Dynamic and Transparent Memory Sharing for Accelerating Big Data Analytics Workloads in Virtualized Cloud

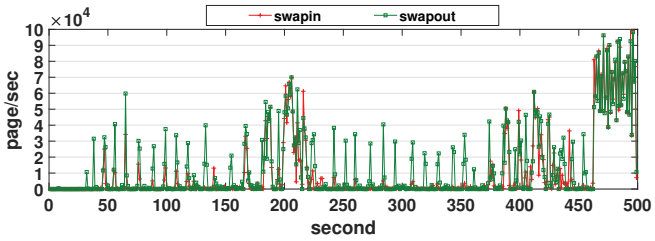
Wenqi Cao, Ling Liu
School of Computer Science, College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
Email: wcao39@cc.gatech.edu, ling.liu@cc.gatech.edu

***Abstract*—Many big data applications are memory-intensive workloads and perform iterative analytics algorithms. When the dataset used in each iteration of the analytic job exceeds the physical memory of their allocation, this type of workloads suffers from serious performance degradation or experience out of memory error. Existing proposals focus on estimating working set size for accurate resource allocation of executors, but lack of desired efficiency and transparency. This paper presents an efficient shared-memory based memory paging service, called *FastSwap*. The design of *FastSwap* makes a number of original contributions. First, *FastSwap* improves VM memory swapping performance by leveraging idle host memory and redirecting the VM swapping traffic to the host-guest compressed shared memory swap area. Second, *FastSwap* develops a compressed swap page table as an efficient index structure to provide high utilization of shared memory swap area through supporting multi-granularity of compression factors. Third, *FastSwap* provides hybrid swap-out and proactive swap-in to further improve the performance of shared memory swapping. Finally, *FastSwap* is by design light-weighted and non-intrusive. We evaluate *FastSwap* using a set of well-known big data analytics workloads and benchmarks, such as Spark, Redis, HiBench, SparkBench and YCSB. The results show that *FastSwap* offers up to two orders of magnitude performance improvements over existing memory swapping methods and more than four orders of magnitude faster than conventional disk based VM swapping facility.**

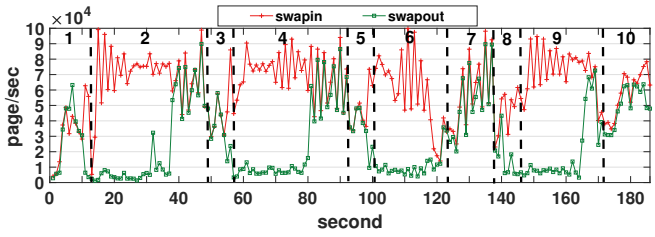
***Keywords*-Virtualization, shared memory, memory swap**

I. Introduction

The growth rate of big data in cyberspace continues to outpace both hardware and software technologies. Many machine learning algorithms and data analytics jobs experience temporal memory usage variations across virtual machines or executors on the same machine and across a cluster. In virtualized clouds, the analytics workloads enjoy optimal performance when their data can fully fit into the physical memory allocated prior to runtime. However, when the data exceeds the allocated physical memory of the analytic computation, the applications will experience serious performance degradation due to excessive memory swapping, which may lead to out of memory error, even though the host machine has enough free memory. Figure 1(a) and Figure 1(b) show periodic burst of memory swapping events during each iteration of the logic regression workloads, because the input dataset cannot fully fit into the available physical memory and some portions of the dataset



(a) Loading



(b) Iteration

Figure 1. Logistic Regression Swapping

are stored in the swap backing storage on the disk swap partition of the OS while keeping their memory foot-print through the conventional OS virtual memory facility.

Figure 1(a) shows the VM swapping events of the LR workload with 5 million samples during data loading and the swap-out events are the dominating swap traffic. Figure 1(b) shows the VM swapping events for the subsequent 10 iterations and the swap-in traffic dominates with relatively moderate amount of swap-out events during each iteration, showing periodic patterns of VM memory swapping traffic.

Existing Solutions. Several solutions have been proposed to address the VM memory pressure problem. The balloon driver is proposed [17], [8] as a dynamic memory balancing mechanism for efficiently moving memory from one VM to another via host with inflation and deflation operations. However, when to start ballooning and how much memory ballooning is adequate have been a challenging problem and rely on manual administration. Several existing efforts propose to estimate the working set size of each VM at run time and add or remove additional memory dynamically based on its estimated memory demands [3], [9], [10], [18]. However, recent study [2] has shown that accurate estimation of VM working set size is difficult under changing workloads. Consequently, any delay in VM memory consolidation, such

978-1-5386-5035-6/18/$31.00 ©2018 IEEE

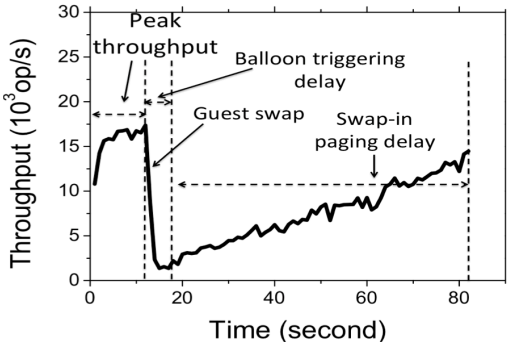


Figure 2. Effect of VM Swapping on Redis performance

as ballooning, can cause the VM under memory pressure to experience increased memory swapping events when the application cannot fit their working set fully in its allocated physical memory. Excessive memory swapping can lead to serious performance degradation for applications, which may suffer out of memory induced system crash due to high latency swapping events induced timeout.

Figure 2 shows some experimental results of Redis [1] (a popular main-memory key-value system). The dataset uploaded to Redis is slightly larger than the VM physical memory allocated at VM initialization time, and Redis transparently stores some data in the form of swap-out pages on external swap storage (e.g., disk in conventional OS). A Redis client is running a key-value workload of 50% uniform read and 50% uniform write generated by YCSB [6]. It is observed from Figure 2 that the throughput of Redis server is increasing during the first 10 seconds, because all its data access is served in memory and there is no memory swap. However, as soon as Redis has to access the portion of its dataset that are not residing in its physical memory, the Redis server experiences drastic throughput degradation due to heavy memory swapping events. At the 15th second, the balloon driver is turned on and it takes only a couple of seconds to balloon 4GB memory from host to VM even when the CPU utilization on both the VM and the host is at 60%-80%. This figure also shows two interesting facts: (1) The Redis server starts swap-in pages at the 18th second (when its memory is inflated via ballooning, and the Redis performance is recovering slowly. (2) Even with sufficient memory, Redis still takes more than 60 seconds to recover to its normal performance. We observe that this is mainly due to the large number of page-fault triggered swap-in operations and the series of disk I/Os to read data from disk swap area to the Redis working memory.

Contributions of the Paper. We have shown through experiments two important observations. First, big data analytic workloads often experience memory swapping events when their datasets cannot fully fit into the physical memory allocated (recall Figure 1(a) and Figure 1(b)). Second, applications (e.g., Redis) suffer from performance degradation when their working set no longer fully fit into the physical memory even when other VMs on the same host have unused idle memory (recall Figure 2). Even after sufficient memory is ballooned from host to the VM, the performance of application takes time to recover to its peak performance due to slow and inefficient memory swap-in operations. In this paper, we address these problems by presenting *FastSwap*, a host-coordinated shared memory swapper with multi-granularity compression on memory swap pages. We show that *FastSwap* can significantly improve the runtime performance of big data analytics workloads in virtualized Clouds. The design of *FastSwap* is original in three aspects:

- *FastSwap* provides an efficient and transparent memory sharing mechanism by creating a shared memory based swap area between the host and its VMs. This shared memory mechanism can effectively leverage idle memory present in the host or the other VMs on the host and speed up the VM memory swapping performance by minimizing or avoiding the high overhead of disk I/O for memory swapping on guest VMs.
- *FastSwap* improves both memory swap-out and swap-in performance by introducing a suite of shared memory optimizations. (1) *FastSwap* compresses swap-out pages before placing them in the shared memory or disk swap area and decompresses swap-in pages before making them available for VM execution. (2) *FastSwap* supports four different compression ratios to group compressed pages into four compressed page groups: 512B, 1KB, 2KB and 4KB. (3) *FastSwap* implements a compressed swap page table as an efficient index structure to enable fast lookup of compressed pages in four granularity groups from its swap partitions. This index enables high compression ratio and fast VM swapping for many big data applications.
- *FastSwap* provides a hybrid swap-out scheme to handle the situation of limited shared memory for VM memory paging, which redirect memory swapping to shared memory first and then managing the remaining swapping out traffic on the slower external backing storage such as disk swap partition. In addition, *FastSwap* provides a two-level proactive batch swap-in facility to speed up the performance of VM memory swap-in operations in the presence of sufficient shared memory.

The first prototype of *FastSwap* is implemented on KVM platform [12]. We evaluate the performance of *FastSwap* using a set of big data analytics applications and benchmarks, such as Redis, Spark, HiBench, SparkBench and YCSB. We show that compared to conventional VM memory management in conventional virtualized Cloud, which allocates memory at VM initialization and shares nothing during VM execution, *FastSwap* can effectively and transparently improve application performance by creating and managing shared memory among hosted VMs. For example, *FastSwap* can speed up the Redis execution throughput by **516x** and improves the Spark runtime by **8.73x** by leveraging unused

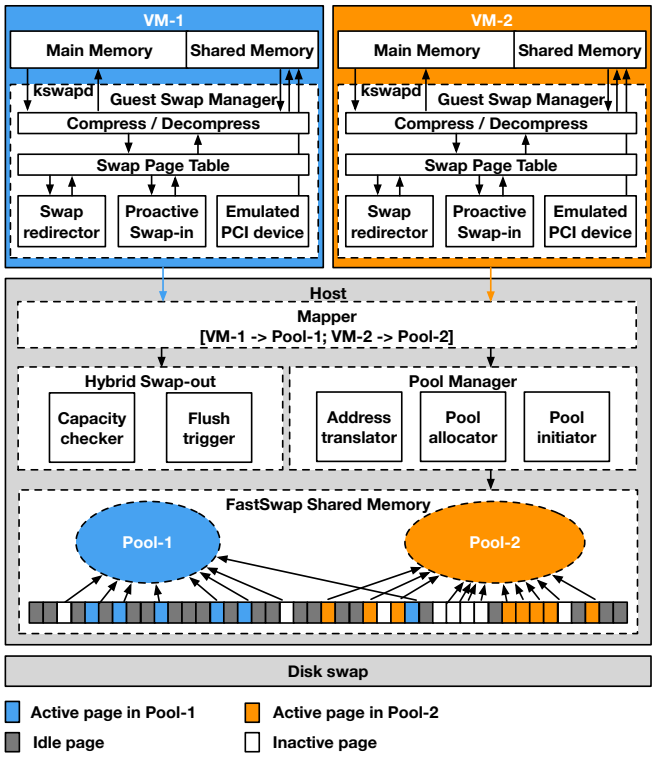


Figure 3. *FastSwap* System Architecture

host memory present in other VMs when the applications are experiencing high memory pressure. *FastSwap* is open sourced on https://github.com/git-disl/FastSwap. The initial test results from several external collaborators show consistent performance gains across both big data driven No-SQL systems and big data driven machine learning workloads.

II. Design Overview

The system architecture of *FastSwap* consists of four core functional components as shown in Figure 3:

- **Dynamic Shared Memory Management.** A piece of shared memory is pre-allocated in the host and shared by multiple VMs as their swap areas. This shared memory is equally divided into small chunks. Each chunk is dynamically assigned to and revoked from a VM based on its swap traffic, enabling dynamic shared memory consolidation.
- **Hybrid Swap-out.** When the amount of shared memory can not hold all the memory pages swapped out from a VM, *FastSwap* uses shared memory swap area as the fast primary swap partition and resort to the secondary storage swap partition for least recently swap-out pages. This functionality improves the overall VM swapping performance even when a small amount of host idle memory can be leveraged as the shared memory swap area.
- **Two Level Proactive Swap-in.** *FastSwap* improves page fault triggered swap-in performance by two level proactive batch swap-in: (i) the first level is a threshold controlled batch swap-in scheme, enabling VM to proactively swap-in all pages or a threshold of pages as a batch from the shared memory swap partition to the VM main memory; and (ii) the second level is an opportunistic batch swap-in scheme, which minimizes the costly page faults and enables fast recovery of application performance when sufficient memory is made available to the VM under memory pressure.

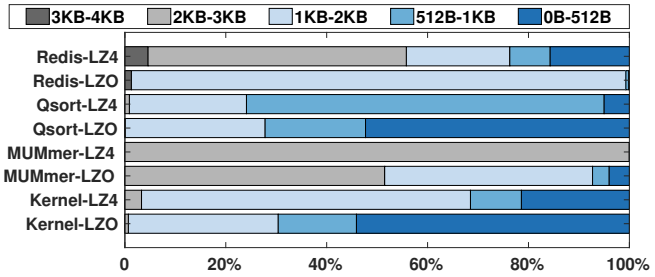


Figure 4. Data Granularity

- **Multi-granularity Compression of Shared Memory.** *FastSwap* by design compresses all swap-out pages before placing them into the proper swap partition. Similarly, all swap-in pages will be decompressed first before returning them to the application's working memory. To further improve the shared memory utilization, *FastSwap* designs an effective multi-granularity scheme for compression of memory swap pages. This is motivated by the argument that the efficacy of compression approach mainly depends on two factors, compressibility of data and choice of data granularity [13], [16].

Figure 4 shows the compression of the memory pages using two popular compression algorithms LZO-1X [15] and LZ4 [5] for four applications: Linux 4.1.0 kernel compilation, MUMmer 3.0, QuickSort algorithm with 20 million random number and Redis 3.0.7 with random workload generated by YCSB. We observe that the percentage of memory pages in different compression size groups varies significantly by the two compression algorithms and also varies from application to application, leading to different compression ratios. These observations motivate us to carefully select a memory compression algorithm with high compression ratio and to design a multi-granularity swap page compression scheme.

There are two alternative ways to implement the host-coordinated shared memory swapping: (i) swapping to host ramdisk and (ii) swapping to host-guest shared memory.

Ramdisk based Swapping builds a ramdisk for a VM in the host memory and mounting this ramdisk as the swapping area of the guest VM. Each ramdisk can be mounted to a different VM at different times based on the swapping demands of the guest VMs. The ramdisk approach is simple and requires no guest kernel modification. However, ramdisk solution has some inherent performance limitations. Although the ramdisk is residing in the host memory, it is still used by the guest VM as a block device. As a result,

certain overheads that are applicable to disk I/O also apply to ramdisk I/O. For example, a disk I/O request from the application running on a guest VM will first go through the guest OS kernel before the request and the I/O data are copied from the guest kernel to the host OS kernel. Hence, when a guest VM swaps its memory pages to a mounted ramdisk, it suffers from two major sources of overhead: the frequent context switch and the data copying between the host user space and host kernel space.

Shared Memory based Swapping is designed and implemented in *FastSwap*: a memory region provided by the host is mapped into the guest VM address space and used as the host-guest shared swapping area for the guest VM. As a part of the system initialization, a shared memory region is divided into multiple elastic memory *pools*, with each corresponding to a specific VM. Each pool is again divided into slabs with a pre-configured slab size. A pool manager at host performs two main tasks: (i) maintaining the mapping between the page offset in the VM swap area and the address in the corresponding shared memory pool, and (ii) dynamically adjusting the size of each VM pool by allocating or deallocating a slab from the host shared memory. The pages in the shared memory are categorized into three types: *active* pages refer to ones being used by the VMs as their swapping destination, *inactive* pages are those allocated to some pool but has not being used yet, and *idle* pages indicate those shared memory pages that do not belong to any pool. The *kswapd* as a default kernel daemon is responsible for memory page swap-in and swap-out. The swap redirector intercepts and redirects the swap in/out pages from/to the VM shared memory area. The *FastSwap* shared memory approach delivers up to 1.7 times faster performance than the Ramdisk solution on Redis, Spark and several benchmark workloads [19] without any modifications to user applications and the OSes.

III. Memory Swap Page Compression

In the first prototype of *FastSwap*, we maintain four *fastswap _pool*s to support four compressed page groups, each is represented by a linked list with the *Pool Header* as the start node of pool, and each member of the list corresponds to a shared memory page. For example, the *BUD Header* that belongs to 512B pool will manage a 4K page that has 8 slots to store eight 512B compressed pages. Every time we put one compressed page in, the available slot reduces by one and the occupied slot is recorded. When there is no more *BUD Header* in the free list, a new *BUD Header* will be created and one free shared memory page will be assigned to it. *FastSwap* has a global pointer that keeps track of all used shared memory pages. *Pool Header* contains two pointers, one points to full list and the other points to free list. We maintain an LRU list for each pool. Whenever one *BUD Header* is updated, such as adding or deleting a slot, this *BUD Header* will delete itself from the current

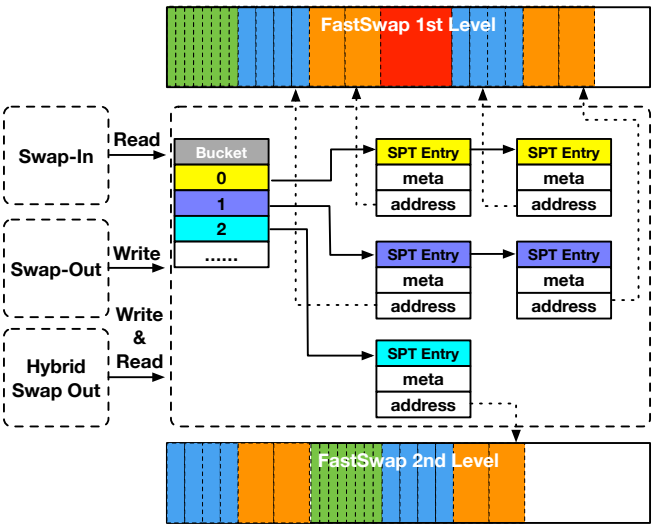


Figure 5. Swap Page Compression Index Table

LRU and re-insert itself to the beginning of this LRU list. By maintaining four pools separately, four *fastswap_pools* can work concurrently to improve the throughput of VM memory swapping.

A *FastSwap* store operation is invoked when a page is selected for swapping-out by the guest OS. *FastSwap* intercepts the page using the *swap_writepage()*. The operation begins by compressing the page into a temporary buffer in order to know which one of the four compressed page-groups this compressed page should belong to. Based on the compressed size, *FastSwap* chooses one of the four compression granularity pools, which is most suitable for holding the compressed swap page. If the free list in the pool is not empty, then we get the free slot address by reading *BUD Header* meta data. Otherwise, allocating a new *BUD Header*, linking it to the free list, assigning it a new free shared memory page, and have temporary buffer copied into the corrected slot address.

A *FastSwap* load operation is invoked when a page fault is triggered to swap-in a page with a page table entry (PTE) that contains a swap entry. This page-fault based swap-in is intercepted by *FastSwap* in *swap_readpage()*. The swap entry contains the page information needed to look up the *swap_page_table_entry* in Swap Page Compression Index Table. Once the entry is located, the address of that compressed page can be read by *swap_page_table_entry*. Finally, the data is decompressed directly into the page allocated by the page fault event and the *BUD Header* containing the page updates its LRU status by putting it to the top of the LRU list (most recent one). When the shared memory is insufficient, *FastSwap* invokes the hybrid swap out algorithm (see Section IV for detail).

To support efficient lookup of compressed swap pages, we create and maintain the Swap Page Compression Index Table (SPCIT) with meta data about each compressed page, and

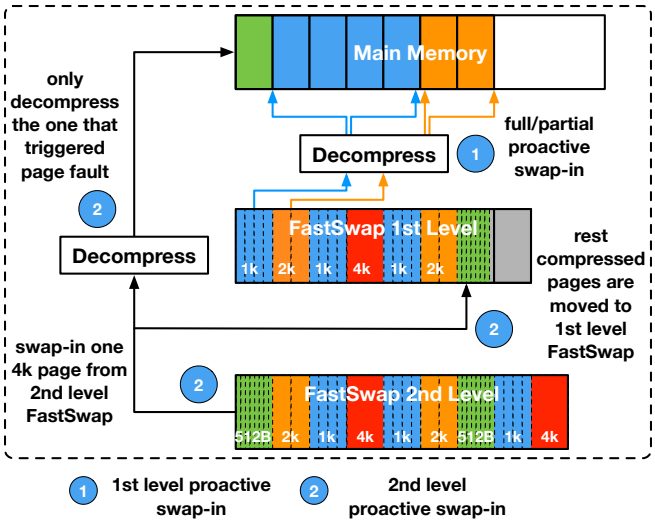


Figure 6. Two Level Proactive Swap In

the address mapping between VM memory page address and its compressed page address using Hashtable, which can be built by *hlist*, a kernel linked list data structure in Linux. In the first prototype, we implement the Hashtable as an array of struct *hlist_head* pointers, where each points to a different list, and each list holds all elements that are hashed to the same bucket. Thus, every element is essentially a member of a *hlist* and the Hashtable only holds the head of these lists. When *FastSwap* intercepts a swap out page request, SPCIT creates a *swap_page_table_entry* for the swapped out page and links it to the Hashtable (see Figure 5). Within that entry, a binary variable field is used to indicate the location of the swap partition where the compressed page currently resides: either 1st-level shared memory or 2nd-level external backing storage (e.g., disk or remote memory). We also record in the *swap_page_table_entry* other metadata, such as the address of this compressed page, and the length of compressed page which is used for decompression. We also add *spt_chain* in each entry for handling hash collision. Similarly, the metadata structure used to implement proactive swap-in is also stored in the *swap_page_table_entry*, such as the address of the PTEs corresponding to this compressed swap page.

IV. Hybrid Swap-out

Hybrid Swap-out is designed to make the shared memory as the fast primary swap partition with smooth transition to the external backing swap storage as the secondary swap partition when the size of shared memory is not large enough to hold all the swapped pages from a VM. We design an optimal hybrid swap-out model to keep most recent swap traffic in shared memory by moving (dumping) least recent swap-out pages to external swap device (e.g., disk). *FastSwap* implements *least recent compressed swap pages to the secondary swap partition* approach, which uses a user-defined system default batch dump threshold, such as 80%, as the default configuration. When the shared memory swap area reaches 80% full, it will flush all pages in the shared memory swap area to the secondary swap partition by batch I/Os. This approach avoids frequent LRU sorting, decompression latency, and per-page based eviction. Alternatively, when the shared memory is full, the newly swap-out pages will be written to the disk swap area. This approach is straightforward but naive and suffers from poor performance because most of the pages stored in the shared memory are older and has the worst utilization of shared memory and reduces its performance to the conventional OS memory paging as soon as the shared memory swap is filled. By the principle of locality, programs, especially big data analytics jobs, tend to reuse data near those they have used recently.

V. Two Level Proactive Batch Swap-in

The idea of proactive swap-in promotes the opportunistic swap-in pages in batch by supporting the first level swap-in from shared memory and the second level swap-in from external swap backing storage such as disk. First, we group a portion of the swapped-out pages in the shared memory swap area into a swap-in batch and proactively perform the batch swap-in without guest memory paging. We implement a threshold-based batch swap-in. By using the threshold of 100% all the pages that are previously swapped out to the shared memory swap partition will be grouped into one swap-in batch. This design allows one to configure and tune the swap-in batch size based on workload characteristics as well as VM available memory.

For example, if a page previously swapped out to disk is being accessed by guest VM via a page fault, then a 4KB disk block containing the requested swap-out page is fetched from the disk swap partition. We first locate the requested page in compressed format by lookup in SPCIT and decompress it before moving it to the VM memory. In addition, if the VM has sufficient memory, we will proactively move those remaining compressed pages contained in the same disk block as the requested swap-in page to the shared memory swap partition without decompression. Finally, we update the locality of these pages from disk swap partition to shared memory swap partition. We maintain some metadata in SPCIT, which can assist *FastSwap* to quickly locate the corresponding PTEs in the page table managed by guest OS. For example, when a page is swapped out, *FastSwap* will keep the address of the PTEs related to this page together with the swapped-out page. The address of the corresponding PTE is kept as the metadata in *swap_page_table_entry*. For each page of 4K bytes, this metadata only takes up 4 bytes. The cost of keeping this metadata in the swap area is only about 1/1000 of the total size of the swapped-out pages. For those shared pages which have multiple PTEs, we allocate a specific area in the shared memory as *PTE store*. In this case, the first

Table I
LR PERFORMANCE SPEEDUP W. DISKSWAP AS THE BASELINE (M:MILLIONS)

	Dataset	DiskSwap	ZSwap	FS-baseline	FastSwap
T-1	5m/18GB	1	1.46	2.2	**2.2**
	5.5m/20GB	1	crash	**2.37**	2.29
	6m/22GB	1	crash	**2.43**	2.35
T-2	5m/18GB	1	1.10	1.35	**1.38**
	5.5m/20GB	1	crash	2.55	**2.58**
	6m/22GB	1	crash	**8.73**	8.35

byte of the metadata specifies the number of PTEs related to this page, while the lasts three bytes is an index pointing the first related PTE in the *PTE store*. This allows *FastSwap* to quickly locate the PTE(s) that needs to be updated by referring without scanning the page table when a page is swapped in. More importantly, the time spent on accessing the PTE of a swapped page will not increase as the size of the system wide page table grows.

VI. EVALUATION

We evaluate *FastSwap* using popular big data benchmarks. All experiments are conducted on host server with eight-core Xeon-E5640 CPU, 96GB memory and a SCSI disk, running RedHat Enterprise Linux 6.8 (kernel 2.6.34) and using KVM 1.2.0 with QEMU 2.0.0 as the virtualization platform. The guest VMs run Ubuntu 14.04 with kernel version 4.1.0.

Five typical data analytics benchmark applications in Spark are used in the experiments: *LogisticRegression* (LR), *KMeans*, *Bayes*, *NWeight* (NW), *PageRank* (PR). For LR, we use the datasets produced by SparkBench [14] with varying sizes of samples from 2, 3, 4, 5, 5.5 to 6 millions, which corresponds to different storage sizes of approximately 12GB, 14GB, 16GB, 18GB, 20GB and 22GB respectively. Datasets used in KMeans, Bayes, NW and PR experiments are generated by HiBench. KMeans uses 20-dimension feature vectors with 35 million samples. Bayes uses 2.5 million pages with 100 classes and n-gram 2. NW dataset has 4 million edges with minimum out-edges of 3 and max out-edges of 30. PR dataset includes 750,000 pages. The maximum JVM heap size of each executor is set to be 28GB. In addition to the five types of Spark analytics workload, Redis is used as memory intensive database workloads. The size of the datasets randomly generated by YCSB varies from 10GB to 30GB. We compare three different memory paging alternatives: (1) *FastSwap*, a compressed shared memory swapper with multi-granularity compression, hybrid swap-out, and two-level proactive swap-in; (2) *DiskSwap*, a conventional OS disk swap facility. We refer to this setup as the baseline. (3) *ZSwap* [11], a compressed RAM cache for disk based swap devices. Swap-out pages are first compressed and maintained in the reserved RAM swap area. When the RAM is exhausted, each swap-out page operation will trigger an eviction from the RAM by finding the least recently used (LRU) page and decompress it and send it to disk swap partition. *ZSwap* can be viewed as a naive baseline implementation of our shared memory optimization.

A. Overall Performance of Fastswap

We use all five types of Spark analytics workloads and the in-memory key-value database benchmark – Redis to evaluate the effectiveness of *FastSwap*.

Performance Comparison when Applications Compete Memory Resource on a VM (T-1 case). We consider the scenario in which two memory intensive applications are running in parallel on one VM and the host has sufficient unused memory since other VMs are idle. We use Spark LogisticRegression (LR) workload and Redis with YCSB read workload. LR is a representative machine learning application that performs iterative computations. It first loads and caches the entire training dataset into the main memory, then iteratively compute the logistic regression and updates the model until the pre-defined convergence condition is met. In all experiments, we run 3 iterations by default unless explicitly stated otherwise. YCSB read-all workload is used to simulate massive data access, with the zipfian distribution. For *DiskSwap* solution, we setup a Redis in-memory database server on a VM with 30GB main memory and pre-load 10GB key-value data to the VM, and then start LR and YCSB read-all workload together. For LR workloads, we use datasets of 18GB, 20GB and 22GB, representing the three scenarios in which the LR dataset can fit in memory with 100% or 75% or 50%. We measure the LR execution time, the Redis throughput, and system resource utilization. We report the measurement results in the T-1 section of Table I and provide the detailed analysis measurements in the T-1 section of Table II. Table I T-1 section displays the speedups achieved by *FastSwap* without compression (*FastSwap baseline*) are ranging from 2.2x to 2.43x, and by *FastSwap* are ranging from 2.2x to 2.35x. In comparison, ZSwap crashes in the second and third LR test (20GB, 5.5 million samples and 22GB, 6 million samples), because the long latency of VM swapping causes the Spark executor heartbeat time out.

The detailed system level measurements are given in the T-1 section of Table II. We make several interesting observations. *First*, compared to DiskSwap case, *FastSwap* and *FastSwap* without compression (*FastSwap baseline*) significantly reduces the CPU iowait by 47% (1-53%) and 70% (1-30%) respectively. It shows that *FastSwap* increases swap-in speed by more than 30x and swap-out speed by 3x, thus effectively minimizing the latency of VM swapping. In comparison, ZSwap incurs higher CPU iowait (106%) than the traditional disk swap solution. ZSwap performs poorly is due to its per-page based swap-in and swap-out solution combined with the cost of making LRU eviction decision for every page to be swapped in or swapped out after the shared memory is full, in addition to the cost of frequent compression/decompression. *Second*, *FastSwap* reduces the

Table II
COMPARISON RESULTS OF LR AND REDIS WITH TWO EXPERIMENTAL SETUPS: T-1 AND T-2

	Metric	DiskSwap	ZSwap	FS-baseline	FastSwap
T-1	Avg. CPU iowait(%)	56.86	60.29(106%)	**16.89(30%)**	19.98(53%)
	Avg. swap-in(page/sec)	54.44	62.55(115%)	**1783.98(3277%)**	1657.86(3045%)
	Avg. swap-out(page/sec)	1856.43	521.58(28%)	**7003.46(377%)**	6912.81(372%)
	Total swap space (MB)	6788.29	11864.18(175%)	6514.11(96%)	**4372.12(64%)**
	Avg. Redis Throughput(ops/sec)	46.24	1100.77(2381%)	**23898.79(51600%)**	20913.04(45227%)
T-2	Avg. CPU iowait(%)	61.29	58.89(96%)	18.78(31%)	**12.37(20%)**
	Avg. swap-in(page/sec)	598.05	221.28(37%)	**15816.58(2644%)**	14510.55(2426%)
	Avg. swap-out(page/sec)	781.88	779.75(100%)	**19460.07(2589%)**	16471.14(2107%)
	Total swap space (MB)	8159.05	7910.57(97%)	8344.68(102%)	**4689.39(58%)**

shared memory consumption to 64% due to the use of multi-granularity of compression groups. Furthermore, by comparing *FastSwap* and *FastSwap* without compression, we observe that the compression incurs very small overhead. *Third*, *FastSwap* and *FastSwap* without compression show a significant throughput performance boost for Redis workloads with 452x and 516x respectively, compared to the performance of the traditional DiskSwap solution. More interestingly, *FastSwap* can improve both Spark workloads and Redis workloads at the same time due to shared memory swap optimization, even when both applications are competing for the scarce physical memory resource.

Performance Comparison under Multiple VMs with Memory Contention (T-2 case). This set of experiments focuses on measuring the effectiveness of *FastSwap* when two VMs both run memory intensive big data applications. We setup two VMs with each allocated 30GB main memory. VM1 runs Spark with LR workload and VM2 runs Redis with YCSB load workload. The Redis client loads 30GB key-value data generated by YCSB to the Redis server running on VM2, which causes high memory pressure on VM2. Given that VM1 is initialized and have unused memory, by using the balloon driver, we move 10GB from VM1 to VM2 [17]. With 40GB running Redis workloads of 30GB and 20GB memory remaining on VM1 to run Spark LR workload of three sizes. The performance results are shown in the T-2 section of Table I and Table II. Compared to the traditional *DiskSwap*, both *FastSwap* and its baseline (without compression) respond well and provide much better performance in the presence of the increasing memory contention on VM1 due to the increased size of the training datasets from 5 million (18GB) to 5.5 million (20GB) and 6 million (22GB). However, ZSwap crashed again for the datasets of 20GB and 22GB. For 22GB dataset, *FastSwap* and *FastSwap* without compression provides 8.35x and 8.73x speedup respectively over DiskSwap. Also compared to *FastSwap* baseline, *FastSwap* offers significant reduction of swap storage (58%) and the smallest CPU on iowait (20%) compared to both *DiskSwap* and *FastSwap* baseline. This shows that multi-granularity compression in *FastSwap* incurs low overhead, with slightly slower performance than *FastSwap* without compression for most benchmarks.

Performance under Multiple Benchmarks. In the next set of experiments, all five types of Spark analytics workloads are used with the same experimental setup. Table III shows the results. We observe that *FastSwap* (and its non-compression baseline *FS-baseline*) significantly outperforms DiskSwap and ZSwap for all 7 benchmarks. For example, *FastSwap* can run the LR with 6 million samples (22GB) at only 11.9% (11.5 mins) of the time spent by using the DiskSwap solution (96 mins). Also when using DiskSwap, two Spark benchmarks (Bayes and PageRank) crashed, and when using ZSwap, five Spark benchmarks (KMeans, Bayes, PageRank, LR 20GB and LR 22GB) crashed. This is due to the Spark heartbeat time out because of the poor performance of VM swapping when using the conventional DiskSwap and ZSwap.

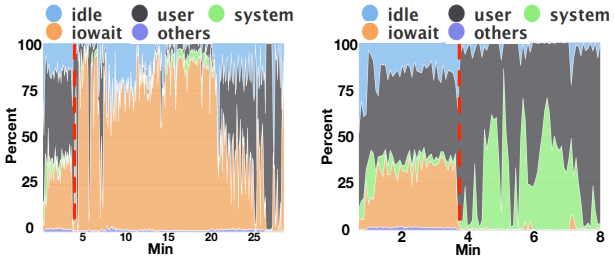
Table III
EXECUTION TIME (MINS) OF BENCHMARKS (M:MILLION)

	DiskSwap	ZSwap	FS-baseline	FastSwap
KMeans	28	crash	8	8.3
Bayes	crash	crash	13	13
NW	25	4	2.8	3
PR	crash	crash	4.5	4.7
LR 5m	11	10	8.1	8
LR 5.5m	24	crash	9.4	9.3
LR 6m	96	crash	11	11.5

To further analyze the effectiveness of *FastSwap*, Figure 7 and Figure 8 show the CPU usage and disk throughput for KMeans workloads respectively. Similar with LogisticRegression, KMeans first loads and caches into the VM main memory the training dataset of 35 million samples with dimension-20, generated by GenKMeansDataset based on Guassian distribution, then it iteratively updates the model until the pre-defined convergence condition is met. Thus, KMeans workload consumes huge amount of memory. The red dotted line in Figure 7 and Figure 8 indicates the starting point of VM swapping due to insufficient physical memory. Given that DiskSwap finishes KMeans workload in 28 mins, the x-axis varies up to 28 minutes. Similar setup of x-axis for *FastSwap*. For ZSwap, it did not finish after executed for 35 minutes and crashed eventually for the KMeans workload. Thus, we omit ZSwap in this comparison.

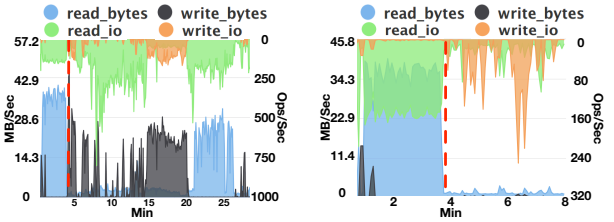
We make several observations. *First*, Figure 7(a) shows that DiskSwap incurs high CPU utilization on iowait during swapping. Figure 8(a) shows large disk write operations due

to memory swapping to disk. In comparison, *FastSwap* spent most of the CPU on shared memory swapping and little CPU on iowait and little disk writes in the presence of VM swapping. We also observe that the average of CPU for system usage for *FastSwap* without compression is slightly lower than that of *FastSwap*. This is mainly due to the CPU cost for compression in *FastSwap*. Table III shows that the compression overhead has negligible overhead on the execution time for all 7 benchmarks. *Third*, DiskSwap has large percentage of CPU spent on iowait, due to slow disk write operations during swap in and out.



(a) DiskSwap (b) FastSwap

Figure 7. CPU Usage of KMeans

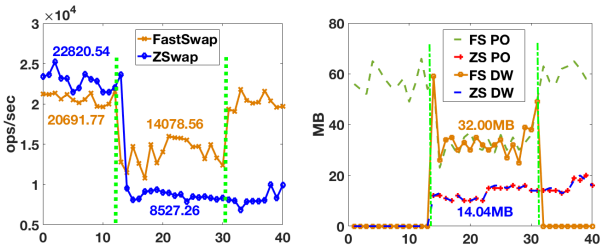


(a) DiskSwap (b) FastSwap

Figure 8. Disk Throughput of KMeans

B. Hybrid Swap-Out

In this section, we evaluate the effectiveness of the hybrid swap-out mechanism in *Fastswap* by considering the cases when the shared memory swap space is insufficient to hold all the pages swapped out from the guest VM. We measure the performance of Redis and Spark workloads, when the shared memory swap area is 80% full (or reaches a pre-defined dump threshold). In this case, *FastSwap* performs the hybrid swap-out by copying pages from the shared memory swap partition to the disk swap partition. For the sake of comparison, we use a similar setup for this set of experiments and measure both *FastSwap* and ZSwap. Figure 9 shows the Redis throughput (ops/sec) and the disk I/O in MB. The two vertical green dotted lines indicate the period of hybrid swap-out dumping process. The first dotted line is also the time when ZSwap physical memory capacity is full, because we choose the dump threshold in *FastSwap* accordingly for fair comparison. We make three observations from Figure 9. *First*, during the hybrid swap-out process, Redis throughput is degraded by 32% with *FastSwap* but

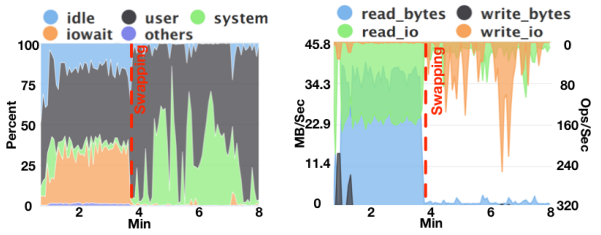


(a) Redis Throughput (b) Disk Throughput (FS = FastSwap, ZS = ZSwap, PO = PagingOut, DS = DiskWrite)

Figure 9. Redis Performance with Hybrid Swap-Out.

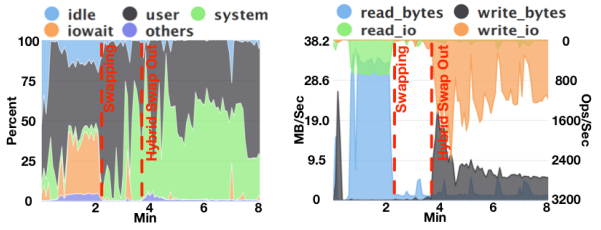
with ZSwap, Redis throughput is degraded by 63%. Also the throughput of Redis with *FastSwap* consistently outperforms that with ZSwap during the entire hybrid swap-out dumping period. *Second*, as soon as *FastSwap* dumping process completes, Redis restores to its best throughput performance. In contrast, using ZSwap, Redis is running with much lower throughput and remains at the poor throughput even after the hybrid swap-out dumping period, due to the lack of optimizations in ZSwap design. For example, in ZSwap, when its swap RAM cache is exhausted, each of its swap-out page operations will trigger an eviction from the RAM by finding the least recently used (LRU) page and decompress it and send it to disk swap partition. This design incurs high overhead due to per-page based swap in and out, combined with the cost of compression/decompression and the cost of making LRU eviction decision for every swapped page after its swap RAM area is full. *Third*, in contrast, *FastSwap* keeps pages that need to be dumped to the disk swap area compressed by leveraging our compressed swap page table. Such design efficiently saves I/O bandwidth and shortens the hybrid swap-out dumping period. Figure 9(b) displays the average disk write speed during the hybrid swap-out dumping process, which is 32MB/sec (compared to 14.04MB for ZSwap) and the dumping process lasts for 18sec. Also the size of data being dumped is 576MB, which is only 56% of 1GB, thanks to the *FastSwap* compression design. Finally, we would like to comment that in the first 10 seconds, the Redis has higher throughput with ZSwap than with *FastSwap*. The reason is primarily related to the performance difference between physical memory and shared memory. ZSwap utilizes physical memory and *FastSwap* utilizes shared memory to store pages.

Next, we evaluate the performance of Spark KMeans workloads during the hybrid swap-out process. We measure CPU usage and disk throughput to compare FastSwap-without-hybrid-swap-out with FastSwap-with-hybrid-swap-out as shown in Figure 10 and Figure 11 respectively. The hybrid swap-out is triggered when the default threshold of shared memory swap partition is reached (e.g., 80%). The first red dotted line in Figure 10 and Figure 11 indicates the start point of guest VM swapping and the second one indicates the starting point of hybrid swap-out. A number



(a) CPU Usage (b) Disk Throughput

Figure 10. KMeans Performance w.o. Hybrid Swap-Out.



(a) CPU Usage (b) Disk Throughput

Figure 11. KMeans Performance with Hybrid Swap-Out.

of observations can be made. *First*, the execution time of FastSwap-with-hybrid-swap-out is 8.2 mins and that of FastSwap-without-hybrid-swap-out is 8 mins, which means hybrid swap-out only causes 2.5% performance degradation. Such overhead mainly comes from two sources: (i) reading pages that had been moved to the disk swap partition; and (ii) additional CPU cycles used by the hybrid swap out thread. *Second*, during the hybrid swap-out process, *FastSwap* did incur large amount of disk write operations as shown in the dark area of Figure 11(b). However, these write operations do not increase CPU iowait, indicating good disk I/O efficiency for writes. *Third*, since a separated thread is created in kernel space only for the hybrid swap out purpose, from 2.2 mins to the end in Figure 11(a) we can see that this kernel thread is working and consuming CPU percentage on the kernel space operations (i.e., the CPU percentage spent on kernel system).

C. Proactive Swap-In

Proactive swap-in speed is measured for DiskSwap, FastSwap without compression (FS-baseline), FastSwap level-1 proactive swap-in and FastSwap level-2 proactive swap-in, and the results are shown in Table IV. For *DiskSwap*, *FS-baseline* and *FastSwap* level 1 (full swap-in) case, all pages in the shared memory swap area will be swapped in. For *FastSwap* partial case (level 2 proactive swap-in, we let *FastSwap* perform partial proactive swap-in of 500MB repeatedly until there is only 1GB free space left in the main memory.

We make four observations from Table IV. (1) Since the process of swapping-in page needs to update PTE, and in *DiskSwap* case, OS needs to scan the page table to find the corresponding page table entries that it needs to update, which is very time consuming, the performance of

Table IV
PROACTIVE SWAP-IN SPEED (PER PAGE)

	Avg. Time(us)	s.d.
DiskSwap	419.66	7.84
FastSwap w.o. compression	5.70 (74x)	0.18
FastSwap (full)	9.56 (44x)	0.30
FastSwap (partial)	9.68 (43x)	0.12

DiskSwap case is very slow. (2) Because in *FastSwap* and *FastSwap* without compression case, the OS swaps out not only memory pages but also some metadata, which can help the OS to quickly identify which page table entries need to be updated during the swap-in process. Thus the cost of scanning the page table is avoided. Both versions of *FastSwap* are a lot faster than *DiskSwap*. (3) Because of the use of compression, *FastSwap* needs to decompress each page before swapping it back into the main memory, leading to higher latency than the version of *FastSwap* without compression. However, comparing with *DiskSwap* case, *FastSwap* is still 44x faster in terms of per-page swap-in performance. (4) The partial batch swap-in option enables small amount of batch swap-in when the hosted VM has insufficient free memory to hold all pages stored in the shared memory swap area. We argue that the full batch swap-in should be treated as a special case of the threshold based batch swap-in to provide high utilization of available VM memory and high utilization of shared memory. From Table IV, the partial proactive swap-in displays very similar performance as the full proactive swap-in in terms of per-page swap-in operation time.

VII. RELATED WORK

Efficient memory management in virtualization platforms is widely acknowledged as a challenging problem for memory intensive workloads. Several orthogonal efforts have targeted at scheduling, allocating and consolidating physical memory among VMs on demand with unmodified guest OS. Ballooning [17] was proposed in 2002. A balloon driver, running in the guest OS, can help the VM Manager to reclaim or repay guest memory by inflating or deflating the balloon manually. But it relies on administrator to manually when to trigger ballooning and how much to balloon from one VM to another. Although research efforts have been dedicated to periodic estimation of VM working set size, it is well known that accurate VM working set prediction remains to be an open issue under change conditions [10].

VSwapper [2] is a disk-based VM swapping facility and designed to address the uncooperative host swapping problems, such as double paging by tracking the correspondences between disk blocks and guest memory pages and between host and guest VMs. It does not improve VM memory paging performance using shared memory or RAM.

Existing efforts in computer networks community have explored disaggregated memory by memory paging to remote nodes via RDMA network, and Infiniswap [7] is the

most recent representative work in RDMA-based remote paging. However, these efforts only support memory swapping to remote host and fail to provide memory sharing across containers/VMs/executors on the same host. Even when there is sufficiently unused memory on the host, VMs or containers have to swap pages to remote host instead.

FastSwap by design can integrate memory sharing across VMs/containers regardless whether they are on the same host and different host machines. This paper has shown that *FastSwap* offers up to 516x performance improvement for both NoSQL and ML workloads over conventional Linux memory swap facility and 22x improvement over existing open source effort, such as ZSwap, on memory sharing across VMs on the same host. Our preliminary measurement of *FastSwap* shows 14x performance improvement over Infiniswap when integrating RDMA based memory paging to remote nodes when there is insufficient memory on the local host. Due to space constraint, we omit the remote memory sharing techniques and detailed comparison, and refer readers to our technical report for detail [4].

VIII. Conclusions

Big data continues to penetrate our work life and everyday life. Performance of big data analytic workloads continues to dominate the wide deployment of systems, applications and services. We have presented the design of *FastSwap*, a highly efficient shared memory paging facility, with three original contributions. (1) *FastSwap* dynamic shared memory management scheme can effectively utilize the shared memory across VMs through host coordination. (2) *FastSwap* provides efficient support for multi-granularity compression of swap pages in both shared memory and disk swap devices. (3) *FastSwap* provides a hybrid memory swap-out scheme to flush the least recently swap-out pages to disk swap partition when shared memory swap partition reaches a prespecified threshold and close to full. (3) *FastSwap* provides two level proactive swap-in optimizations (shared memory to memory and disk to shared memory). Our extensive experiments using big data analytics applications and benchmarks demonstrate that *FastSwap* offers up to two orders of magnitude performance improvements over existing memory swapping methods. *FastSwap* will be officially released on github this fall at https://github.com/git-disl/FastSwap.

IX. Acknowledgment

This research was partially sponsored by NSF under grants SaTC 1564097, CNS-1115375, IIP-1230740, and an RCN BD Fellowship provided by the Research Coordination Network (RCN) on Big Data and Smart Cities, and an IBM Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies and companies mentioned above.

References

[1] Redis, an in-memory data structure store. https://redis.io.

[2] N. Amit, D. Tsafrir, and A. Schuster. Vswapper: A memory swapper for virtualized environments. In *SIGPLAN*, 2014.

[3] R. Birke, L. Y. Chen, and E. Smirni. Data centers in the wild: A large performance study. Technical Report, IBM Research, 2012.

[4] W. Cao and L. Liu. Efficient host and remote memory sharing for big data and machine learning applications. Technical Report, Georgia Institute of Technology, 2018.

[5] Y. Collet. Lz4–extremely fast compression. Technical Report, 2015.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

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

[8] F. Guo. Understanding memory resource management in vmware vsphere 5.0. Technical Report, VMware, Inc, 2011.

[9] 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. In *Communications of the ACM*, 2010.

[10] 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.

[11] S. Jennings. The zswap compressed swap cache. In *LWN.net*, 2013.

[12] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, 2007.

[13] M. Kjelso, M. Gooch, and S. Jones. Empirical study of memory-data: Characteristics and compressibility. In *IEE Proceedings-Computers and Digital Techniques*, 1998.

[14] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *International Conference on Computing Frontiers*, 2015.

[15] M. Oberhumer. Lzo real-time data compression library. User Manual for LZO, 2005.

[16] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy. Pinnacle: Ibm mxt in a memory controller chip. In *IEEE Micro*, 2001.

[17] C. A. Waldspurger. Memory resource management in vmware esx server. In *ACM SIGOPS Operating Systems Review*, 2002.

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

[19] Q. Zhang. *Dynamic Shared Memory Architecture, Systems, and Optimizations for High Performance and Secure Virtualized Cloud*. PhD thesis, Georgia Institute of Technology, 2017.