VOLUME: Enable Large-Scale In-Memory Computation on Commodity Clusters

Zhiqiang Ma Ke Hong Lin Gu
The Hong Kong University of Science and Technology
{zma,khongaa,lingu}@cse.ust.hk

***Abstract*—Traditional cloud computing technologies, such as MapReduce, use file systems as the system-wide substrate for data storage and sharing. A distributed file system provides a global name space and stores data persistently, but it also introduces significant overhead. Several recent systems use DRAM to store data and tremendously improve the performance of cloud computing systems. However, both our own experience and related work indicate that a simple substitution of distributed DRAM for the file system does not provide a solid and viable foundation for data storage and processing in the datacenter environment, and the capacity of such systems is limited by the amount of physical memory in the cluster. To overcome the challenge, we construct VOLUME (Virtual On-Line Unified Memory Environment), a distributed virtual memory to unify the physical memory and disk resources on many compute nodes, to form a system-wide data substrate. The new substrate provides a general memory-based abstraction, takes advantage of DRAM in the system to accelerate computation, and, transparent to programmers, scales the system to handle large datasets by swapping data to disks and remote servers. The evaluation results show that VOLUME is much faster than Hadoop/HDFS, and delivers 6-11x speedups on the adjacency list workload. VOLUME is faster than both Hadoop/HDFS and Spark/RDD for in-memory sorting. For k-means clustering, VOLUME scales linearly to 160 compute nodes on the TH-1/GZ supercomputer.**

I. Introduction

Modern cloud and big-data processing systems rely on high-performance computation in datacenters. Each datacenter contains one or more clusters composed of a large number of compute servers and a high-bandwidth interconnect, both built upon commodity hardware but often customized. Programming such a large loosely-coupled system is, however, very challenging. Traditional solutions, represented by MapReduce and Hadoop, use a distributed file system to provide a global name space and data substrate [1]–[3]. This enables the system to scale, but makes it slow and difficult to support interactive, real-time or sophisticated computation. More subtly, the file system based data substrate induces designers to assume long latencies when they design other components in the system, making the system increasingly slow.

The reliance on the file system was a reasonable design choice when hard drives were the only economical random-access device for storing hundreds of gigabytes of data [4]. However, the hardware cost structure and technological context today have been dramatically different from those constraining designers a decade ago. Along with several recent research initiatives, we believe that a memory-based abstraction is viable and can significantly enhance the programmability and performance of cloud computing systems. In 2009, Phoenix showed MapReduce computation can use shared memory on a multicore system [5]. In 2010, MRlite implemented in-memory MapReduce computation on multiple compute servers [6], RAMCloud proposed to build DRAM-based storage [7], and Spark is proposed for in-memory computation across a number of compute servers [8]. Today several DRAM-based cluster computing systems, including DVM [9], RAMCloud [10] and Spark/RDD [11], have been implemented and exhibited significant advantages in efficiency and performance.

However, a simple substitution of DRAM for the file system abstraction cannot provide the functionality and viability required for a solid data substrate on the datacenter platform. Both our own experience and related work indicate that pure DRAM-based systems have the following limitations in comparison to disk-based file systems.

- *Cost and scalability*: Although the price of DRAM has been sharply reduced in the recent decade, so does the price of hard drives. Today the cost of DRAM is still two orders of magnitude higher than that of hard disks for the same capacity. For example, a representative configuration of RAMCloud organizes 64TB storage using 1000 servers [7]. The unit cost, $60/GB based on listing prices, is at least 60 times higher than that of disks. At the same cost, a disk-based system can easily provide 4PB storage capacity.
- *Persistency*: A file system stores data persistently but DRAM is volatile. While replicating data in the main memory of multiple servers can emulate persistent data storage, such mechanisms must span wide geographic regions for it to be resilient to correlated faults, such as power outage in a city. The cost of long-distance replication is concerning, and latency multiplies even for in-datacenter replication.
- *Semantic gap*: The role of the file system in the cloud computing technology is, in fact, much wider than providing storage. It exposes a global name space for inter-process communication, enforces atomicity for data writes, and may assist in the concurrency control of the system. For example, multiple tasks on the Google cluster can exchange data in the GFS name space, and MapReduce uses GFS' atomic rename to implement idempotent reduce semantics. Such file system semantics are sometimes subtle, but they are crucial to the correctness of the computation in a large distributed system. Current DRAM-based storage systems do not provide such semantics.

We believe that the memory-based abstraction can indeed significantly enhance the capability and performance of cloud computing systems, but the design shall construct a solid data storage and exchange substrate with clearly-defined and useful

semantics, and the abstraction does not necessarily fixate on only DRAM-based organizations. In fact, DRAM and magnetic disks have their own unique advantages that complement each other's, and traditional virtual memory systems are an example of integrating the strengths of both to achieve a memory abstraction that is "as fast as memory, as large as the disks". Similarly, a synergy of memories and hard disks, facilitated by today's high-bandwidth network, will provide a fast, scalable and cost-effective way to store and transport data in a cluster.

Hence, we design a distributed memory system called VOLUME (Virtual On-Line Unified Memory Environment) to unify main memories and hard drives on many servers and construct a uniform memory space shared by potentially a large number of tasks. VOLUME handles the data in physical memory when possible, and can also smoothly scale to store large data using disks. Importantly, scaling to handle large data and the optimization of data flows are transparent to the programmers, hiding the fact that data reside on distributed compute nodes. Moreover, VOLUME allows programs to make data persistent, and defines memory operation semantics so that groups of memory operations behave in a way similar to transactions with atomicity, isolation and persistency.

It is also our goal to make VOLUME general-purpose—it should be designed to support a wide spectrum of computation, including not only offline data processing but also low-latency, iterative, interactive and consistency-critical computations. As a general-purpose substrate, VOLUME should readily benefit existing cloud computing technologies without requiring creation of new instruments or an overhaul of the existing architecture—it should be possible to substitute VOLUME for the existing data substrate in a cloud computing technology and immediately enhance its functionality and performance. Designed as a general-purpose data substrate, VOLUME can also be applied in other file system based cluster computing systems, such as MapReduce, and accelerate computation without losing the functionality, economy and scalability provided by the file systems. Our performance evaluation shows that VOLUME facilitates low-latency and high-throughput data exchanges, and that VOLUME can be 7 time faster than Hadoop on some workloads. We highlight the challenges, outline our approaches and state the contributions below.

- VOLUME aims to provide a large uniform memory space in which each processor (node) can address any subset of the memory space. The data that a single node accesses may well exceed the size of its main memory and local hard drives combined together, and, consequently, data may reside in local main memory, local hard drives, remote main memory or remote hard drives. VOLUME must transparently fetch data from and swap them to appropriate locations. This is one of the reasons why a simple design that "expands" the memory on individual nodes with large swap areas cannot work for our purpose. VOLUME must hide the location details without introducing excessive overhead, and maintain a uniform global address space with consistent semantics on a distributed system with commodity hardware. A large computation in a datacenter can potentially scale to thousands of concurrent tasks on hundreds of nodes. To the best of our knowledge, VOLUME is the first "virtual memory" system that coordinates transparent memory accesses to distributed memories and disks at such a scale.
- VOLUME should provide a mechanism for application programs to store data persistently and read data from I/O devices. We introduce persistent memory in the system, and VOLUME automatically writes data to hard drives in a way that allows other tasks or programs to retrieve the data at a later time. This mechanism advances the memory-based cloud technology to systematically specify, store and read persistent data.
- We design VOLUME to provide a transactional behavior and ensure the writes of multiple data by a task are atomic. To the best of our knowledge, VOLUME is the first system to provide transactional memory semantics on virtualized memory among many computers.

The rest of the paper is organized as follows. We present the design in Section II and show evaluation results in Section III. Section IV surveys related work and Section V summarizes our conclusions.

II. Design

We design VOLUME, a distributed transactional virtual memory system with persistency support, as the substrate of data handling in datacenters. VOLUME provides a unified memory space, and virtualizes distributed physical memory and disks from compute nodes connected through the network. In addition to the physical memory on the local node, the data in VOLUME may reside in the local disks, remote physical memory or remote disks. The data distribution and communication are transparent to programs running in VOLUME. The data access is as fast as using physical memory when the data is on local physical memory. The data size can scale to the combined capacity of all memory and disks in the system, and VOLUME automatically optimizes the system latency by storing and servicing data from memory as much as possible.

In VOLUME, tasks of a program share the same memory space, access their data structures during the execution and make the data persistent if it is needed with normal memory operations, following a snapshot-based transactional semantics. In our implementation, we dispatch tasks to processors for execution using Layer Zero [12] which can manage hundreds of hosts in datacenters to form a big virtual machine and provides a meta-scheduler that schedules tasks to the compute nodes in the system.

A. Distributed Virtual Memory

Fig. 1 shows the architecture of VOLUME which runs on top of one, multiple or many physical hosts. We abstract groups of memory and disk resources to be *virtual memory containers* (VMC). The system has many processors and VMCs. A VMC manages the physical memory and disks in it and provides the virtual memory resource. All the VMCs together form the unified memory system. VMCs handle the resource requests

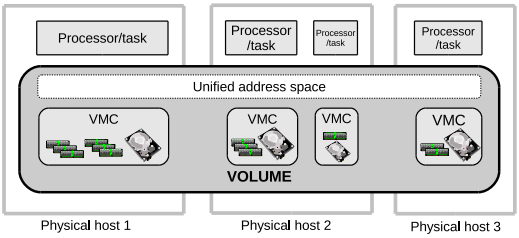


Fig. 1: VOLUME architecture

from the upper layer, and swap data among the physical memory, disks, and remote VMCs through the network.

As many tasks of a program access the same unified memory space of VOLUME concurrently, it is challenging to efficiently control and support the concurrent accesses. VOLUME provides the atomicity and snapshot semantics to programs and implements the consistency model that ensures the same properties as standard snapshot isolation [13] adopted by many relational databases. Each task can use a specific list of pages grouped to be a snapshot which is a set of memory ranges instantiated with the memory state at a particular point of time. Each task's updates to the virtual memory during its execution only affect its own snapshot. A task can only commit its changes in its snapshot to the global memory space when it exits. Hence, tasks are isolated and each has a consistent view of the memory space. After a task exits and commits successfully, its changes are visible to other tasks whose snapshots are created after the commit. If a task's commit has any write-write conflict with other tasks', the commit fails and the task is aborted. Major parts of workloads in datacenters, such as MapReduce jobs, usually can be organized to tasks that have no write-write conflicts and the rate of failed commits is low. To tune the performance of programs, programmers can follow the similar rules for traditional transactional memory systems [14], such as "large transactions are preferable" and "small transactions should be used when violations are frequent".

For managing the memory space and snapshots, VOLUME has a home service to coordinate the snapshots and each task has its own page table. The home service is a centralized service which maintains the metadata of the memory pages in the whole memory space for creating snapshots and guarantees the atomicity of commit operations, acting like the directory controller for directory-based distributed shared memory systems [15], [16]. Different from these systems, the home service of VOLUME manages only 2 versions of the metadata for each page, and snapshot creating and committing only happen when the meta-scheduler creates, starts and terminates tasks, as in the previous work which shows that this approach is efficient and scalable [9], [17].

To assist a task's accesses to the memory space, the VMC for the processor/task makes use of the virtual memory hardware of the host to capture the page fault when the task accesses a memory location on a page that is not used before, similar to demand paging. Different from demand paging and traditional distributed shared memory, VOLUME designs placement and eviction mechanisms for both local and remote memory and both local and remote disks. When handling a page fault, the VMC searches this page in the task's page table which is prepared before the task's execution according to its snapshot and stores the location information of the pages. A task that accesses pages out of its snapshot is aborted. If the page is in the page table, the VMC for this task sets up the page, retrieves the content of the page locally or from a remote VMC, and updates the page table entries. Multiple tasks that use the same memory pages can have multiple copies of the pages and access these pages concurrently.

Locality and scheduling. To better control the scheduling of tasks and improve the locality, we add and implement two scheduling algorithms—the deterministic scheduling and the delay scheduling algorithms—to Layer Zero's meta-scheduler which uses a FIFO scheduling algorithm to assign tasks in the task queue to available processors by default. The deterministic scheduling algorithm assigns tasks to processors by hashing the IDs of tasks to processor IDs. With this algorithm, the scheduling for tasks to processors is deterministic and remains the same for repeated executions of a program, which makes it easier to analyze the behavior of the program. Besides providing repeatability, the deterministic scheduling algorithm also gives programmers an instrument to control the scheduling policy of the meta-scheduler. In general, it also improves the locality compared to the FIFO scheduler for jobs that run repeatedly. The more sophisticated delay scheduling algorithm uses the delay scheduling techniques [18] and may skip some tasks from the front of the task queue and assign the task that has better locality on an available processor/VMC. The locality information is collected from the memory range usage history of tasks according to their snapshots when the meta-scheduler dispatches tasks to processors. If a task at the head of the queue has been skipped for a certain period of time, it is dispatched to a processor even if the locality is poor.

B. Persistency

Data-intensive computation inevitably requires a way to make the data persistent. Programs need a mechanism to make some of its data persistent for storing results which may be used by other programs at a later time. Files are the common form of persistent data stored on disks. However, the programmers need first convert the in-memory data structures to a format suitable for a file and copy them from the memory to file systems. The data preparation, system calls and I/O lead to extra performance overhead in both the programs generating the persistent data and the ones using them.

To provide programmers a flexible and efficient interface to store persistent data, we design the *persistent memory* mechanism in VOLUME. The persistent memory is a memory region which programs can access through regular memory operations. In this way, the distinction between in-memory and persistent data structures is eliminated, and the same set of operations can be applied to both kinds of data.

To make the state changes of the persistent memory easy to reason about by programmers, the memory ranges that a task

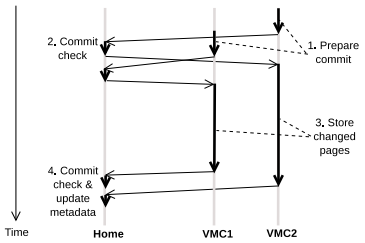


Fig. 2: VOLUME commit protocol. Two commits proceed concurrently.

accesses in the persistent memory are also in its snapshot. After the task commits successfully, the changes to the persistent memory by a task in its snapshot are guaranteed to be persistent on disks. Another task can access and update data in the persistent memory deterministically at a later time.

As the disks are unified in the memory space with the physical memory, we should design the mechanism that can efficiently serve tasks' memory accesses in the persistent memory with data on local or remote disks. For efficiently finding the data on disks, VOLUME maintains a one-to-one mapping of pages in the memory space to blocks (pages) on disks and the page fault handling mechanism for non-persistent memory ranges can be reused. When a task reads a memory range in the persistent memory for the first time, a page fault is triggered and captured by the VMC. Different from handling page faults for non-persistent memory pages, the VMC sets up the page and retrieves the content from local or remote disks. After being committed successfully, the memory pages in a task's snapshot are stored in the corresponding disk blocks.

C. Atomic commits

One possible way to implement atomic commits is to let the home service decide whether a commit succeeds or aborts without requiring a distributed atomic commit protocol [19], [20]. The VMC for handling the commit first stores copies of the changed pages locally, and then the home checks whether the commit can be applied and changes the metadata if the commit passes the check.

However, as VOLUME unifies the physical memory and disks, there are challenges for supporting atomic commit efficiently. To handle possible faults of the home, the changes to the persistent memory's metadata should be made persistent atomically if the commit succeeds. Additionally, VOLUME requires saving the changes in the persistent memory to disks which takes more time than storing the data in physical memory. If a commit is aborted by the home, simply saving the changes to disks in the persistent memory before committing is wasted and should be avoided.

Hence, we design the commit protocol of VOLUME, as shown in Fig. 2, that detects and aborts commits that conflict with others early and enables the recovery of the metadata of persistent memory if the home service crashes or restarts by storing changes to the persistent memory's metadata to a write-ahead log. Specifically, the protocol for a task's commit consists of the following steps.

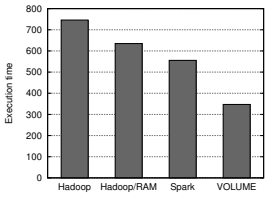
1) The VMC prepares the commit by collecting the metadata of updated pages by the task in its snapshot, and queries the home whether the commit can proceed.
2) The home decides whether the commit should be aborted by checking the in-memory metadata and returns the result to the VMC. This can abort the commits that have conflicts with other commits early and avoid useless I/O to disks (*early abort*).
3) If the commit is aborted by the home, the VMC discards the changes. Otherwise, the VMC saves changed pages in the persistent memory to disks, stores copies of changed pages in another memory area, and submits a commit request to the home.
4) The home checks the commit request with the *latest* metadata again. If the commit passes the checking, the home allocates an ID for this commit, writes "*start-commit ID*" and the metadata modifications for the persistent memory to the write-ahead log, applies the modifications to the in-memory metadata and logs "*complete-commit ID*" to complete the commit.

With this protocol, the changes made by a task are committed atomically. Any commit that fails to complete the step that the "*complete-commit*" log entry is written is aborted when the home restarts. All successful commits exhibit a serial order according to the corresponding "*complete-commit*" log entries. When a snapshot is created, it includes the pages updated by the successful commits in the log. Since the pages are stored before the log entries are written, the snapshot includes the latest pages committed successfully. In the commit protocol, only a very small part of the total work for a commit is on the home. Hence, the home can handle many commit requests efficiently.

Using the transactional semantics of VOLUME, a program can also ensure that only one task commits its changes successfully when multiple tasks conduct the same work, such as the backup executions in MapReduce [2]. These tasks can share a *status code* s which is the value in a memory range that indicates whether a task has completed successfully. The task, say, T_s, which exits and commits first will commit successfully at time t_s, and updates s. The other tasks for this work are in two classes, both of which will abort:

- The task, say, T_1, created its snapshot before t_s. s in T_1's snapshot indicates the task is not completed successfully, yet and T_1 proceeds with the computation, updates s and tries to commit. As there is write-write conflict between T_s and T_1's commits on s, T_1 aborts.
- The task, say, T_2, created its snapshot after t_s. s in T_2's snapshot indicates the task is already done and T_2 aborts.

With the atomic commits, VOLUME handles faults gracefully and provides an easy-to-reason semantics that the memory ranges in the persistent memory contain the content updated by the latest successfully committed snapshots. VOLUME handles faults from the home service or VMCs by restarting them. The home service recovers the metadata for the persistent memory from the write-ahead log and the VMCs continue servicing the pages according to the mapping between the pages in the persistent memory and the blocks of disks on the hosts where the VMCs reside. This semantics makes it easy



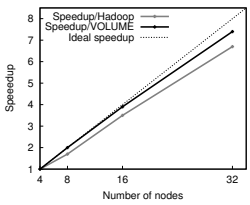
(a) Execution time

	Input /Map	Sort+output /Reduce
Hadoop	465s*	281s
Spark/RDD	47s	509s (287s+222s)
VOLUME	56s	291s

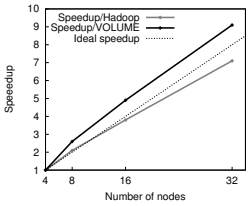
∗ 14% of shuffle phase included.

(b) Time breakdown

Fig. 3: Execution time and breakdown of sorting on 16 nodes



(a) Word count



(b) Adjacency list

Fig. 4: Speedups on VOLUME and Hadoop

for programmers to design programs to handle the faults in the cluster and possibly recover the execution of long jobs after faults to avoid complete re-runs—the content in the persistent memory is always consistent and programs can store necessary state for recovery in it. In our current implementation, we rely on the underlying operating systems and storage systems (e.g. RAID) for the data availability. In the future work, we may replicate the committed data to other VMCs for higher availability and faster recovery.

III. Implementation and evaluation

We implement VOLUME on a cloud computing research testbed and the TH-1/GZ supercomputer which follows the architecture of Tianhe-1A [21]. VOLUME is implemented in around 11,000 lines of C code. We measure the performance, scalability and overhead of VOLUME, and compare VOLUME with Hadoop and Spark using 4 diverse and widely used workloads: *word count*, a MapReduce benchmark [2], [5], [22]–[25], *k-means* clustering, a data mining workload [26], [27], *adjacency list*, a graph processing workload [28], and *sort*, a benchmark for data processing systems [2], [29]–[31].

Word count counts the number of occurrences of each unique word in a set of documents, which can also represent a large subset of real-world MapReduce jobs that extracts a small amount of interesting data from a large dataset [2]. The dataset for the *word count* is the 15GB text content of the static HTML dump of English Wikipedia in June of 2008 [32].

K-means iteratively partitions a dataset into k clusters [33], which is one example of the set of algorithms that require iterative computation. The datasets are a large number of 4-dimensional data points which are randomly generated by the standard *gensort* record generator for TeraSort [34]. We cluster the data points into 1000 groups and make each k-means experiment compute for 20 iterations.

Adjacency list generates the adjacency and reverse adjacency lists of nodes in a graph with the edges as its input, which is a shuffle-heavy workload that transfers data more than twice the size of the input across the network. Two input datasets are generated following the power law as in the related work [28], and contain around 100 million (30GB) and 500 million (150GB) nodes with an average out-degree of 7.2.

Sorting is an important part of many computing tasks. We generate *sort*'s input consisting of 100-byte records with 10-byte keys by a program, *teragen*, which produces the same dataset for the TeraSort benchmark [34].

Following identical algorithmic designs suitable for the programming models, we implement the benchmark programs or use standard implementations on VOLUME, Hadoop and Spark [35]. The implementations on VOLUME follow the MapReduce programming model including many map tasks in the map phase and many reduce tasks in the reduce phase. The evaluation is conducted on the research testbed using 32 compute servers and on TH-1/GZ using 160 compute nodes. To make the performance comparison fair, we use the same dataset and equivalent job configurations for Hadoop, Spark and VOLUME. We also configure the HDFS to store only one copy of data to avoid additional cost of data replication. In fact, the configuration slightly favors Hadoop which requires static slot allocation on each compute node for map and reduce tasks. We allocate 1 map task slot and 1 reduce task slot on each node for Hadoop, but run only 1 task on each compute node for VOLUME and Spark. This configuration also minimizes the I/O contention to which Hadoop is more sensitive.

A. Performance of unifying physical memory and disks

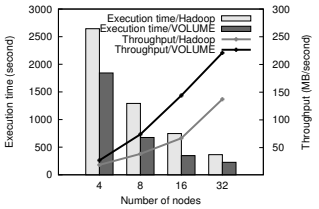
As VOLUME unifies the physical memory and disks on many nodes connected through the network, we measure its performance with different sizes of datasets that can be processed totally or partially in physical memory.

We first run *sort* on 50GB input using 16 nodes to evaluate VOLUME's performance when the intermediate dataset can be processed in physical memory. We also run *sort* on both Hadoop and Spark for comparison. The input is on disks managed by HDFS for Hadoop and Spark or the persistent memory for VOLUME, and the output is written back to disks. To emulate a solution that uses the file system based systems and stores data in memory, we also set up Hadoop in the Linux ramdisk */dev/shm*. As shown in Fig. 3(a), both Spark and VOLUME outperform the file system based Hadoop that stores data on disks or in memory (Hadoop/RAM). These three systems share the similar overall structure of the execution organization and we break down the time as shown in Fig. 3(b). We also run *k-means*, *word count* and *adjacency list* on the research testbed to evaluate the scalability and performance of VOLUME on datasets of moderate sizes. Fig. 4 and Fig. 5 show the execution time, throughput and speedups. As shown in the results, VOLUME scales better and exhibits superiority in efficiency than the file system based solution.

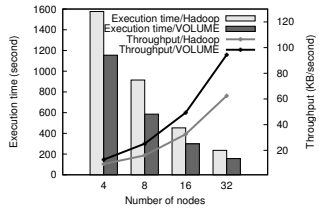
Understanding the performance. VOLUME accelerates the computation in several aspects: low-latency operations on the intermediate data and small system overhead, efficient sharing of data among tasks and jobs and efficient support to input and output. We find two of those are most effective. First, VOLUME transparently helps the program save part of the input and intermediate data generated by map tasks

Number of nodes	Hadoop (second)	VOLUME (second)
8	27261.75	3610.83
16	14295.00	1882.50
32	7835.82	1032.26

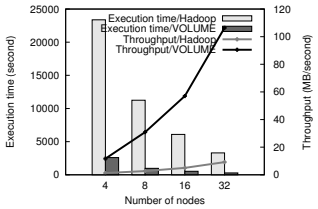
(a) Execution of *k-means* clustering on 30 million points



(b) Execution time of sorting 50GB data

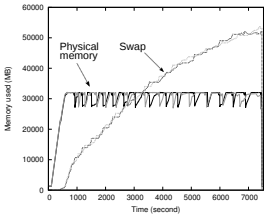


(c) Execution time and throughput of *word count* on 15GB input

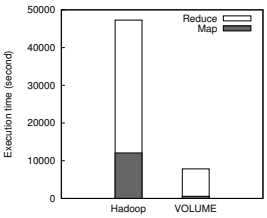


(d) Execution time and throughput of *adjacency list* on 30GB input

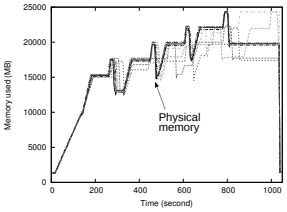
Fig. 5: Performance of VOLUME on datasets of moderate sizes



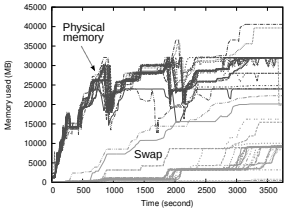
(a) Adjacency list on 2 nodes



(b) Execution time of *adjacency list* on 2 nodes



(c) Adjacency list on 8 nodes



(d) Sort 400GB data on 32 nodes

Fig. 6: VOLUME memory usage

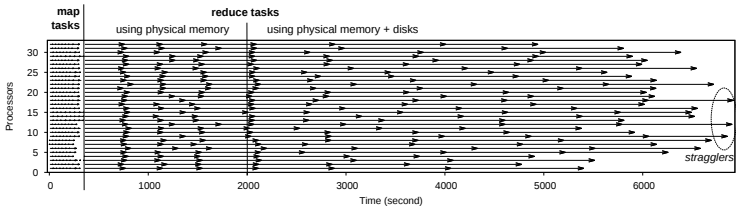
in memory and significantly accelerates data exchange in the shuffle phase. For *sort*, VOLUME and Spark/RDD that store intermediate data in memory [11] deliver more than 8 times higher performance for the map or the input phase than Hadoop that stores the intermediate data in local file systems. For the shuffle-intensive *adjacency list* workload, VOLUME delivers 6 to 11 times higher throughput. For the iterative workload *k-means*, VOLUME stores the input dataset (after the first iteration) and centroids in memory and delivers 7.5 times as high performance as that on Hadoop. Second, VOLUME supports efficient saving of the results by allowing programs to directly store data structures in the persistent memory and commit the changes without the need of converting objects in files from the file system to in-memory data structures and vise versa. The sort program on RDD first stores the sorted data in an RDD and then saves it to HDFS. From Fig. 3(b), VOLUME finishes the sorting and output emission in the similar amount of time that Spark uses for sorting data and generating the RDD of the sorted data. Spark takes additional 222 seconds to store the data to HDFS. Through these mechanisms, VOLUME delivers significant higher performance, especially for iterative workloads and workloads that generate intermediate data much larger than the input data, and also improves the performance of workloads that are computation or disk I/O intensive, such as *word count* and *sort*.

To evaluate VOLUME's performance with different amounts of data stored on disks, we stress it by running *adjacency list* on 2 compute nodes that have 64GB physical memory in total which is far less than the overall data in VOLUME during the execution of *adjacency list*. We examine the memory usage and compare the execution time of *adjacency list* on VOLUME and Hadoop. Fig. 6(a) shows the VOLUME usage on 2 nodes. We can see that the physical memory usage increases to around 64GB quickly in around 665 seconds. After that the usage of disk space keeps increasing on both nodes at a lower rate than the previous ones. This shows that, when the data size exceeds the memory capacity, VOLUME transparently swaps data between DRAM and hard disks, providing a lower data access speed, but ensures the success of the computation. Fig. 6(b) shows the execution time. Although disks are heavily used by VOLUME to store data, VOLUME exhibits more than 6 times higher performance than Hadoop. The map phase on VOLUME uses less than 1/20 of the time than Hadoop's map phase takes, which shows the efficiency of data inputting and intermediate data handling in VOLUME. The reduce phase on VOLUME takes much less time than that needed by the reduce phase on Hadoop. This shows that the cost for memory swapping in VOLUME is smaller than the file system based solution in Hadoop. As shown in Fig. 6(c), when there are more than 8 compute nodes, all the data can fit into the physical memory, which is why it shows superlinear speedups in Fig. 4(b). Fig. 6 also shows that the memory footprint of workloads in datacenters may be multiple times larger than the input, which requires large capacity of the data substrate. For these workloads, VOLUME is more cost-effective than physical memory based systems.

To check how VOLUME performs when the total memory usage is much larger than the overall physical memory and disks are extensively used at a large scale, we scale the input size for *sort* to 512GB on VOLUME with 32 compute nodes. Fig. 7(a) shows the execution time and throughput. As the dataset scales, the throughput decreases gracefully because the usage of disks increases. We measure the VOLUME usage during running *sort* on 400GB datasets as shown in Fig. 6(d). From the memory usage, we can see that physical memory of the 32 compute nodes is used up and around 300GB disk space is used for swapping in total when sorting the 400GB data. To look into how the usage of disks affect tasks' execution time, we plot the execution time of tasks during sorting 512GB data in Fig. 7(b). Each arrow in the figure represents a task. The map phase consisting of 250 tasks in total completes in 360 seconds, which shows that VOLUME can efficiently service

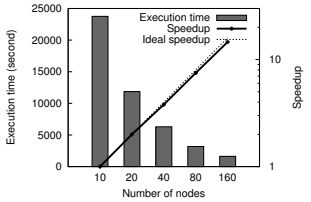
Input size (GB)	Execution time (second)	Through-put (MB/sec)
256	850.7	300.9
320	1969.9	162.5
400	3666.4	109.1
512	6926.6	73.9

(a) Sorting on 32 nodes

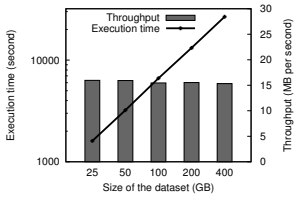


(b) Progress of sorting 512GB data

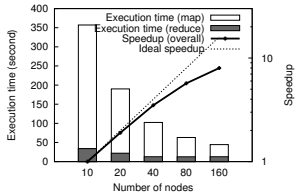
Fig. 7: Execution time, throughput and execution progress of sorting



(a) Execution time and speedup of *k-means* clustering on TH-1/GZ



(b) Execution time and throughput of *k-means* as the dataset grows



(c) Execution time and speedup of *word count* on TH-1/GZ

Platform number of nodes	Execution time (second)	Per-node throughput (MB/sec)
Research testbed		
32 nodes	2039.85	2.30
TH-1/GZ		
84 nodes	752.17	2.37

(d) Execution time of *adjacency list* on 150GB input

Fig. 8: Scalability of VOLUME on large datasets and many compute nodes

the input from disks. After reading the input, the reduce tasks start to run and those started after the 2000^{th} seconds take longer time to complete, which demonstrates that VOLUME can smoothly handle large datasets in disks and gracefully decrease the performance. Fig. 7(b) also shows that there are stragglers. We can start backup executions of them as discussed in Section II-C to further improve the overall performance, which we leave as future work.

B. Scalability

To evaluate VOLUME's scalability on many nodes, we conduct experiments of *k-means* and *word count* on the TH-1/GZ supercomputer using up to 160 compute nodes. Fig. 8(a) shows the execution time and speedup of *k-means* that processes 300 million points. We can see that *k-means* scales linearly as we scale the compute nodes number from 10 to 160. This shows the good scalability of VOLUME. Fig. 8(c) shows the execution time and speedup of *word count*. The execution time decreases as we scale the number of compute nodes and *word count* scales near-linearly on 10 to 40 nodes. However, the speedup is sub-linear on more than 40 nodes and the time for reduce tasks does not change much on 40 to 160 nodes. This is because the number of reduce tasks in *word count* is 30 and the additional nodes do not improve the reduce phase's parallelization. The execution time of the map phase decreases with more nodes and *word count* executes faster.

To evaluate how VOLUME performs as the dataset grows, we run *k-means* with 30 compute nodes on datasets of varying sizes from 25GB to 400GB. As shown in Fig. 8(b), the time increases proportionally to dataset sizes and the throughput remains mostly stable, indicating that VOLUME scales smoothly with the data size.

To examine the efficiency of VOLUME on supporting shuffle-heavy tasks and the performance of VOLUME in HPC environments, we run the *adjacency list* workload with 150GB input. Fig. 8(d) shows the execution time of *adjacency list* in the research testbed and TH-1/GZ. We can see that the per-node throughput on TH-1/GZ is as higher than that on the research testbed, thanks to the high-performance hardware.

TABLE I: Time for writing persistent data in VOLUME and recovering the home service

Data size	Generating	Commit	Recovery
8TB	6874.8s	129.3s	145.0s

C. Storing data persistently and recovery

To investigate the capability of VOLUME to store large data persistently and the time for recovering the VOLUME in the presence of faults, we run *teragen* to perform parallel I/O with 8TB data on 30 nodes and force the home to restart the VOLUME to emulate crashes. As VOLUME maintains a one-to-one mapping of disk blocks to memory pages in the persistent memory, recovering the VMCs is very fast. Hence, we only measure the time for recovering the home.

TABLE I shows the time for generating the datasets and the time used by the home service for committing snapshots and recovering the metadata after being restarted. Although much of the time is for generating the random input, the per-node throughput for generating 8TB data is 38.8MB/s which is about 50% of the raw disk I/O throughput in our cluster. The time used by the home service for committing the 8TB data changes takes 2.8% of the overall execution time, delivering a 61.9GB/s throughput. This shows that the home service is very efficient to support commit requests at a high aggregate rate. For recovery, the home takes 145 seconds for recovering the metadata from the log for the persistent memory storing 8TB data. Applying certain techniques, such as checkpointing, can improve the recovery speed, which we leave as future work.

IV. Related work

Unifying physical memory from a cluster of servers is studied for many years. Examples include distributed shared memory systems such as Ivy [36] and Treadmarks [37], virtual machines, such as vNUMA [38] and DVM [9], operating system extensions, such as GMS [16], caching systems such as Memcached [39], and storage systems, such as RAMCloud [7]. Researchers suggest using the main memories from many servers in the datacenters to create the storage system [7] and some MapReduce implementations make use of the main memories on working nodes, such as Twister [40], Phoenix [24],

MRlite [6] and EMR [41]. On the other hand, the capacity of the main memories on the commodity servers are far smaller than the disks' and the storage capacity of these systems is consequently limited. Although a memory-based storage system with large capacity can be built with a large number of servers, the cost is very high and the utilization of the processor resources on these servers is possibly low most of the time. VOLUME unifies the physical memory and disks in a memory space, provides a transactional semantics, supports storing data persistently, and provides a strong substrate for general, efficient and scalable parallel computation systems.

Emerging data processing systems, such as Spark/RDD [11] and Piccolo [42], handle the datasets using in-memory data structures and deliver higher performance than file system based systems. VOLUME provides a different way for organizing data in memory. For example, programs on Spark can organize data as RDDs in a lineage graph and apply transformations or actions on RDDs. Programs on VOLUME store data in the memory space, update the data in snapshots and commit the changes. The persistent memory of VOLUME provides a way to make the data persistent for data-intensive computation in datacenters. Programs can make some of its data persistent for storing results which may also be used by other programs at a later time.

In recent years, MapReduce-style systems are becoming increasingly popular for a set of applications [2], [22], [26]. The large data processed are usually stored and shared as files in distributed or local file systems. GFS is used as the substrate of the MapReduce in Google's datacenter. Over the past decades, the performance of disks has not improved so quickly as the disk capacity has [7]. The disk performance is becoming a bottleneck of the MapReduce-like systems, especially for workloads with datasets of moderate sizes. Different from the file system based approaches, VOLUME builds a distributed virtual memory which unifies the physical memory and disk resources as the substrate for computation in datacenters.

V. Conclusion

We construct VOLUME, a distributed virtual memory as a system-wide data substrate. VOLUME provides a general memory-based abstraction with transactional semantics, takes advantage of DRAM in the system to accelerate computation, and, transparent to programmers, scales the system to handle large datasets by swapping data to disks and remote servers for general-purpose computation in datacenters.

References

[1] "Apache Hadoop," *http://hadoop.apache.org*.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI'04*, 2004.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP'03*, 2003, pp. 29–43.

[4] L. Barroso, J. Dean, and U. Hoelzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[5] R. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," in *IEEE Intl. Sympsium on Workload Characterization*, 2009, pp. 198 –207.

[6] Z. Ma and L. Gu, "The limitation of MapReduce: A probing case and a lightweight solution," in *CLOUD COMPUTING 2010*, 2010.

[7] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: scalable high-performance storage entirely in DRAM," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 92–105, January 2010.

[8] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," University of California, Berkeley, Tech. Rep. UCB/EECS-2010-53, May 2010.

[9] Z. Ma, Z. Sheng, L. Gu, L. Wen, and G. Zhang, "DVM: Towards a datacenter-scale virtual machine," in *VEE'12*, 2012.

[10] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011, pp. 29–41.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12*.

[12] "Layer Zero," *http://www.lazero.net*.

[13] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. of SIGMOD'95*, 1995, pp. 1–10.

[14] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *SIGARCH Comput. Archit. News*, vol. 32, no. 2, 2004, p. 102.

[15] B. Nitzberg and V. Lo, "Distributed shared memory: a survey of issues and algorithms," *Computer*, vol. 24, no. 8, pp. 52–60, 1991.

[16] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing global memory management in a workstation cluster," in *SOSP '95*, 1995, pp. 201–212.

[17] Z. Ma, Z. Sheng, and L. Gu, "DVM: A big virtual machine for cloud computing," *IEEE Transactions on Computers*, no. 99, April 2013.

[18] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys'10*, 2010, pp. 265–278.

[19] R. Guerraoui, "Non-blocking atomic commit in asynchronous distributed systems with failure detectors," *Distributed Computing*, vol. 15, no. 1, pp. 17–25, 2002.

[20] J. Stamos and F. Cristian, "A low-cost atomic commit protocol," in *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, 1990, pp. 66–75.

[21] X. Yang, X. Liao, K. Lu, Q. Hu, J. Song, and J. Su, "The TianHe-1A supercomputer: its hardware and software," *Journal of Computer Science and Technology*, vol. 26, no. 3, pp. 344–351, 2011.

[22] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for data intensive scientific analysis," in *Fourth IEEE Intl. Conf. on eScience*, 2008, pp. 277–284.

[23] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *ICDEW'10*, march 2010, pp. 41 –51.

[24] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *HPCA'07*, 2007.

[25] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *OSDI'08*, 2008, pp. 29–42.

[26] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for machine learning on multicore," in *NIPS'07*, 2007, pp. 281–288.

[27] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on MapReduce," in *CloudCom'09*, 2009, pp. 674–679.

[28] M. T. Faraz Ahmad, Seyong Lee and T. N. Vijaykumar, "MapReduce benchmarks," *http://web.ics.purdue.edu/~fahmad/benchmarks.htm*.

[29] "Sorting 1PB with MapReduce," *http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html*.

[30] O. O'Malley and A. Murthy, "Winning a 60 second dash with a yellow elephant," *Tech report, Yahoo!*, 2009.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI'08*, 2008, pp. 1–14.

[32] "Wikimedia Downloads," *http://dumps.wikimedia.org/*.

[33] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 281-297, 1967, p. 14.

[34] "Sort Benchmark," *http://sortbenchmark.org/*.

[35] Z. Ma, K. Hong, and L. Gu, "MapReduce-style computation in distributed virtual memory," HKUST, Tech. Rep. HKUST-CS14-03, 2013.

[36] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.

[37] P. Keleher, A. Cox, S. Dwarkadas, and W. Treadmarks, "Distributed shared memory on standard workstations and operating systems," in *Proc. 1994 Winter Usenix Conf.*, 1994, pp. 115–131.

[38] M. Chapman and G. Heiser, "vNUMA: A virtual shared-memory multiprocessor," in *USENIX ATC'09*, 2009.

[39] "Memcached," *http://memcached.org/*.

[40] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *HPDC'10*, 2010, pp. 810–818.

[41] K.-T. Rehmann and M. Schoettner, "An in-memory framework for extended MapReduce," in *ICPADS'11*, 2011, pp. 17–24.

[42] R. Power and J. Li, "Piccolo: building fast, distributed programs with partitioned tables," in *OSDI'10*, 2010, pp. 1–14.