Linux, improves over Infiniswap by 2.3x average. For 50% configuration, FastSwap improves application completion time by 45x on average and up to 85x over Linux, improves over Infiniswap by 4.4x (best case) and 2.6x on average.

Next, we compare the performance impact of varying the disaggregated memory distribution at node-level and cluster level on application performance for all four systems: Linux, Infiniswap, NBDX and Fastswap. Using 50% configuration, we have 50% of working set sent to external disaggregated memory via paging. For FastSwap, FS-SM denotes 100% of paging events are handled in node-level shared memory pool. FS-RDMA denotes 100% using remote memory via RDMA network, and zero node level shared memory is reserved by FastSwap. The remaining three node-level to cluster-level distribution ratios are FS-9:1, FS-7:3, and FS-5:5. FS-9:1 denotes that 90% of swapping traffic is served in node-coordinated shared memory pool and 10% is sent to the remote memory pool.

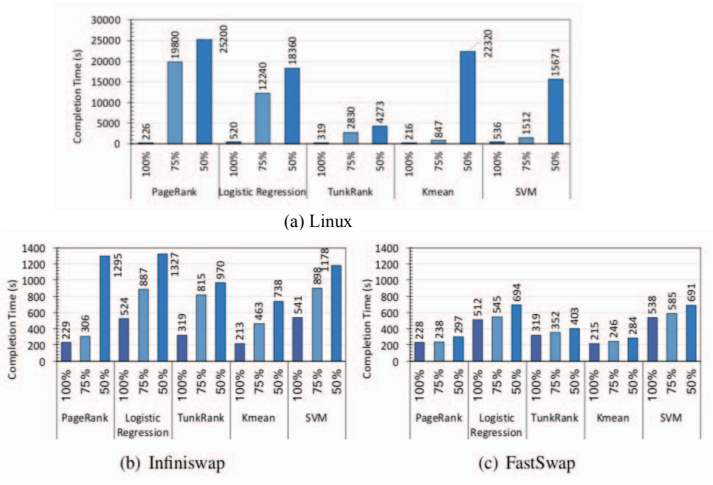

Fig. 7. Machine Learning Workloads Comparison

Figure 8 shows the throughput results. We highlight three interesting observations. First, using FS-SM, throughputs of Redis, Memcached and VoltDB increase by up to 571x, 171x, and 240x respectively, compared with Linux, increase by 11.4x, 5.1x, and 2.0x compared with Infiniswap, and increase by 10.5x, 4.9x, and 1.8x compared with NBDX.

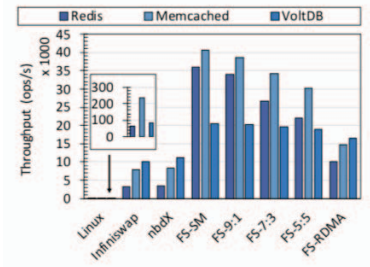

Fig. 8. Varying distribution ratio of disaggregated memory access

Second, using FS-RDMA, throughput of Redis, Memcached and VoltDB increase by 3.2x, 1.8x, and 1.6x respectively, compared to Infiniswap, and increase by 2.9x, 1.8x, and 1.5x respectively, compared to NBDX. Third, as the percentage of remote memory increases in the swapping-out operations, ranging from FS-SM, FS-9:1, FS-7:3, FS-5:5 to FS-RDMA, throughputs of all three applications drop accordingly. Figure 9 shows the throughput measurement of 300 seconds for Memcached ETC workload. We observe that using *FastSwap* with PBS (proactive batch swap-in), Memcached throughput performance quickly recovers to its optimal performance, which is about 50-thousand ops/sec. Using *FastSwap* w/o PBS, it takes more than 150 seconds for Memcached to recover to its best performance. In comparison, Infiniswap will take more than 2x longer than *FastSwap* w/o PBS. Also using Infiniswap with 300 seconds measurement, Memcached only recovers to 60% of its best performance at the end of measurement.

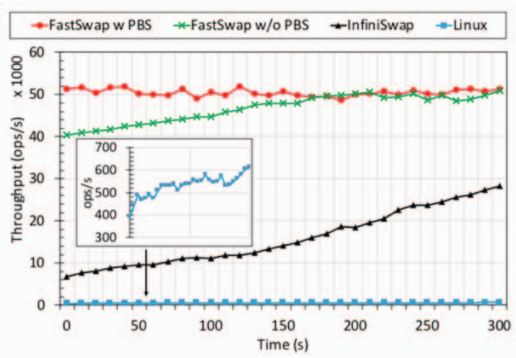

Fig. 9. Performance of Memcached (ETC, 50% configuration

B. *Disaggregated Memory for In-memory RDD Caching*

In this section, we show the experiments of the second disaggregated memory system – DAHI [37-38], which we developed for in-memory caching of Spark RDD partitions.

Spark is a well-known data processing platform for memory-intensive applications. Resilient Distributed Datasets (RDDs) [33] is a trademark of Spark. An RDD is an *abstraction* representing a set of immutable objects. By partitioning a large RDD into multiple partitions, we can distribute the large dataset over the cluster for parallel processing. RDDs can be created either from the data on the stable storage or by transformation from other RDDs through Spark operations, such as map, filter, join and so forth. Spark makes use of RDD to enhance memory utilization. For example, RDDs to be generated/used between consecutive map operations are kept in memory. RDDs to be reused can also be explicitly cached in memory. These enhancements improve application performance by reducing disk access, minimizing inter-process communication between Spark executors, which also makes Spark greedy on memory usage. When Spark can fit 100% of the working set in memory, its applications can enjoy the peak time performance. However, as soon as the working set cannot fully fit in memory, the application running on Spark executors experiences server performance degradation, causing imbalanced memory utilization in Spark executors and premature spilling, even though there are idle memory present in Spark executors [38].

DAHI is developed as a disaggregated memory system for providing node-level shared memory pool and cluster-level remote memory for in-memory caching RDD partitions. Figure 10 shows the evaluates and compares the effect of partial RDD caching on DAHI with vanilla Spark. This set of experiments are performed using three different categories of input datasets: small, medium, and large. For all applications, using *small* category dataset, the RDDs generated can be cached fully in memory, while using *medium* and *large* category datasets exhibit partial caching, as some partitions of the RDDs do not fit in memory.

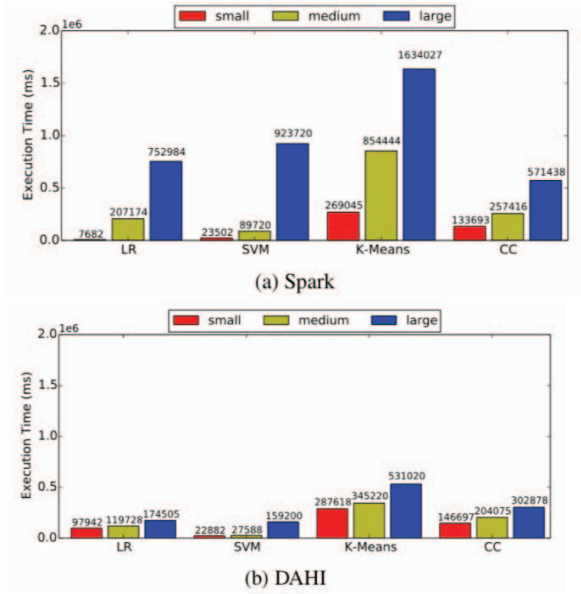

Fig. 10. Vanilla Spark v.s. DAHI powered Spark [38]

It is observed that, using DAHI, the completion time of LR obtains 1.7x and 4.3x speedup over vanilla Spark for *medium* and *large* datasets respectively. The speedup with respect to *medium* and *large* datasets for SVM is 3.3x and 5.8x, for K-Means is 2.5x and 3.1x, and for CC is 1.3x and 1.9x. This set of experiments shows the benefit of exploiting disaggregated memory orchestration across executors through node level and cluster level coordination via off-heap caching of the RDD partitions that cannot fit in their executors' memory.

VI. Discussion

Traditional memory hierarchy consists of on-chip memory (SRAM) with latency (cycles) of 1~30, off-chip (DRAM) memory with latency of 100~300, Solid State Disk with latency (25,000 ~ 2,000,000), to HDD with latency larger than 5,000,000.

Emerging non-volatile memory (NVM) technologies, such as Phase Change Memory (PCM), Intel 3D Xpoint memory array, are making rapid advancement, though no-single NVM technology dominates in terms of both energy and efficiency (capacity and latency). At the same time, high throughput, low latency network technologies, such as RDMA and its family of technologies, are being deployed in many Cloud datacenters.

These exciting technological trends have fueled the new forms of convergence of memory, networking and storage. Memory disaggregation is one step towards leveraging the latency gap between network I/O and storage I/O to enable DRAM memory to expand to the faster tier(s) in the memory hierarchy before resorting to the slower external storage tier. Many promising research problems are emerging, including identify the killer applications for different types of memory technologies and for different combination of memory, networking, and storage technologies, to name a few.

VII. Conclusion

We have described the problems and the challenges of memory usage imbalance in virtualized Clouds and discussed why efficient memory disaggregation can be a feasible solution to the problems. We define and characterize the concept of memory disaggregation, and discuss the challenges of efficient memory disaggregation in cloud datacenters. We have examined some research issues, design choices and directions to overcome challenges posed by memory disaggregation at both node-level and cluster-level. Specifically, we have proposed two major new research challenges and solution directions for enabling elastic, on-demand disaggregated memory orchestration: (1) *virtual server memory and node level memory co-design* and (2) *local memory and remote memory co-design*. We illustrate these two solution directions by presenting a brief description and the experimental results of the two ongoing research projects. We conclude with a brief discussion on other advanced or emerging memory and storage technologies (e.g., NVM, SSD) and potential opportunities for memory disaggregation.

Acknowledgment

The source code and datasets of some of the works we have discussed are available under open source software license at https://github.com/git-disl/ [95-99]. This work is partially sponsored by NSF grants 1564097, 1547102, 1115375, 1230740, IBM faculty awards during 2011-2017.

References

[1] Amazon EC2 High Memory Instances. [Online] https://aws.amazon.com/ec2/instance-types/high-memory/

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

[3] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[4] A.Ghodsi, M.Zaharia, B.Hindman, A.Konwinski, S.Shenker and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. *NSDI*, 2011.

[5] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. *EuroSys*, 2013.

[6] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *SIGCOMM*, 2014.

[7] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *CACM* 2010.

[8] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *CloudCom*, 2011.

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *SIGCOMM*, 2010.

[10] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Sa- tria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *SoCC*, 2016.

[11] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[12] C. A. Reiss. *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, UC Berkeley, 2016.

[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *SoCC*, 2013.

[14] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E.Tune and J. Wilkes. Large-scale cluster management at google with borg. In *EuroSys*, 2015.

[15] J. Hwang, A. Uppal, T. Wood, and H. H. Huang. Mortar: Filling the Gaps in Data Center Memory. ACM VEE 2014.

[16] 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.

[17] 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.

[18] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: exploiting page shar- ing for smart colocation in virtualized data centers. In *ACM SIGOPS Operating Systems Review*, 2009.

[19] Scaling memcached at Facebook, *http://www.facebook.com/note.php? note id=39391378919*.

[20] J. Gray and F. Putzolu, "The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time," *SIGMOD Rec.*, 16(3), pp. 395–398, 1987.

[21] HP: The Machine. http://www.labs.hpe.com/research/themachine.

[22] K.Lim,J.Chang,T.Mudge,P.Ranganathan,S.K.Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *ISCA*, 2009.

[23] K.Lim,Y.Turner,J.R.Santos,A.AuYoung,J.Chang,P.Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. *HPCA*, 2012.

[24] 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. *SoCC* 2017.

[25] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for linux clusters. *European Conference on Parallel Processing*, 2003.

[26] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. *NSDI*, 2017.

[27] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. *OSDI*, 2016.

[28] J. S. Vitter, "External memory algorithms and data structures: dealing with massive data," *ACM Comput. Surv.*, vol. 33, no. 2, pp. 209–271, 2001.

[29] Cray Inc., Cray XT4 and XT3 Datasheet http://www.cray.com/downloads/cray xt4 datasheet.pdf [online]

[30] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. *Usenix ATC* 2010.

[31] A. W. Richa, M. Mitzenmacher, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 2001.

[32] S Jennings. The zswap compressed swap cache, 2013.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mc-Cauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*12.

[34] Wenqi Cao and Ling Liu. ``Dynamic and Transparent Memory Sharing for Accelerating Big Data Analytics Workloads in Virtualized Cloud", Proceedings of IEEE 2018 International Conference on Big Data. Seattle, USA, Dec. 10-13, 2018

[35] FastSwap, https://github.com/git-disl/FastSwap

[36] XMemPod, https://github.com/git-disl/XMemPod

[37] Semih Sahin, Wenqi Cao, Ling Liu. "Host and Remote Caching of Spark RDD", Technical Report. Dec. 2018.

[38] DAHI, https://github.com/git-disl/DAHI