LARGE-SCALE IN-MEMORY DATA PROCESSING

by

ZHIQIANG MA

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

August 2014, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

ZHIQIANG MA

LARGE-SCALE IN-MEMORY DATA PROCESSING

by

ZHIQIANG MA

This is to certify that I have examined the above Ph.D. thesis

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by

the thesis examination committee have been made.

PROF. LIN GU, THESIS SUPERVISOR

PROF. SIU-WING CHENG, ACTING HEAD OF DEPARTMENT

Department of Computer Science and Engineering

August 2014

ACKNOWLEDGMENTS

This thesis could not be finished without the help and support from many people who are gratefully acknowledged here.

My first and foremost thanks go to my supervisor, Prof. Lin Gu, for his insightful guidance on my research, tremendous amount of time and energy on training me into a systems researcher and continuous and unconditional support to both my study and my life in the past five years. His strong understanding of systems and patient advice bring me to the beautiful world of systems research. Besides research, his rich industrial experience and connections broadened my perspective. I could not have imagined having a better advisor and mentor for my PhD study and I can never repay Prof. Gu for what he has given to me.

Besides my supervisor, I am also grateful to every member in my thesis defense, thesis proposal and qualifying examination committees, Prof. Shueng-Han Gary Chan, Prof. Pak Wo Leung, Prof. Cho-Li Wang, Prof. Ke Yi, Prof. Kai Chen, Prof. Qian Zhang, Prof. Qiong Luo and Prof. Lionel M. Ni, for their kind help and insightful comments on my research.

I also would like to express my appreciation for the support of my friends at the Hong Kong University Science and Technology. Ke Hong, Zhonghua Sheng, Ang Li, Tong Zhu, Mengmeng Cheng, Xinjie Yu, Xiaofei Zhang, Xiaojun Feng, Liping Gao, Jeff Shaoming Huang, Derek Hao Hu, Yang Peng, Xiaonan Guo, Mian Lu, Da Yan, Weiwei Jia and Ping Xu should not be forgotten.

At last but not least, I would like to thank my family for their unwavering support all the way from the very beginning of my PhD study. Their sincere love strengthened me to complete this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LARGE-SCALE IN-MEMORY DATA PROCESSING

by

ZHIQIANG MA

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

As cloud and big data computation grows to be an increasingly important paradigm, providing a general abstraction for datacenter-scale programming has become an imperative research agenda. Researchers have proposed, designed and implemented various computation models and systems on different abstraction levels, such as MapReduce, X10, Dryad, Storm and Spark. However, many abstractions expose the distributed detail of the platform to the application layer, and lead to increased complexity in programming, decreased performance, and, sometimes, loss of generality. At the data substrate layer, traditional cloud computing technologies, such as MapReduce, use disk-based 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 processing and storage in the datacenter environment, and the capacity of such systems is limited by the amount of physical memory in the cluster.

To support general, efficient, flexible, and concurrent application workloads with sophisticated data processing, we present programmers an illusion of a big virtual machine built on top of one, multiple or many compute nodes and unify the physical memory and disks of the nodes to form a globally addressable data substrate. We design a new instruction set architecture, i0, to unify myriads of compute nodes to form a big virtual machine called MAZE where thousands of tasks run concurrently in VOLUME, a large, unified, and snapshotted distributed virtual memory. i0, MAZE and VOLUME form the foundation of the Layer Zero systems which provide a general substrate for cloud computing. i0 provides a simple yet general and scalable programming model. VOLUME mitigates the scalability bottleneck of traditional distributed shared memory systems and unifies the physical memory and disks on many compute nodes to form a distributed transactional virtual memory. VOLUME provides a general memory-based abstraction, takes advantage of DRAM in the system to accelerate computation, and, transparently to programmers, scales the system to process and store large datasets by swapping data to disks and remote servers. Along with the efficient execution engine of MAZE, the capacity of a MAZE can scale up to support large datasets and large clusters.

We have implemented the Layer Zero systems on several platforms, and designed and implemented various benchmarks, graph processing and machine learning programs and application frameworks. Our evaluation shows that Layer Zero has excellent performance and scalability. On one physical host, the system overhead is comparable to that of traditional VMMs. On 16 physical hosts, Layer Zero runs 10 times faster than Hadoop and X10. On 160 physical compute servers, Layer Zero scales linearly on a typical iterative workload.

CHAPTER 1

INTRODUCTION

Cloud computing is an emerging computing paradigm, which is increasingly popular and strongly promoted in the industry in recent years. Nowadays, the most popular applications are Internet services with millions of users and the computation is large and complex. Cloud computing becomes a promising model to support processing these large datasets housed on clusters with commodity hardware.

1.1 Background and motivation

Cloud computing provides the users massive computing resources which are available on demand and cloud-based computation is becoming increasingly important today. With almost ubiquitous Internet accesses, users can store data and conduct a large part of their computation in datacenters, and more and more applications run in the cloud. However, writing efficient distributed parallel applications is complex and challenging.

1.1.1 Organization of computation

Virtualization is a fundamental component in cloud technology. Particularly, ISA-level virtual machine monitors (VMM) are used by major cloud providers for packaging resources, enforcing isolation [1, 16], and providing underlying support for higher-level language constructs [21]. However, the traditional VMMs are typically designed to multiplex a single physical host to be several virtual machine instances [25, 62, 88]. Though management or single system image (SSI) services can coordinate multiple physical or virtual hosts [30, 74, 88], they fall short of providing the functionality and abstraction that allow

users to develop and execute programs as if the underlying platform were a single big "computer" [27]. Extending traditional ISA abstractions beyond a single machine has only met limited success. For example, vNUMA extends the IA-64 instruction set to a cluster of machines [33], but finds it necessary to simplify the memory semantics and encounters difficulty in scaling to very large clusters.

On the other hand, exposing the distributed detail of the platform to the application layer leads to increased complexity in programming, decreased performance, and, sometimes, loss of generality. In recent years, application frameworks [42, 59] and productivity-oriented parallel programming languages [23, 34, 78, 92] have been widely used in cluster-wide computation. MapReduce and its open-source variant, Hadoop, are perhaps the most widely-used framework in this category [67]. Without completely hiding the distributed hardware, MapReduce requires programmers to partition the program state so that each map and reduce task can be executed on one individual host, and enforces a specific control flow and dependence relation to ensure correctness and reliability. This leads to a restricted programming model which is efficient for "embarrassingly parallel" programs, where data can be partitioned with minimum dependence and thus the processing is easily parallelizable, but makes it difficult to support sophisticated application logic [47, 67, 80, 89]. The language-level solutions, such as X10 [34], Chapel [31] and Fortress [23], are more general and give programmers better control over the program logic flows, parallelization and synchronization, but the static locality, programmer-specified synchronization, and the linguistic artifacts influence such solutions to perform well with one set of programs but fail to deliver performance, functionality, or ease-of-programming with another. It remains an open problem to design an effective system architecture and programming abstraction to support general, flexible, and concurrent application workloads with sophisticated processing.

1.1.2 Data substrate

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.

A major cause of the limitation is the way current cloud systems store, distribute, manipulate and persist data. Traditionally, cloud systems use the file system abstraction as the basic system substrate for storing data and distributing them to compute nodes for processing. The MapReduce framework, for instance, is built on the GFS file system [51], and assisted by the local file systems on individual compute nodes for servicing input and intermediate data. Similarly, Hadoop, the open-source variant of MapReduce, relies on HDFS as the data substrate [2]. This enables the system to scale, but makes it slow and difficult to support interactive, real-time or sophisticated computation. Although key/value and table based abstractions, which are typically built upon the file system [32, 44], are often provided to programmers, disk-based technology is still the foundation for data storage and exchange in cloud systems. Not surprisingly, the file system I/O increases the overhead of data accesses, and makes the system operations intrinsically high-latency. This fact, in turn, invites system designers to assume high latency in many system design points and favor high throughput more than low latency, resulting in systems tailored for large offline data analytics where input and intermediate data can be naturally partitioned but unsuitable for many other applications and 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 [26]. 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 [91]. In 2010, MRlite implemented in-memory MapReduce computation on multiple compute servers [67], RAMCloud proposed to build DRAM-based storage [77], and Spark is proposed for in-memory computation across a number of compute servers [95]. Today

several DRAM-based cluster computing systems, including DVM [69, 70], RAMCloud [76] and Spark/RDD [94], 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 [77]. 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 physical 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.

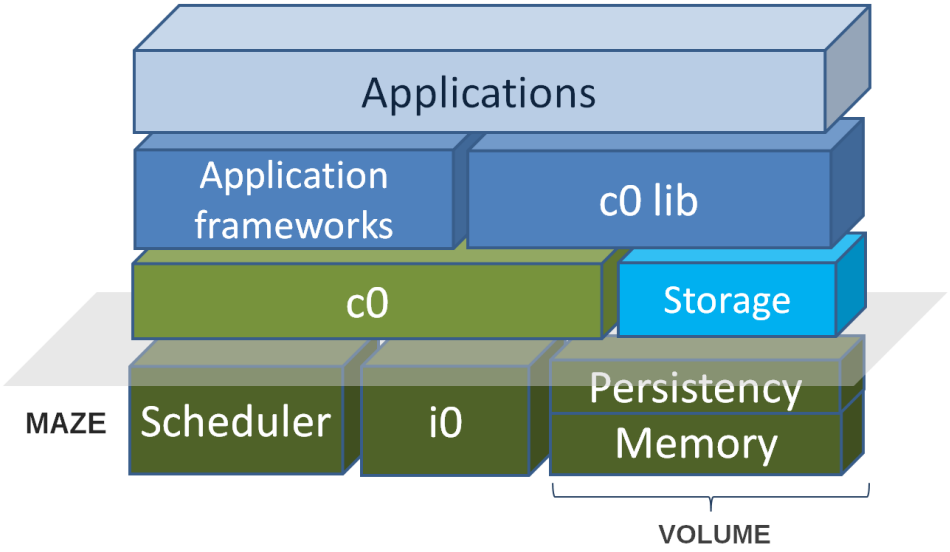


Figure 1.1: Layer Zero technologies

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 memory 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.

1.2 Contributions

As discussed in Section 1.1, it is challenging both for the programmers to write cloud and big data computation programs and for the execution engines to efficiently run them on a large number of computers in datacenters, particularly when the applications evolve from effortlessly parallelizable to sophisticated with complex dependence. This thesis presents the work on the fundamental components of the Layer Zero technologies [12] as shown in

Figure 1.1 to address the problem. This thesis work covers the design, implementation and evaluation of i0, MAZE and VOLUME.

- **i0** An instruction set for distributed computing. i0 abstracts the resource of many hosts.
- **MAZE** A big virtual machine that can execute programs in i0 across many physical hosts.
- **VOLUME** A snapshotted distributed transactional virtual memory system where an i0 program's tasks execute in.

MAZE unifies many physical hosts in datacenters to a big virtual machine by i0 for datacenter-scale programming. Layer Zero also provides languages and tools for designing programs to run on MAZE. c0 is a variation of the C programming language with some extensions to support language-level multi-tasking and other mechanisms on i0 and MAZE [4]. cc0 [6] is a compiler for c0 and generates i0 code.

Specifically, this thesis work contributes in the following aspects.

i0 and MAZE Considering that ISAs indeed provide a clearly defined interface between hardware and software and allow both layers to evolve and diversify, we believe it is effective to unify the computers in a datacenter at the instruction level, and present programmers the view of a big virtual machine that can potentially scale to the entire datacenter. The key is to overcome the scalability bottleneck in traditional instruction sets and memory systems. Hence, instead of porting an existing ISA, we design a new ISA i0, for cluster-scale computation. Through its instruction set, memory model and parallelization mechanism, the i0 architecture presents the programmers an abstraction of a large-scale virtual machine, called the MAchine ZEro (MAZE), which runs on a plurality of physical hosts inside a datacenter. It defines an interface between the MAZE and the software running above it, and ensures that a program can execute on any collection of computers that implement i0.

Different from existing VMMs (e.g., VMware [88], Xen [25], vNUMA [33]) on traditional architectures (e.g., x86), MAZE and i0 make design choices on the instruction set,

execution engine, memory semantics and dependence resolution to facilitate scalable computation in a dynamic environment with many loosely coupled physical or virtual hosts. There are certainly many technical challenges in creating an ISA scalable to a union of thousands of computers, and backward compatibility used to constrain the adoption of new ISAs. Fortunately, the cloud computing paradigm presents opportunities for innovations at this basic system level—first, a datacenter is often a controlled environment where the owner can autonomously decide internal technical specifications; second, major cloud-based programming interfaces (e.g., RESTful APIs, RPC, streaming) require minimum instruction-level compatibility; finally, there is only a manageable set of "legacy code".

VOLUME and vMR To support the execution of programs written in or compiled to i0, we design a distributed virtual memory system called VOLUME (Virtual On-Line Unified Memory Environment) to unify physical memory 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 technology, e.g., MapReduce, and immediately enhance its functionality and performance. Designed as a general-purpose data substrate, VOLUME can also replace the file system based data substrate of other cluster computing systems, such as MapReduce, and accelerate computa-

tion without losing the functionality, economy and scalability provided by the file systems. To verify this and to improve the widely used MapReduce computation model, we have designed a new MapReduce framework named vMR, which stores and exchanges data on VOLUME. Maintaining the scalability to handle large datasets, vMR demonstrates a flexibility to handle more complex computational logic and an efficiency comparable to recent in-memory cluster computing systems.

A large computation in a datacenter can potentially scale to thousands of concurrent tasks on hundreds of nodes. To the best of our knowledge, MAZE based on i0 is the first virtual machine to organize large-scale parallel computation in datacenters, scale to many nodes and datasets larger than the overall physical memory capacity and provide the abstraction and mechanism for dependence representation at the task level and automatic dependence resolution, and VOLUME is the first "virtual memory" system that coordinates transparent memory accesses to distributed physical memory and disks at such a scale. The persistency mechanism in VOLUME advances the memory-based cloud technology to systematically specify, store and read persistent data. To the best of our knowledge, VOLUME is the first system to provide transactional memory semantics on virtualized memory among hundreds of computers.

We have implemented the i0, MAZE and VOLUME on several platforms, and designed and implemented various benchmarks, graph processing and machine learning programs and application frameworks. Our evaluation shows that Layer Zero has excellent performance and scalability. On one physical host, the system overhead is comparable to that of traditional VMMs. On 16 physical hosts, Layer Zero runs 10 times faster than Hadoop and X10. For in-memory sorting, Layer Zero is faster than both Hadoop and Spark. For k-means clustering, Layer Zero scales linearly to 160 physical compute servers on the TH-1/GZ supercomputer. For processing of large datasets exceeding the overall physical memory capacity of the cluster, Layer Zero is much faster than Hadoop/HDFS and delivers 6-11x speedups on the adjacency list workload.

1.3 Programming on Layer Zero

On Layer Zero, programmers can write programs at levels from i0, programming languages like c0 to programming frameworks like vMR. In this section, we present a simple example program `adders.c0` in c0 to calculate sums in parallel to show how a program on MAZEs looks like. We refer the readers to the online c0 tutorials [5] and the c0 manual [4] for more details of programming in c0. More details of the designs and mechanisms of the MAZE will be discussed in the following sections.

```
 1 // Headers from libi0 ( https://github.com/layerzero/libi0 )
 2 // that declare the ‘putlong()’ and ‘wrln()’ functions to print
 3 // long integers and new lines.
 4 #include <stdio.h>
 5
 6 // The input data shared among tasks. Tasks need to create snapshot on
 7 // these variables by ‘using’ to use them.
 8 // In this example, each of the 2 tasks sums 2 of the 4 integers.
 9 long data[4];
10
11 // Task for adding 2 long integers
12 void adder(long id)
13 {
14     long start, result;
15     start = id * 2;
16
17     // read the input from data[] and calculate the sum
18     result = data[start] + data[start+1];
19
20     // print the result to STDOUT
21     putlong(result);
22     wrln();
23
24     // exit and commit
25     commit;
26 }
```

```
27
28 void init()
29 {
30     long i;
31     for (i = 0; i < 4; i = i + 1) {
32         data[i] = i;
33     }
34
35     // Create 2 adder tasks to calculate sums in parallel.
36     // The used array segments are specified by 'using'.
37     // After the init task exits and commits, the newly created
38     //   adder tasks can be scheduled to execute in parallel.
39     runner adder(0) using data[0,,2];
40     runner adder(1) using data[3,,4];
41
42     // exit and commit
43     commit;
44 }
45
46 // main is a task. It is the first task started for a program.
47 void main()
48 {
49     // Create the init task to initialize data[0] to data[4] and
50     //   create adder tasks.
51     // After the main task exits and commits, the newly created
52     //   init task can be scheduled to execute.
53     runner init()
54         using data[0,,4];
55
56     // exit and commit
57     commit;
58 }
```

The `main` task is created by the MAZE as the starting point of the program. In this program, the `main` task creates the `init` task and the `init` task creates 2 `adder` tasks to calculate sums in parallel. Programmers can compile this program to i0 binary code by "**cc0**

adders.c0 -o adders.bin" using the c0 compiler and run `adders.bin` on a MAZE.

As shown in the example, the grammar of c0 is like C with differences in the grammar and programming model aspects. First, programs are organized as tasks which may execute in parallel and a task commits changes at the end of its execution. Second, new tasks such as `adder` are scheduled to execute only after the parent task such as `init` exits and commits. Third, tasks are created by `runner` statements by specifying data structures like array segment `data[0,,4]` used by the tasks. This programming model may look different at the first glance. However, the model is easy to get familiar with for programmers especially if they have experience with database systems or transactional memory systems which also organize operations on data as transactions. Importantly, this semantics and model enable the system to scale to large clusters and present the programmers a view of a single computer.

1.4 Outline

The thesis is organized as follows. Chapter 2 surveys the related work. We show the limitation of disk-based systems by a probing case in Chapter 3. Chapter 4 presents the design of i0 and MAZE. We discuss the design of VOLUME in Chapter 5. We show the implementation of evaluation of MAZE and VOLUME in Chapter 6. Chapter 7 concludes the thesis. This thesis includes materials and results from our previous published work [67, 68, 69, 70].

CHAPTER 2

RELATED WORK

Many systems, programming frameworks, and languages are proposed and designed to exploit the computing power of constellations of compute servers inside today's gigantic datacenters and help the programmers design parallel programs easily and efficiently.

2.1 MapReduce-like systems

Dean et al. created the MapReduce programming model to process and generate large data sets [42]. MapReduce relies on a restricted programming model so that it can automatically run the programs in parallel and provide fault-tolerance while ensuring correctness. The open-source variant of MapReduce, Hadoop [2], as well as its underlying data persistency layer, HDFS [11], which is loosely modeled after GFS [51], has seen active development and increasing adoption. Without completely hiding the distributed hardware, MapReduce requires programmers to partition the program state so that each map and reduce task can be executed on one individual host, and enforces a specific control flow and dependence relation to ensure correctness and reliability. This leads to a restricted programming model which is efficient for "embarrassingly parallel" programs, where data can be partitioned with minimum dependence and thus the processing is easily parallelizable, but makes it difficult to support sophisticated application logic [47, 67, 80, 89]. Dryad takes a more general approach by allowing an application to specify an arbitrary "communication DAG (directed acyclic graph)" [59].

While these frameworks are successful in large data processing, the restricted programming model such as MapReduce is not general enough to cover many important application domains and makes it difficult to design sophisticated and latency-sensitive applications [47, 67, 80, 89]. Building the DAG for Dryad applications is a non-trivial task to the

programmers and automatic DAG generation is only seen implemented for certain high-level languages.

The limitation of MapReduce is also manifested in problems with large data sets. Chen et al. points out that it is tricky to achieve high performance for programs using MapReduce, although implementing a MapReduce program is easy [36]. 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 [77]. The disk performance is becoming a bottleneck of the MapReduce-like systems, especially for workloads with datasets of moderate sizes.

2.2 High-level languages

As another approach, high-level languages may help programmers write clearer, more compact and more expressive programs on a cluster. High-level languages, such as Sawzall and DryadLINQ implemented on top of programming frameworks (MapReduce and Dryad), make the data-processing programs easier to design [78, 92]. However, these languages also suffer from the limitation of the underlying application frameworks.

Charles et al. design X10 to write parallel programs in non-uniform cluster computing system, taking both performance and productivity as its goals [34]. The language-level approach gives the programmers more precise control of the semantics of parallelization and synchronization. However, the static locality, programmer-specified synchronization, and the linguistic artifacts influence such solutions to perform well with one set of programs but fail to deliver performance, functionality, or ease-of-programming with another.

2.3 Message passing and distributed shared memory

Different from the frameworks and languages that make the data communication transparent to programmers, message passing-based technologies [50] require that the programmer

handle the communication explicitly [66]. This can simplify the system design and improve scalability, but often imposes extra burden on programmers. It creates dichotomized programming domains and requires the programmers to explicitly handle the complexity of organizing the program logic around the APIs and filling any semantic gap. In contrast, the distributed shared memory (DSM) keeps the data transmission among computers transparent to programmers [72]. The DSM systems, such as Ivy [65] and Treadmark [61], combine the advantages of "shared-memory systems" and "distributed-memory systems" to provide a simple programming model by hiding the communication mechanisms [72], but incur a cost in maintaining memory coherence across multiple computers.

2.4 Shared-memory multiprocessor systems and single system image

In contrast to the traditional ways of providing DSM in middleware which provide a limited illusion of shared memory, vNUMA uses virtualization technology to build shared-memory multiprocessor (SMM) system and provides a single system image (SSI) on top of workstations connected through an Ethernet network that provides "causally-ordered delivery" [33]. This also differs from widely used virtualization systems, such as Xen and VMware on the x86 ISA, which divide the resources of a single physical host into multiple virtual machines [25, 88]. Both vNUMA and the emerging "inverse virtualization" [55] aim to merge a number of compute nodes to be one larger machine. However, vNUMA is designed to be used on small clusters and its scalability is limited by the ISA it used which is designed for single computer with up to a moderate number of cores.

2.5 Large-scale in-memory data processing systems

Unifying physical memory from a cluster of servers is studied for many years. Examples include distributed shared memory systems such as Ivy [65] and Treadmarks [61], virtual machines, such as vNUMA [33] and DVM [70], operating system extensions, such as

GMS [49], caching systems such as Memcached [14], and storage systems, such as RAMCloud [77]. Researchers suggest using the physical memory from many servers in the datacenters to create the storage system [77] and some MapReduce implementations, such as Twister [46], Phoenix [80], MRlite [67] and EMR [81], make use of the physical memory on working nodes. Emerging data processing systems, such as Spark/RDD [94] and Piccolo [79], handle the datasets using in-memory data structures and deliver higher performance than file system based systems.

On the other hand, the capacity of the physical memory 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.

CHAPTER 3

LIMITATION OF DISK-BASED SYSTEMS

As discussed in Chapter 1, a major cause of the limitation of many current cloud systems is the way they store, distribute, manipulate and persist data. In this chapter, we study the limitation of the disk-based systems by a probing case on MapReduce and a new parallelization framework and system processing data in memory.

MapReduce [42] is one of the most successful parallelization framework used in datacenters comprising commodity computers [26]. The open-source variant of MapReduce, Hadoop [2], has seen active development activities and adoption. Many cloud computing services provide MapReduce functions [1], and the research community uses MapReduce and Hadoop to solve data-intensive problems in bioinformatics, computational finance, chemistry, and environmental science [37][38][47][82].

On the other hand, the MapReduce model has its discontents. DeWitt et al. argues that MapReduce is much less sophisticated or efficient than parallel database query systems [45]. It is pointed out that the MapReduce model imposes too strong assumptions on the dependence relation among data, and the correctness often depends on the commutativity, associativity, and other properties of the operations [92]. Others point out that the unreliable communication model and retry mechanisms are far from being satisfactory, and the master node can easily become a single point of failure. The performance study on MapReduce-based algorithms exhibits mixed results [38]. Finally, the recently granted MapReduce patent raises question on the long-term viability of using this parallelization mechanism in open environments [43].

We argue, however, that the facts and observations above do not reveal the real limitation of the MapReduce technology–they are either not significant enough to taint the technical merits of MapReduce, or not technical issues at all. In addition to the capability of exploring

massive parallelism, the MapReduce framework has its generality to make it attractive to a wide class of analytics applications. Otherwise, it would not have been used for many years as a fundamental piece of software in the Google architecture, which is a complex system solving many challenging problems [26].

The intrinsic limitation of MapReduce is, in fact, the "one-way scalability" of its design. The design allows a program to scale up to process very large data sets, but constrains a program's ability to process smaller data items. The one-way scalable design reflects assumptions made in a design context where large data sets were the dominating challenges, and affects several important design choices in the MapReduce framework.

While the one-way scalability was a legitimate choice when MapReduce was initially designed, it introduces severe difficulty in extending this programming framework to more general computation. It has become imperative to design a new parallelization framework that is not only scalable but also flexible and generally applicable as cloud computing evolves to cover more dynamic, interactive, and semantic-rich applications, such as multiple-user collaborative applications [7], scientific computing, development tools, and commercial applications [17].

We use a specific case to probe the limitation of MapReduce, and design a new lightweight parallelization framework to mitigate the one-way scalability problem and improve the system performance. The probing case is a distributed compilation tool, and the new parallelization framework is called "MRlite", which can efficiently scale down to process moderate-size data. Our evaluations on MRlite show that it is more than 12 times faster than Hadoop in the distributed compiling workload.

3.1 A case study

To probe the limitation of the MapReduce framework, we design mrcc [15], a distributed compilation system, and examine its performance and overhead. MapReduce is not designed for the compilation workload which contains moderate-size data with complex dependency.

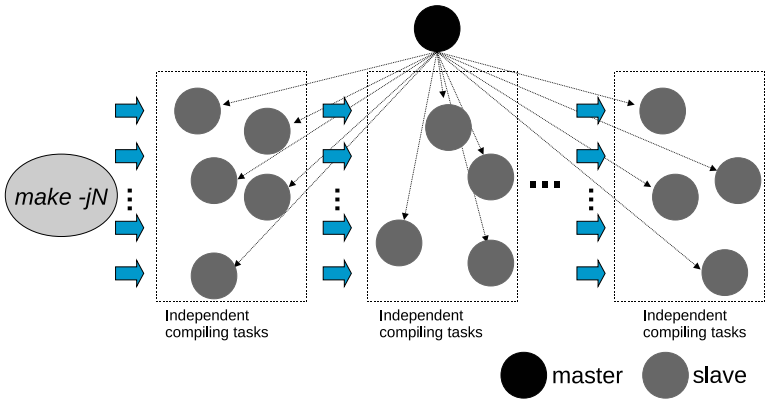


Figure 3.1: The architecture of mrcc

We choose the compilation workload to probe the limitation of MapReduce. Meanwhile, a large class of applications share the features of compilation workload, such as variable-size data and dependency among them, and variable degree of parallelization at different algorithmic steps.

mrcc consists of one master node that controls the compilation job and many slave nodes that handle the compilation tasks as shown in Figure 3.1.

When one project is compiled on the master node, *make* builds the dependency tree for this project, and invokes multiple mrcc program instances to compile multiple source files in parallel. Hence, the parallelization are leveraged by *make* invoking multiple concurrent mrcc instances. Each mrcc instance runs one compilation task on a slave node. A slave node is one of the worker machines that receive map/reduce tasks from MapReduce master.

When conducting remote compilation on a slave node, mrcc preprocesses the source file, places a batch of preprocessed source files into a network file system used by the framework, then starts a compilation job on MapReduce. The map operation of this MapReduce job is done by a program called “mrcc-map”. Running on the slave node, mrcc-map first retrieves the source file from the network file system, then calls the compiler locally to process the source file on the slave. After the compilation finishes, mrcc-map places the object file which is the result of the compilation back into the network file system. After the mrcc-map task is finished, mrcc on the master node retrieves the object file from the network file system and

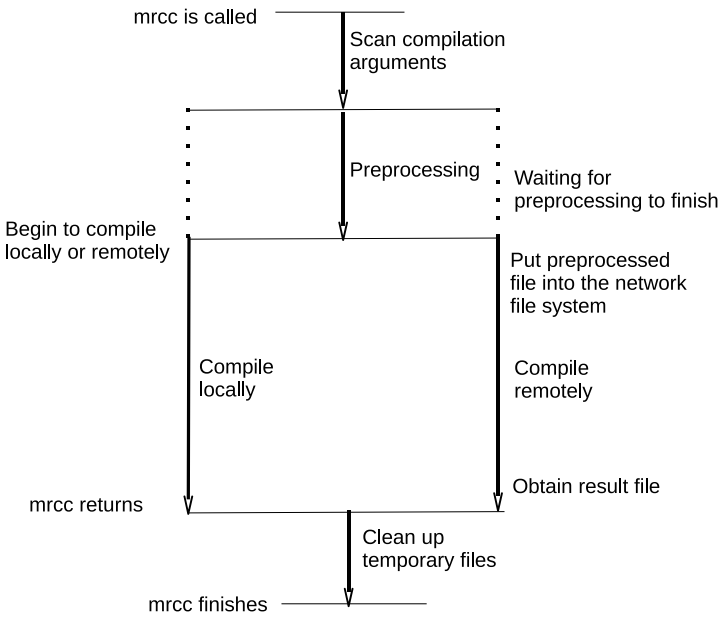


Figure 3.2: The work flow of mrcc

places it into the master node's local file system. After one batch of files are compiled, *make* continues to release more files to be compiled that depend on the completed ones.

Implementation The mrcc compilation system consists of two core parts: the main program mrcc which runs on the master node and mrcc-map which runs on the slave nodes. mrcc is an open-source project under the GNU General Public License, version 2. The source files of mrcc can be downloaded from [15]. The work flow of the mrcc program is shown in Figure 3.2.

mrcc forks the preprocessor process after scanning the compiler arguments. The preprocessor inserts the header file(s) into the source file so that the remote nodes can assume a much simpler execution environment. mrcc then places the preprocessed file into a network file system and conducts remote compilation. When running mrcc on Hadoop, we use Hadoop Streaming [9] which can run MapReduce jobs with any executable or script to perform the map or reduce operation. mrcc submits the job to Hadoop and mrcc-map on the slave node is invoked by Hadoop to perform the map operation.

Table 3.1: Node configuration

	CPU	Memory (GB)	Number
mrcc master	4	2	1
Slaves	2	2	10
Hadoop or MRlite master	2	2	1
NFS server	2	14	1

mrcc-map is implemented as a program residing in local directories of all the slave nodes because the mrcc-map program does not change during the process of compiling one project. This also makes it "easier" for the MapReduce framework to handle the compilation tasks, and, hence, the performance penalty we observe shall reflect more accurately the intrinsic limitations of the methodology.

mrcc-map first parses its arguments to obtain the source file name on the network file system and the compilation arguments. mrcc-map then retrieve the preprocessed file from the network file system. After that, mrcc-map calls the local *gcc* compiler and passes the compilation arguments to it. When *gcc* exits with a successful return value, mrcc-map places the object file into the network file system and returns immediately.

Upon the completion of all mrcc-map tasks, the Hadoop Streaming job returns. mrcc obtains the object files from the network file system and returns.

Performance of mrcc on Hadoop We set up an experiment platform that consists of Xen virtual machines. We isolate these virtual machines by assigning each virtual machine except the mrcc master a physical CPU core which contains two CPUs. The configuration of the slave nodes are identical. Details of the node configuration are listed in Table 3.1. The Hadoop master node and three slave nodes reside on one physical machine while the other seven slave nodes is on top of another physical machine. The mrcc master node is on top of the third physical machine. The physical machines are connected by a Netgear JG5516 1Gbps switch. The bandwidth between two virtual machines on top of one physical machine is also 1Gbps.

Table 3.2: Time for compiling projects using gcc on one node and mrcc on Hadoop

Project	Time for gcc	Time for mrcc/Hadoop
Linux	48m56.2s	150m44.4s
ImageMagick	5m12.1s	10m52.7s
Xen tools	2m7.6s	23m38.9s

We use mrcc to compile the Linux kernel 2.6.31.6, ImageMagick 6.6.3-8, and Xen tools 4.0.0 on Hadoop to examine the performance of Hadoop when it processes jobs in which complex dependencies exist between tasks while the input data files are also dynamically generated. The Hadoop version is 0.20.2 [2].

Table 3.2 shows the performance data. The compilation time using mrcc on 10 nodes is at least twice as long as that on one node (sequential compilation). Further investigation reveals that Hadoop takes more than thirty seconds to complete one compilation task. We also measures that Hadoop takes more than 20 seconds to finish one "null" job even though the job does not do any work. Storing or retrieving one file on the network file system takes at least 2 seconds while compiling one file on one node usually takes less than 2 seconds. While such overheads are acceptable in the special class of applications where relatively simple processing logic is applied to a large number of independent data, the prohibitive tasking and data transportation cost limits the applicability of MapReduce/Hadoop in more general workloads.

3.2 MapReduce on a memory-based file system

The experimental results in Section 3.1 show that the current design and implementation of the MapReduce framework cannot provide the flexibility and efficiency required by programs with numerous parallelizable steps, instead of one massive parallelizable step. Hence, the current MapReduce framework does not work effectively for a large class of applications with not only sizable data but also non-trivial application logic. Representing the "common" case in scientific and business computing, such applications require the programming frame-

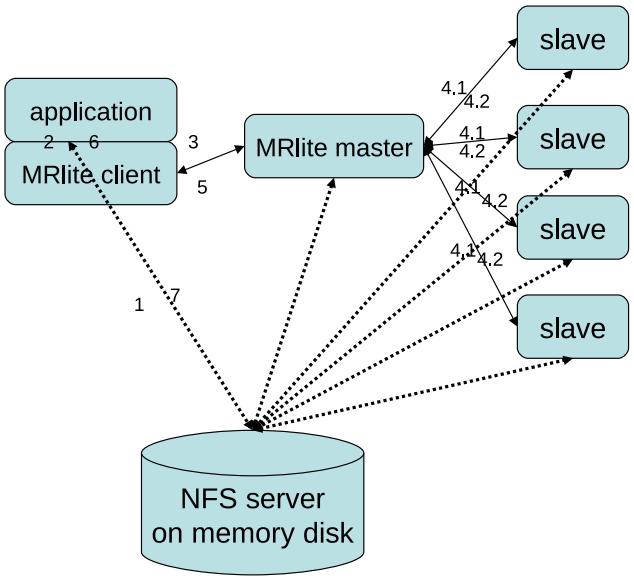


Figure 3.3: The architecture of MRlite

work to efficiently handle variable-size data, support data dependence, and harness variable degrees of parallelization with controlled latency.

To overcome the limitation of MapReduce, we have designed and implemented MRlite, a lightweight parallelization framework that optimizes for not only massive parallelism, but also low latency to provide a more general and flexible parallel execution capability in cloud computing environments. The data in a complex computing system are often dynamically generated, thus introducing dependence among data. In fact, "data with dependence" is the common case in general applications. Such dependence naturally divides the processing into numerous steps to be taken in order, but each step may have sufficient parallelism to be exploited. The key is, consequently, to significantly reduce the overhead in data transportation and task management so that most of the parallelizable steps can invoke the parallelization mechanism, with the performance gain from parallel execution outweighing the latency induced by the overheads.

Architecture The MRlite system consists of four parts as shown in Figure 3.3: the MRlite master, MRlite slaves, the NFS server in memory, and the MRlite client.

The MRlite master controls the parallel execution of tasks. It accepts jobs from the MRlite client, and distributes the tasks of the jobs to MRlite slaves. The MRlite slaves accept and execute the tasks from the MRlite master. Implemented as a library that can be linked to the user application, the MRlite client accepts the application's parallelization requests, and submits the job to the MRlite master. MRlite includes an NFS server whose files are stored in one participating node's memory to provide a file system abstraction. The NFS file system is mounted on the MRlite master node, all the MRlite slave nodes, and the node on top of which the user application runs.

Latency optimization The MRlite master cooperates with the MRlite client to minimize overhead and perform low-latency operations. In addition, the MRlite framework includes a timing control feature in its design as part of the low-latency execution mechanism. The application can estimate a timeout limit for the job and provide the timeout limit when it sends parallelization request for one job to the MRlite client. According to the timeout limit for the whole job, the MRlite master provides a suggested timeout value for the tasks to be executed on the slaves. In the current design, the timeout value is specified by the programmer. After receiving the task command from the master, the slave tries its best to complete the task during the time slot specified by the timeout value. The timing enforcement also take care of reliability through retries. If one task times out, the master treats it as a failed one and may retry that task on another slave or just report the failure to the master. Similarly, a timed-out job may be treated as a failed one by the MRlite client, and the client may retry the job for a certain number of times or report the failure to the application according to that job's configuration. The application logic ultimately decides how to proceed when a job fails.

Besides the execution time enforcement, the MRlite master submits tasks to slaves without sophisticated queueing to maximize the possibility of finishing the job within the timeout limit. As there are dependences between jobs and among map and reduce tasks, the master

submits the tasks as soon as the dependence is resolved.

The latency caused by the run-time overhead in each step may become critical when processing moderate data sets though it may not be a concern for processing a huge amount of data. Unlike the Hadoop design, MRlite uses a run-time daemon and thread pools to support the operations of the master and the slaves. This design reduces the cost of creating a process every time a job request or task request is issued. As the multi-thread and multi-core technology is widely available in modern computing platforms, we believe it is a pleasantly acceptable cost to dedicate one thread for each task on a slave and one thread for managing a job on the master.

Data transportation is an important aspect in distributed computing. In our lightweight parallelization framework, it is convenient to provide a distributed file system to store data, but we only store intermediate data files and the run-time data in the network file system. The design choice on the network file system implementation must balance the performance and usability. MRlite includes an NFS server, which runs on one participating node, to provide a file system abstraction, and mount the NFS file system on the MRlite master, all MRlite slaves, and the MRlite clients. The NFS server rides on a *tmpfs* file system [84], a virtual memory file system in Linux, so that the I/O speed of the NFS directory is as fast as operations in memory. This design choice reflects the observation that modern gigabit and 10 gigabit NICs provide comparable throughput between networked computers to the I/O bandwidth between memory and hard drives. To maximize the I/O speed of the network file system, we do not duplicate the intermediate files as some distributed file systems do [51].

In the current design of MRlite, data persistency is supposed to be provided by a separate layer of data storage. The software based reliability through multi-way replication is not included in MRlite since these are not the focus of this work, and solutions that provide these features already exist [11][51]. Replication can potentially increase the serving bandwidth of read operations, at the cost of first increasing the cost of writing operations. Both GFS and HDFS employs 3-way replication [11][51] to improve concurrency and reliability. In our experiments, it has not been observed that the lack of 3-way serving bandwidth limits

the parallelization capability or the overall application-level performance when the network bandwidth is sufficient. Nevertheless, the MRlite architecture does not prohibit the addition of a replicated data storage, given that intermediate data files are still stored on and served from the low-latency network file system.

3.3 Prototype and evaluation

We have prototyped the MRlite parallelization service, and implemented mrcc on MRlite. In this section, we discuss the implementation details of MRlite, and report the evaluation results of mrcc on MRlite.

Implementation The MRlite jobs are represented as sets of native Linux applications written in any programming language of choice. After each job is split into numerous tasks, each task is executed as a Linux program by one of the MRlite slaves.

At each parallelizable step, the MRlite framework dispatches a group of concurrent tasks, emulating the MapReduce model inspired by list primitives in Lisp. The tasks executed on MRlite slaves are defined as map and reduce tasks, with reduce tasks aggregating the intermediate results generated by the map tasks. It is worth noting that we do not restrict map and reduce tasks to use key/value pairs. The MRlite slaves monitor the tasks' execution and report the execution's status and return values to the MRlite master.

There are 7 steps during the process of executing one job as shown in Figure 3.3:

1. The application places input data files to the input NFS directory.
2. The application submits the MapReduce job to the MRlite client.
3. MRlite client accepts the job, and submits it to the MRlite master.
4. The MRlite master submits tasks of the job to slaves (4.1), and waits for slaves to respond (4.2). Perform steps 4.1 and 4.2 for all tasks.

Table 3.3: Comparison of speedups of mrcc on Hadoop and MRlite

	Speedup on Hadoop	**Speedup on MRlite**	**MRlite vs. Hadoop**
Linux	0.32	5.8	17.9
ImageMagick	0.48	6.2	13.0
Xen tools	0.09	2.0	22.0

5. The MRlite master sends a success or fail message back to MRlite client.

6. MRlite client returns to the application with a return value that represents the result.

7. The application retrieves the output data files from the output NFS directory.

Evaluation Because MRlite is a general parallelization framework, we can port the mrcc program to MRlite, and compare the performance with mrcc on Hadoop using the workloads as described in Section 3.1 with complex dependencies existing among tasks and some input data files dynamically generated. This evaluation examine MRlite's ability to scale "down" to handle moderate-size data, mixed sequential and parallel work flow, and latency-aware tasks.

Most of the components in the mrcc/MRlite implementation are identical to those in the Hadoop based implementation except the parts invoking programming interface. The configurations of the nodes and the client nodes are listed in Table 3.1. The MRlite platform has the same number and hardware configuration of slave nodes as Hadoop in Section 3.1.

Figure 3.4 shows the evaluation results. The compilation of the three projects using mrcc on MRlite is much faster than compilation on one node, with a speedup of at least 2 and the best speedup reaches 6. Table 3.3 lists the speedup mrcc achieves on Hadoop and MRlite, and shows that the average speedup of MRlite is more than 12 times better than that of Hadoop. This comparison shows that the MRlite is more effective than MapReduce for workloads with numerous computational steps where each step may have parallelization opportunities. From the result we also find that a project that is easier (a higher speedup) for mrcc on Hadoop to compile is easier for mrcc on MRlite. When compiling the "easier"

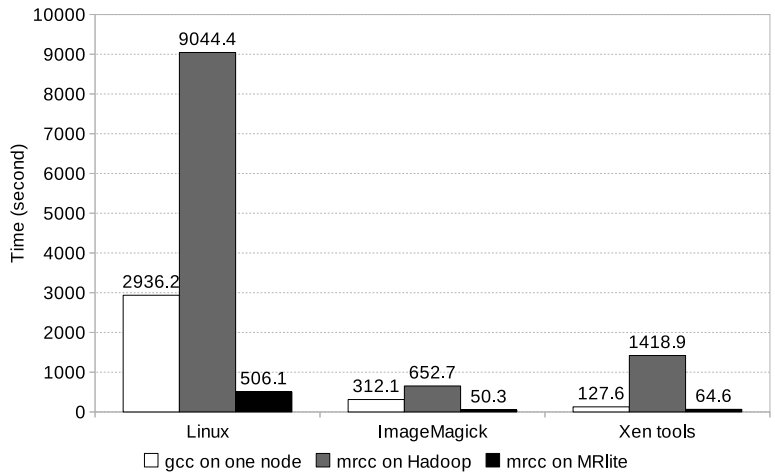


Figure 3.4: Execution time for compiling projects

project such as ImageMagick, mrcc on MRlite can achieve better speedup than mrcc on Hadoop.

The evaluation shows that MRlite is one order of magnitude faster than Hadoop on problems that MapReduce has difficulty in handling. The MRlite framework can still handle the massive parallelism with simple dependency as MapReduce does. MRlite shows that we can implement a flexible and efficient parallelization framework which programs can easily invoke to handle both large and small data sets.

3.4 Discussion

We use the distributed compilation case to probe the limitation of MapReduce. The probing case shows that the overhead of Hadoop is too high for mrcc and the performance of mrcc on Hadoop is far from satisfactory.

We design MRlite, a new lightweight parallelization framework, to mitigate the one-way scalability problem and improve the system performance. The evaluation result shows that MRlite can efficiently process moderate-size data and handle data dependence. The MRlite framework can still handle the massive parallelism with simple dependency. The design significantly improves the parallel execution performance for general applications.

On the other hand, MRlite has its limitations and can not support general large-scale data processing in datacenters. First, the datasets that MRlite can handle is limited by the capacity of the physical memory of the NFS server node. Second, the memory-based NFS server does not provide persistency. Third, the programming model of MRlite is not general enough for sophisticated programs and imposes burden to programmers to resolve dependence among phases of tasks.

In the following sections, we will discuss the design and implementation of Layer Zero which efficiently handles data in-memory as MRlite does, and also provides a simple yet general and scalable programming model and scalable data substrate with atomicity and persistency support.

CHAPTER 4

ORGANIZATION OF IN-MEMORY COMPUTATION

In Chapter 3, we study the limitation of disk-based systems and show that, while significantly improving the performance of computation, in-memory data processing imposes many challenges. To provide a general substrate to support datacenter-scale computation, we design the Layer Zero components including i0, MAZE and VOLUME. In this chapter, we discuss the organization of in-memory computation on i0 and MAZE. We will present the design of VOLUME in Chapter 5.

As Layer Zero is an approach to supporting general computation in datacenters and MAZE is the fundamental system of the Layer Zero technologies, the design of MAZE has the following goals.

- **General**. MAZE should support the design and execution of general programs in datacenters.
- **Efficient**. MAZE should be efficient and deliver high performance.
- **Scalable**. MAZE should be scalable to run on a plurality of hosts, execute a large number of tasks, and handle large datasets.
- **Portable**. MAZE should be portable across different clusters with different hardware configurations, networks, etc.
- **Simple**. It should be easy to program on MAZE; i0 should be simple to implement.

With the design goals stated, we revisit the approach and rationale of unifying the computers in a datacenter at the ISA layer. The programming framework, language, and system

call/API are also possible abstraction levels for large-scale computation. However, as aforementioned, the frameworks and languages have their limitations and constraints. They fall short of providing the required *generality* and *efficiency*. Using system calls/APIs to invoke distributed system services can be *efficient* and reasonably *portable*. However, an abstraction on this level creates dichotomized programming domains and requires the programmers to explicitly handle the complexity of organizing the program logic around the system calls/APIs and filling any semantic gap, which fails to provide the illusion of a "single big machine" to the programmers and leads to a far less *general* and *easy-to-program* approach than what we aim to design. The existing ISAs, designed for a single computer with up to a moderate number of cores, can hardly provide the *scalability* required by the "machine" that scales potentially to the datacenter size. On the other hand, the design of the Layer Zero is motivated by the necessity of developing a next-generation datacenter computing technology that overcomes several important limitations in current solutions. The MAZE should provide the basic program execution and parallelization substrate in this technology, and should meet all the goals on *generality*, *efficiency*, *scalability*, *portability* and *ease-of-programming* (with compilers' help). This leads us to choose the ISA layer to construct a unified computational abstraction of the datacenter platform. To break the *scalability* bottleneck in traditional ISAs and retain the advantages of *generality* and *efficiency*, we design the new ISA, i0, and build the MAZE on this architecture.

4.1 System overview

The architecture of MAZE is illustrated in Figure 4.1. Many MAZEs can run in one physical host as the traditional VMMs to multiplex the resources of the host, and a MAZE can run on top of multiple or many physical hosts to form a big virtual machine in a datacenter.

System organization We abstract a group of computational resources including processor cores and memory to be a *resource container* (RC). In the simplest form, an RC is a physical computer. However, it can also materialize as a traditional virtual machine or other container

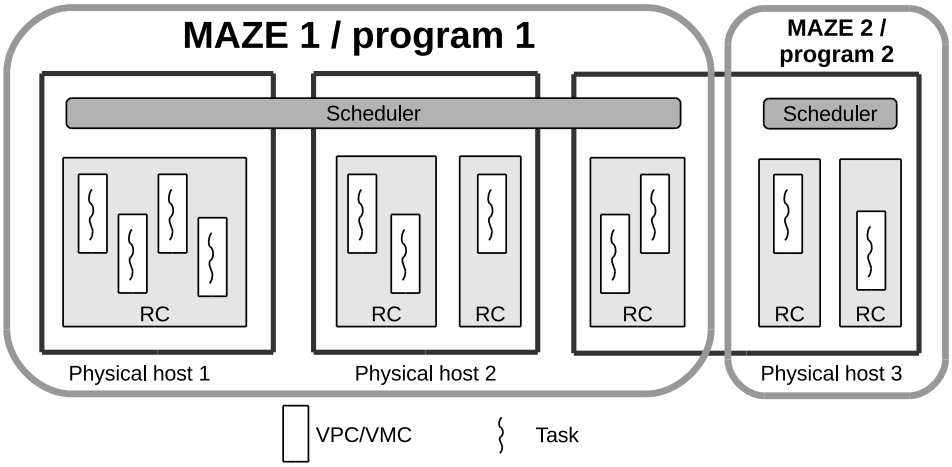


Figure 4.1: Organization of 2 MAZEs on 3 physical hosts

structure, given the container provides an appropriate programming interface and adequate isolation level. Although one RC is exclusively assigned to one MAZE during one period of time, the RCs in one physical host may belong to one or more MAZEs and one MAZE's capacity can be dynamically changed by adding or removing RCs. Hence, the capacity of a MAZE is flexible and scalable—a MAZE can share a single physical host with other MAZEs, or include many physical hosts as its RCs.

A MAZE accommodates a set of tasks running in a unified address space. Although MAZEs may have different capacity and the size of a MAZE can grow very large depending on the available resources, the MAZEs provide the same architectural abstraction of a "single computer" to programs through the i0.

The programs in MAZEs are in i0 instructions and a program running inside a MAZE instantiates to be a number of *tasks* executing on many processor cores in many RCs. Inside an RC, many tasks, each residing in one *virtual processor container* (VPC), can execute in parallel. The VPC is a semi-persistent facility that provides various system functions to tasks and a meta-scheduler (scheduler) to facilitate and manage tasks' execution and the scheduler is a distributed service running on all constituent hosts of a MAZE, with one scheduler master providing coordination and serialization functions. Specifically, the VPC a) accepts the meta-scheduler's commands to prepare and extend tasks' program state and start tasks,

b) helps the memory subsystem handle tasks' memory accesses, c) facilitates the execution of instructions such as `spawn` (see Table 4.1), d) sends scheduling-related requests to the meta-scheduler, and e) provides I/O services. As these functions are tightly correlated with other subsystems of MAZE, we introduce pertinent details of the VPC design along with the treatise of the other system components in Section 4.2 and Section 4.3. Suffice it at present to know that one VPC contains at most one task at a specific time, and it becomes available for accommodating another task after the current incumbent task exits.

Programming in i0 We use one example to illustrate how to write a simple program on a MAZE using i0. Details of the i0 architecture will be introduced in Section 4.2.

The following i0 code performs a sum operation of two 64-bit integers by a *sum_task* task. The starting address of the memory range storing the two integers is in `0x300001000`, and `0x300001008` contains the address where the results should be stored. The text after ";" in a line is a comment, and the task's name is in italic.

```
1  sum_task:
2  add (0x300001000):ue, 8(0x300001000), (0x300001008)
3                                          ; :ue indicates 64-b data
4  conv $1:ue, 8(0x300001008):ue           ; set exit code
5  exit:c                                  ; exit and commit
```

The code above adds two 64-bit integers with all operands stored in memory. Figure 4.2 shows the execution of the `sum` operation. Suppose the memory location `0x300001000` and `0x300001008` store values `0x800000000` and `0x900000000`, respectively. The `add` instruction at line 2 obtains the operands at `0x800000000` and `0x800000008`, computes the sum, and stores the sum in `0x900000000`. The `conv` instruction at line 4 sets the exit code for the task at address `8(0x300001008)` (`0x900000008` after dereferencing). Finally, the `exit:c` instruction at line 5 makes the task exit and commit its changes to the memory. After being committed, the changes become visible to other tasks.

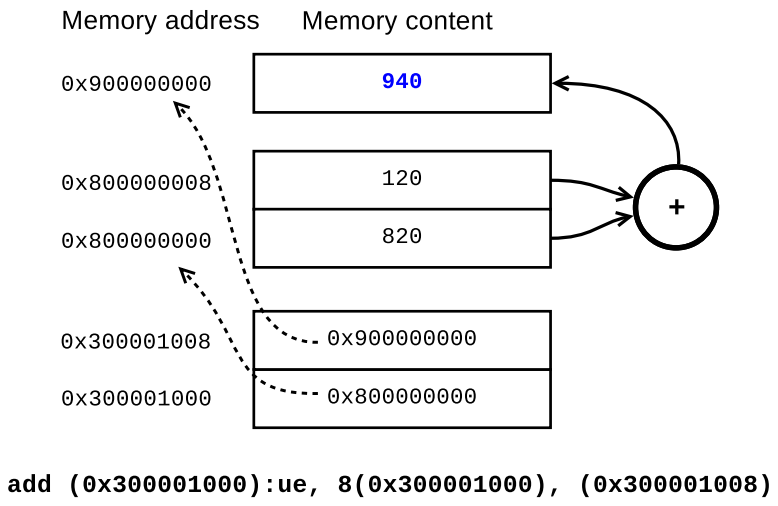


Figure 4.2: Execution of an `add` instruction

Table 4.1: i0 instructions

Instr.	Operands	Effect
conv	D_1, M_1	Convert D_1 to M_1
add	D_1, D_2, M_1	Add D_1 and D_2; store the result in M_1
sub	D_1, D_2, M_1	Subtract D_2 from D_1; store the result in M_1
mul	D_1, D_2, M_1	Multiply D_1 and D_2; store the result in M_1
div	D_1, D_2, M_1	Divide D_1 by D_2; store the result in M_1
and	D_1, D_2, M_1	Store the bitwise AND of D_1 and D_2 in M_1
or	D_1, D_2, M_1	Store the bitwise OR of D_1 and D_2 in M_1
xor	D_1, D_2, M_1	Store the bitwise exclusive OR of D_1 and D_2 in M_1
shift	D_1, I_1, M_1	Store the result of shifting D_1 by I_1 bits in M_1
b	D_1, D_2, M_1	Compare D_1 and D_2; branch to M_1 if a specific condition is met
brl	M_1, M_2	Branch and link (procedure call)
spawn	M_1, M_2, M_3, M_4	Create a new task
spawnx	M_1, M_2, M_3, M_4, M_5	Create a new task in a new space (discussed in Chapter 5)
exit		Exit and commit or abort
nop		No operation
int	I	Soft interrupt

M_n specifies an operand in memory; I_n specifies an immediate operand; D_n specifies an operand in memory or an immediate operand.

The example illustrates several important design choices of i0. First, it uses memory addressing, not combined register-memory addressing, for all operands. With an emphasis on instruction orthogonality, i0 allows an instruction to freely choose memory addressing modes to reference the operands. Second, i0 provides a low-level abstraction which is close to hardware and the semantics of most of the instructions are common in other widely used ISAs. With this design, i0 retains traditional ISAs' advantages, such as generality and efficiency, and it is easy to map i0 to hardware or another common ISA at low cost. Finally, the commit operation, which makes a task's changes to the memory visible to other tasks, occurs at the end of a task's execution. This arrangement enforces a clearly defined task lifecycle and its semantics concerning the program state of the MAZE.

4.2 i0

We design i0 as a new ISA targeting a large-scale virtual machine running on a plurality of networked hosts and build the MAZE above this architectural abstraction. We first introduce the design of i0's instructions in this section, and then present i0's parallelization mechanism in Section 4.3. We will discuss the memory model in Section 5.2.

Given the system goals of the MAZE design, the goals of the i0 design are as follows.

- i0 should support a memory model and parallelization mechanism scalable to a large number of hosts.
- i0 programs should efficiently execute on common computing hardware used in datacenters, which are usually made of commodity components. Certain hardware may provide native support to i0 in the future, although we emulate i0 instructions on the x86-64 architecture in our current implementation.
- i0 instructions should be able to express general application logic efficiently.
- i0 should be a simple instruction set with a small number of instructions so that it is easy to implement and port.

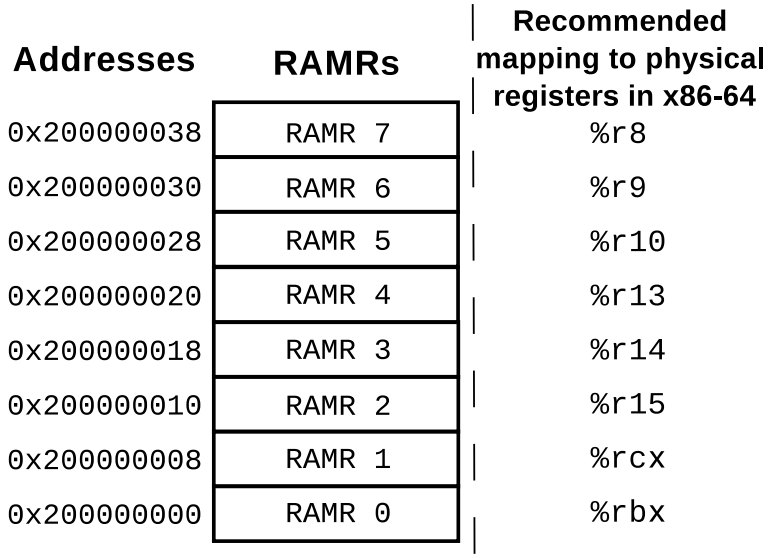


Figure 4.3: Mapping of the RAMRs to physical registers in x86-64

- i0 should be Turing-complete.

In summary, i0 should be scalable, efficient, general, simple, and Turing-complete. To ensure programmability, simplicity and efficiency, i0, as shown in Table 4.1, includes a selected group of frequently used instructions. In addition, the `spawn` and `exit` instructions facilitate construction of concurrent programs with many execution flows. The Turing completeness can be proved by using i0 to emulate a Turing-complete `subleq` machine [70].

An i0 instruction may use options to indicate specific operation semantics, such as the option `:c` in the *sum_task* example in Section 4.1, and the operands can reference immediate values or memory contents using direct, indirect or displacement addressing modes. Although not explicitly providing registers, the unified operand representation permits register-based optimization because the memory model allows some memory ranges in DR (*Direct-addressing Region*, discussed in Section 5.2), which can only be referenced with direct addressing, to be affiliated with registers. We name these ranges *Register-Affiliated Memory Ranges* or RAMR for short.

Figure 4.3 shows the RAMR to physical register mapping in our implementation on x86-64. i0 currently supports 8 registers named RAMR 0 to 7. The programmers or compilers

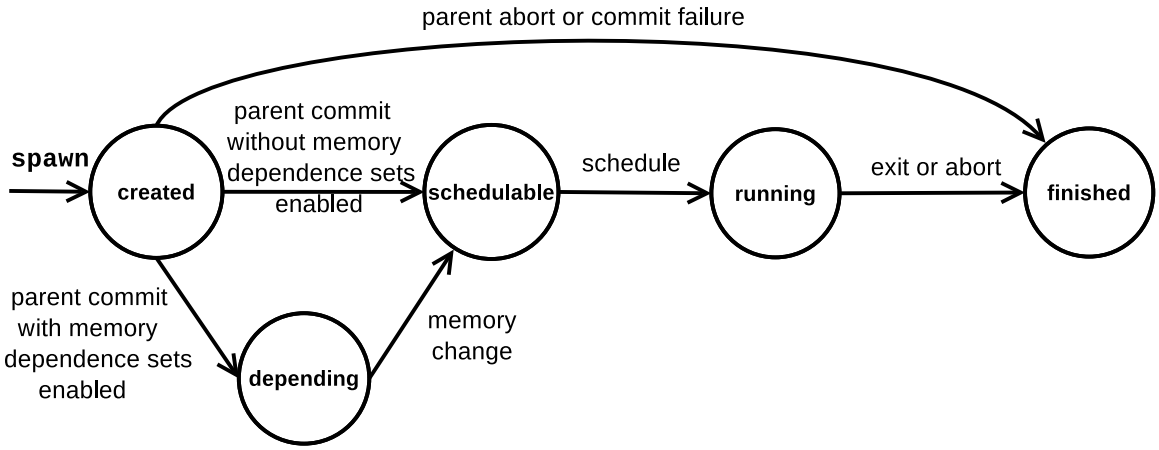


Figure 4.4: State transition of the task and depending task

can use these RAMRs to store most frequently accessed data in the program to improve the performance. In our evaluation discussed in Chapter 6, we show that using RAMRs can speed up the *loop* microbenchmark by more than 5 times.

At first glance the RAMR design seems to make i0 less portable. However, the programs using RAMRs need not to be changed for MAZEs on different platforms. On a platform that has fewer physical registers in the processors than the RAMRs, the RAMRs simply revert to direct-addressed memory ranges. Hence, a MAZE can speed up the programs with physical registers as much as possible while ensuring the portability and correctness.

4.3 Parallelization and scheduling

We design MAZE's parallelization mechanism so that it is easier to develop not only "embarrassingly parallel" programs, but also more sophisticated applications, and the programs can efficiently execute in a datacenter environment. The goal is to provide a scalable, concurrent and easy-to-program task execution mechanism with automatic dependence resolution. To achieve the goal, we design a many-task parallelization mechanism for large-scale parallel computation, a scheduler to manage the tasks, and depending tasks to express and resolve task dependences.

4.3.1 Task

Tasks are the abstraction of the program flows in a MAZE. A task is created by its parent task and terminates after it exits or aborts. Figure 4.4 summarizes the state transitions in a task's life cycle. The parent task creates a new task by executing the `spawn` instruction. A "created" task moves to the "schedulable" state after the parent task exits and commits successfully. The task moves to the "running" state after being scheduled and to the "finished" state after it exits or aborts. The "depending" state applies only to a class of special tasks called "depending tasks" which are introduced in Section 4.3.4.

Each task uses a range of memory as its stack range (stack for short), changes to which are discarded after the task exits, and multiple ranges of memory as its heap ranges. Because tasks are started by the scheduler, the scheduler needs to know certain control information about the task, such as the location of code and the task's stack and heap ranges. MAZE uses a Task Control Block (TCB) to store such control information. Figure 4.5 shows the content of a task's TCB as well as the task's initial stack. The TCB contains the stack base address sb and stack size ss. fi specifies the address of the first instruction from which the task starts to execute. The $heap\_ranges$ and $heap\_range\_count$ fields stores the locations and the number of the heap ranges, respectively. The MDS_ranges and MDS_range_count fields are used by the depending task mechanism discussed later.

Every task resides in one VPC. Providing system functions related to the management and operations of tasks, the VPC reduces the overhead of dispatching and starting a task. VPCs interact with other parts of the MAZE system, and hide the complexity of the system so that the tasks can focus on expressing the application logic on the abstraction provided by i0. Because VPCs are reused, a large portion of the task startup overhead, such as loading modules and setting up memory mapping, is incurred only once in the lifetime of a MAZE. To start a task, the VPC only needs to notify the memory subsystem to set up the task's snapshot, and the supporting mechanisms, such as the network connections, can be reused. Hence, VPCs make it efficient to start tasks in a MAZE. Chapter 6 quantitatively studies the overhead of creating and starting tasks.

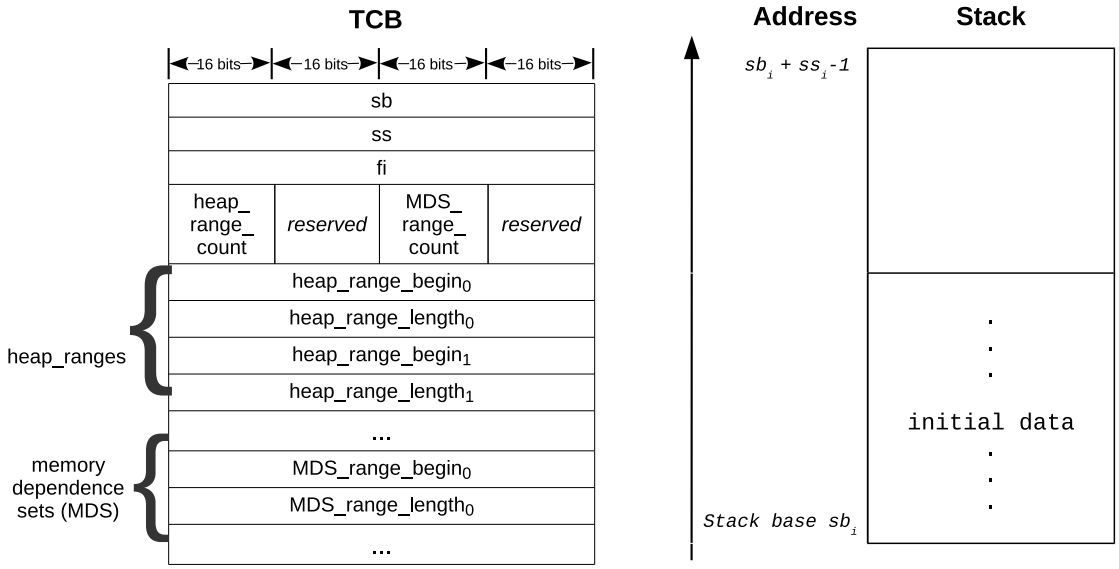


Figure 4.5: Task i's initial stack and TCB

4.3.2 Scheduling

In a complex program, myriads of tasks may be created dynamically and exit the system after completing their computation. VPCs, on the other hand, represent a semi-persistent facility to support tasks' execution. Hence, there can be more tasks than VPCs in a MAZE, and the scheduler assigns tasks to VPCs, as well as manages and coordinates the tasks throughout their life cycles. Specifically, the scheduler is responsible for deciding when tasks should start execution, assigning tasks to VPCs, preparing tasks' memory snapshots, handling tasks' requests to create new tasks and managing the tasks that the MAZE does not have resources (VPCs) to execute currently. As the child tasks created by a parent task are not schedulable until the parent task exits and commits successfully, the scheduler also maintains the "premature" tasks (the tasks in the "created" state) in the MAZE. In addition, the scheduler implements the MAZE's depending task mechanism (to be discussed in Section 4.3.4).

The scheduler of MAZE is a distributed service consisting of two parts: the scheduler master (the master) and the scheduler agent (the agent). There is a single master in a MAZE, and there are many agents residing in VPCs. The master and the agents communicate with

each other through the datacenter network. The master makes high-level scheduling decisions, such as dispatching tasks for execution and assigning them to VPCs, and commands the agents to handle the remaining task management work, such as preparing and updating tasks' snapshots. In the other direction, an VPC can send scheduling-related requests to its associated agent and the agent may ask the master for coordination if it is needed.

The scheduler dispatches multiple tasks from its pool of schedulable tasks according to the scheduling policy, which takes locality along with other factors into consideration, to an VPC when it is free (no task is running in it), creating a memory snapshot for the task and notifying the VPC to execute the task. A task may create as many child tasks as the entire MAZE (or datacenter) can handle. When a task requests to create new tasks, the scheduler constructs TCBs for the child tasks, and extends the parent task's memory snapshot so that the parent can construct and access the child tasks' stacks. After the parent exits and commits successfully, the scheduler adds the child tasks to the task pool and marks them as schedulable.

Although there is only one scheduler master in a MAZE, it has not been found that the single master constrains the MAZE's scalability. As aforementioned, the master is only responsible for high-level decisions and occasional coordination and arbitration, and is, consequently, very lightweight. Most of the work is handled in parallel by the scheduler agents in the VPCs. Hence, the scheduler does not constitute a bottleneck in the MAZE in practice. Several large-scale systems, such as MapReduce [42], GFS [51], and Dryad [59], follow a similar single-master design pattern.

Together with the design of the scheduler, the design of other components of MAZE optimize the workflow to ensure scalability. The snapshot-based memory model of i0 enables massive task-level parallelism by invoking scheduling and coordination only when a task starts and commits. The ISA-level lightweight task creation through the `spawn` instruction enables programs to create tasks at low cost. The two-version design makes the metadata management fast and efficient so that the memory subsystem can scale to handle many snapshots. These designs ensure that MAZE scales well as the processor count and the data size

increase. In Chapter 6, we examine MAZE's scalability.

4.3.3 Many-task parallel execution

The MAZE should be able to accommodate a large number of tasks and execute them in parallel to exploit the aggregate computing capacity of many physical hosts. This requires that the parallelization mechanism of MAZE provide scalable, concurrent and efficient task execution and a flexible programming model to support many-task creation and execution.

To support many-task parallel execution on the ISA level, the task creation and dispatching mechanism must be highly efficient. We design an instruction-level task creation mechanism. To create a new task and prepare it for execution, the program only needs to specify the fi and the heap ranges for the new task, issue a "`spawn`" instruction, and instantiate the stack.

We use an example to show how a task creates a new task and how the MAZE handles the task creation. When a task, T_i, creates a new task, T_j, T_i executes the instruction "`spawn`" with addresses of T_j's stack range, heap ranges and fi, as operands. $VPC(T_i)$, the VPC in which T_i runs, sends T_j's control information specified by these operands to the scheduler. As T_i may write the input data into T_j's stack and a memory range can be used by T_i only after $VPC(T_i)$ has the range's snapshot, the scheduler creates a snapshot of T_j's stack and merges the snapshot into T_i's current one. After the memory range for T_j's stack is ready, T_i writes the initial application data into T_j's stack. The scheduler constructs and records T_j's TCB along with the TCBs of the other tasks created by T_i.

When T_i exits and commits, the scheduler starts the newly created tasks by T_i, including T_j, according to the recorded TCBs. Using T_j as an example, the task start-up procedure is as follows.

1. The scheduler chooses $VPC(T_j)$ for T_j.
2. The scheduler creates snapshot A of T_j's stack range and heap ranges according to TCB_j.

3. The scheduler sends A and TCB_j to $VPC(T_j)$ and commands $VPC(T_j)$ to start T_j.

4. $VPC(T_j)$ sets T_j's local context according to TCB_j.

5. $VPC(T_j)$ starts to execute T_j from fi.

4.3.4 Task dependency

Task dependency control is a key issue in concurrent program execution. Related to this, synchronization is often used to ensure the correct order of the operations, avoid race conditions, and, with execution order constrained, guarantee that the concurrent execution does not violate dependence relations. The existing approaches have their limitations and constraints: the synchronization mechanisms are not natural or efficient to represent dependence [35, 87]; MapReduce employs a restricted programming model and makes it difficult to express sophisticated application logic [47, 80, 89]; DAG-based frameworks, such as Dryad [59], incur non-trivial burden in programming, and automatic DAG generation has only been implemented for certain high-level languages [92].

As an important goal in the MAZE design, the i0 architecture should provide a task dependency representation and resolution mechanism with which task-level dependence can be naturally expressed, concurrent tasks can proceed without using synchronization mechanisms such as locks [29], and sophisticated computation logic can be processed effectively with dynamically scheduled parallelism. Departing from existing solutions, we design a depending task mechanism in the i0 architecture to provide a flexible way of declaring dependence and enable the construction of sophisticated application logic.

A depending task type, or depending task, is a task template that depends on other tasks. Depending task types are not actual tasks to be scheduled for execution. Instead, a depending task type will be activated and instantiated to one or many tasks, called depending task instances, with the same stack and heap ranges and fi as specified in the depending task type's TCB, and the depending task instances are tasks that will be scheduled to VPCs for execution. The type and instance model can efficiently support the common situations in

large-scale data processing programs where the same tasks are created and executed multiple times upon data changes.

A depending task type W is not activated until other tasks change specific memory ranges that are "depended" by W. We call these ranges W's "memory dependence sets" (MDS). The MDS is specified by the parent task using an operand of the `spawn` instruction, and we use the MDS_range_count field in the TCB (refer to Figure 4.5), which stores the number of ranges in the MDS (0 for a regular task), to distinguish a depending task from a regular task. The depending task-specific data structures impose almost no space overhead for regular tasks as the MDS_ranges list, which stores the memory dependence sets, is variable-size.

Suppose a task, R, has created a depending task type W. When R exits and commits, W is enabled by the scheduler (transition to the "depending" state in Figure 4.4) and stays in the depending task type pool. When some other tasks or depending tasks write W's memory dependence sets and commit the changes to the global memory space, the memory subsystem notifies the scheduler, and the scheduler creates a depending task instance W^* and moves W^* to the task pool, making it schedulable. At the same time, W is still at the "depending" state and new commits to its memory dependence sets will also activate W and new depending task instances from W will be instantiated. A depending task instance can delete the corresponding depending task type by specifying the `:cd` or `:ad` option of the *exit* instruction when it commits or aborts its changes. The depending task instance can also choose to only commit or abort its changes by specifying the `:c` or `:a` option and leave the depending task type in the "depending" state to create more instances. This way, many instances can be created efficiently in the system without invoking many `spawn` instructions and it is ensured that after each commit to the MDS ranges of a depending task type, at least one depending task instance is created. Hence, no changes are missed by the depending task type before a depending task instance of it deletes the type.

With the depending task mechanism, the programs can naturally express and focus on the true dependence among tasks and data. Therefore, programmers can easily construct programs for not only embarrassingly parallel but also complex, iterative, or even interac-

tive computation. The MAZE system automatically resolves the dependence and efficiently executes the program with parallelization.

4.4 I/O, signals and system services

For simplicity and flexibility, i0 adopts the memory mapped I/O for MAZE to operate on I/O channels by accessing memory ranges in DR.

For example, we map the 1-byte memory range at `0x100000200` to the standard input (STDIN) and the range at `0x100000208` to the standard output (STDOUT). The task can input from STDIN by reading the address `0x100000200` and output to STDOUT by writing to the address `0x100000208`. The MAZE translates the memory accesses to these addresses to requests to the VPC for I/O operations. For example, one task can read one byte from STDIN and write the value to STDOUT by the following instructions.

```
conv 0x100000200:sb, 0x300001000:sb # read one byte from STDIN
conv 0x300001000:sb, 0x100000208:sb # write the byte to STDOUT
```

This design does not require additional instructions—all i0's memory access instructions can also be used for I/O. We can use this mechanism to support many kinds and a large number of I/O channels which may be distributed in many VPCs.

In a MAZE, many system events may occur. We design a signal mechanism to enable tasks to catch the information of events of interest in the system. Signals are notifications of these events. The signal mechanism is built on the depending task mechanism and the system services (to be discussed below). The signals are represented as changes to specific memory ranges, and tasks (depending tasks) can "depend" on these memory ranges to get notified.

To support system functions, such as the I/O and signal mechanisms, several distributed system services cooperate with the scheduler and VPCs in MAZE. For example, the I/O services on VPCs manage I/O channels, and the "system state" service broadcasts "system

idle" signals. On top of the compact and efficient core parts of the MAZE as described, it is easy to implement various system services to provide a rich set of functions.

4.5 Programming on MAZE

Being Turing-complete, i0 instructions are capable of expressing any computational logic. They can also emulate the semantics of instructions in other ISAs. Additionally, we design i0 as an extensible instruction set so that more instructions can be added to i0 if it is sufficiently beneficial to do so. More importantly, the parallelization and dependency control mechanisms along with the snapshot-based memory model provide a convenient and efficient way to design concurrent programs in the "lock-free" style [56].

We use an example to show how to naturally express task dependence and avoid race conditions using depending tasks. In Section 6.2, we show how to design and implement an improved MapReduce framework, vMR, on Layer Zero. Using the `spawn` instruction, we can easily create 2 *sum_task* tasks introduced in Section 4.1 to calculate sums in parallel. In this example, we design a new task to add the 2 sums together. The new task, implemented as a depending task shown in the following code, "depends" on the output of the 2 *sum_task* tasks.

```
1  sum_depending_task:
2  ;read the heap addresses of the depended tasks and
3  ;  the address to store the result from the stack, and stores
4  ;  them in 0x300001000, 0x300001008, and 0x300001010 (omitted)
5  ;check whether the depended data are ready
6  b:e $0:ue, 8(0x300001000), keep_depending
7  b:e $0:ue, 8(0x300001008), keep_depending
8  ;sum the two integers as the sum_task
9  add (0x300001000):ue, (0x300001008), (0x300001010)
10 conv $1:ue, 8(0x300001010):ue              ; exit code
```

```
11  exit:cd                                     ; commit and delete itself
12  keep_depending:
13  exit:a                                      ; abort
```

The *sum_depending_task* is activated and a depending task instance is created when either of the *sum_task*s commits and can check their exit codes to determine whether both operands for the sum operation are ready. This mechanism can scale to handle the paradigm that one depending task depends on one, several or many tasks by depending on an array of memory ranges updated by the depended tasks and checking the data in these memory ranges. The array of depended memory ranges are specified as an operand to the `spawn` instruction for creating the depending task. The depending task keeps "depending on" the data by the depending task type it depends on if either of the two depended tasks has not exited and committed. Only after all data are ready, one *sum_depending_task* instance executes `exit:cd` to commit its changes, exit and delete the depending task type. As shown in this example, programmers only need to create the depending tasks, and the dependences are automatically resolved by the scheduler and depending task related mechanisms in the MAZE.

Although we show the examples in i0 in this section, compilers can certainly be involved in the programming in the Layer Zero environment and transform programs written in high-level languages into i0 code. With a sufficiently sophisticated compiler, it is also possible to port traditional software to Layer Zero by recompiling them. We have designed and implemented the c0 programming language [4], the cc0 compiler [6] and the libi0 library [13] to program on MAZE. Providing richer support for programming on MAZE is one piece of our future work.

Although one program instantiated as many tasks runs in a MAZE at a specific time, many MAZEs can run concurrently in one cluster by isolating the computing resources to RCs. In the CCMR research testbed (introduced in Chapter 6), we set up multiple MAZEs on the same cluster and many users can develop and run their programs in a time-sharing manner.

4.6 Discussion

Cloud computing is an emerging and important computing paradigm backed by datacenters. However, developing applications running in datacenters is challenging. We design MAZE—a big virtual machine that is general, efficient, scalable, portable and easy-to-program—as an approach to developing a next-generation computing technology for cloud computing. Giving programmers an illusion of a "big machine", we design the i0 as the programming interface and abstraction of MAZE.

In addition to the programming interface, the underlining data substrate also plays an important role for large-scale data processing. In next chapter, we will discuss VOLUME, the solid data substrate for large-scale data processing on MAZE.

CHAPTER 5

IMPROVED DATA SUBSTRATE FOR LARGE-SCALE DATA PROCESSING

Chapter 4 presents the design of i0 and MAZE for in-memory computation. In this chapter, we will discuss the design of VOLUME, a distributed transactional virtual memory system with persistency support, as the substrate of data handling for programs on MAZE.

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.

We highlight the challenges and outline our approaches as follows.

- VOLUME aims to provide a large uniform memory space in which each VPC or node can address any subset of the memory space. The data that a single node accesses may well exceed the size of its physical memory and local hard drives combined together, and, consequently, data may reside in local physical memory, local hard drives, remote physical 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.

- 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.
- We design VOLUME to provide a transactional behavior and ensure the writes of multiple data by a task are atomic. Our implementation of vMR leverages the atomic guarantee to ensure its correctness.

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.

5.1 System overview

Figure 5.1 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 container*s (VMC). The system has many VPCs 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 from the upper layer, and swap data among the physical memory, disks, and remote VMCs through the network.

To assist a task's accesses to the memory space, the VMC for the VPC/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.

VOLUME is the memory subsystems of the MAZE. As discussed in Chapter 4, the MAZE manages 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 and dispatches tasks to VPCs for execution.

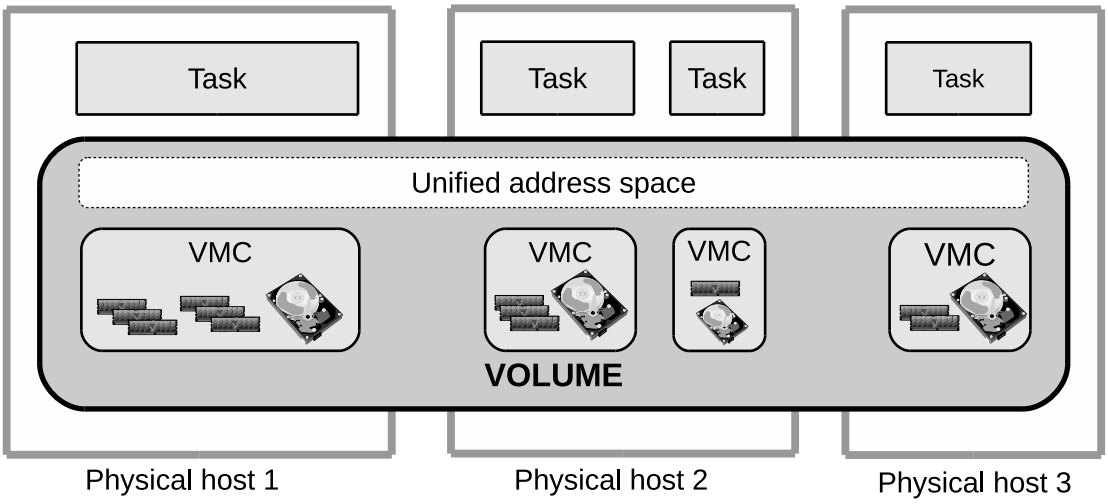


Figure 5.1: VOLUME architecture

5.2 Unified memory space

VOLUME's memory space should provide the memory capacity of exabytes to support large-scale programs in datacenters. However, implementations of commonly used 64-bit CPUs in datacenters only support limited memory capacity of up to hundreds of terabytes. In the near future, the situation is not likely to change significantly. To support processing of large datasets, we should extended the capacity of the memory space on top of the commodity CPUs. On the other hand, as VOLUME abstracts the computing resources at a low level that is close to the hardware and emphasizes efficiency, the system should introduce little overhead. Hence, we design the multi-space memory organization and mechanism in VOLUME. A space is like a segment with a space selector. Inside each space, tasks access memory ranges by the offsets. The memory layout is shown in Figure 5.2.

In the memory space, each task runs within one space and references memory ranges by the offsets in the space, and space-related work, such as mapping an offset to the full memory address, is needed only when a task starts/exits or a page fault is handled. The space size is configured to be within the underling CPUs' capacity so that the memory accesses in

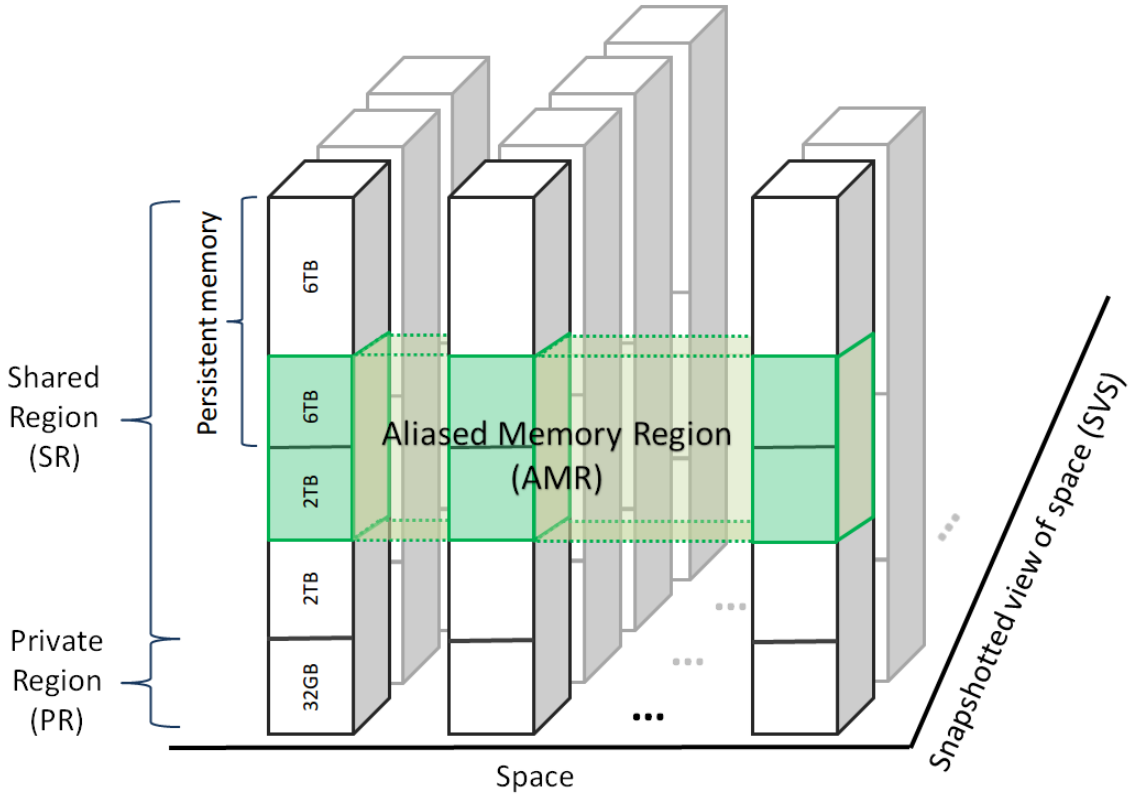


Figure 5.2: Organization of the memory space

VOLUME can be directly mapped to memory accesses on the physical CPUs.

On the other hand, tasks which run in different spaces need to share data across the spaces. For example, a task T1 in one space that creates a child task T2 to run in another space by the `spawnx` i0 instruction may need to construct T2's stack in the other space. To support data sharing among spaces, we design the aliased memory region (AMR) which consists of ranges (aliased ranges) with the same offset and length from all spaces as shown in Figure 5.2. Changes committed to one aliased range are also "committed" to all other aliased ranges. The design enables the efficient implementation that actually only maintain the content of one aliased range for the AMR. In the above example, T1 can allocate T2's stack in the AMR inside T1's space and construct it, and T2 can access its stack in the AMR inside T2's space.

To facilitate different usage of the memory ranges by programs, each space is divided into two regions–the *private region* (PR) and the *shared region* (SR). The starting address of the SR is the *region boundary* (RB) which is just after the address of PR's last byte. Inside PR, there is a special region, *direct-addressing region* (DR), in which memory ranges can only be referenced in the direct addressing mode. The DR allows the program to access system resources, such as RAMR and I/O channels (discussed in Section 4.4), with the unified operand representation using memory addressing.

A write in an individual task's PR does not affect the content stored at the same address in other tasks' PRs. In contrast, the SR is shared among all the tasks in the sense that a value written by one task can be visible to another task in the same MAZE, with VOLUME's memory consistency model controlling when values become visible. A task can use the PR to store temporary or buffered data because of its very small overhead, and use the SR to store the main application data which may be shared among tasks.

5.3 Snapshotted memory space

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. The consistency model of a shared memory system specifies how the system appears to programmers, placing restrictions on the values that can be returned by a read, and is critical for the system's performance [22].

5.3.1 Snapshot-based consistency

VOLUME provides the atomicity and snapshot semantics to programs and implements the consistency model that ensures the same properties as standard snapshot isolation [28] adopted by many relational databases. Each task has its own page table and 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. In the snapshot-based memory consistency model, every task sees a consistent view of the memory space, called snapshotted view of space (SVS), of a specific instantiation time as shown in Figure 5.2, and the system imposes a sequential ordering of tasks' commits. 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 SVS. 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 unsuccessful commits is low.

VOLUME's snapshot-based consistency model *relaxes the read/write order* on different snapshots. After a snapshot is created for a task, later updates to the associated memory ranges by other tasks do not affect the state of the snapshot. Hence, when reading data from an address in a snapshot, a task always receives the value the task itself has written most

recently or the value stored at the address at the time the snapshot is created. VOLUME permits concurrent accesses optimistically, and detects write conflicts automatically. With this consistency model, it is possible to implement atomic writes of a group of data, which are commonly used in transactional processing and other reliable systems requiring ACID properties. This model indeed introduces slight constraints on the organization of programs: only one of many tasks that concurrently write to the same location can commit successfully and the other tasks need to give up or retry. However, it assures programs written this way are much easier to understand and reason about their correctness, as well as improves the system's scalability.

As discussed in Section 5.1, the VMC captures 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.

The page table for a task is organized as a list of page metadata arrays for the list of memory ranges in the task's snapshot. Hence, to locate a page's metadata in the page table, the VMC takes $O(logM)$ time where M is the number of memory ranges. The number of memory ranges is usually much smaller compared to the number of pages in a task's snapshot. Hence, page metadata locating operations take approximately $O(1)$ time and the metadata management cost is tiny as compared to fetching pages through the network. The VMC may prefetch up to a limited number of pages when it fetches pages from a remote VMC to avoid page faults for later page accesses. In addition, the scheduler schedules tasks to VMCs considering the locality information so that fewer pages are fetched through the

network. Hence, the number of pages fetched through the network is at most $O(P)$ for a task that accesses P pages.

5.3.2 Managing snapshot metadata

The consistency model of VOLUME provides programs a clearly-defined memory semantics and enables scalable parallel processing. However, it is challenging to *efficiently* manage a large number of snapshots in the memory space. The snapshot's semantics, which ensures that any later changes of the memory content should not affect the existing snapshots, requires that the VOLUME maintain more than one version of the memory content. To efficiently support the snapshot management, the VOLUME organizes the SR as a sequence of fixed-size pages and each page has two versions—a *current version* and a *non-current version*. The current version is the latest committed version, and a snapshot consists of a number of pages of their current versions at the time the snapshot is created. The non-current version of a page is not included in newly created snapshots. A page's previous version before the current one, which may be still used by some tasks, can be tracked with the non-current version.

With the two-version design, VOLUME only needs to maintain a small amount of metadata for the pages, and can efficiently handle a large number of snapshots. Meanwhile, the current and non-current version design is friendly to the read-only sharing of data—many tasks can read shared data with snapshot isolation and complete their execution given they write to disjoint locations. The two-version design indeed implies a slight limitation that some tasks may need to wait for available page versions. Typically, there are many tasks schedulable in a MAZE and the waits usually occur to a small fraction of them. Hence, the limitation does not lead to noticeable performance penalty in practice. On the other hand, this design enables the system to scale, efficiently managing many concurrent snapshots.

To assist the tasks in using the shared and snapshotted SR in the unified memory space, components of the MAZE cooperate to manage the snapshots and facilitate the memory accesses. The scheduler manages the tasks and coordinates tasks to execute on VPCs taking

locality into consideration. For managing the tasks' snapshots and the memory space, VOLUME has a home service to coordinate the snapshots. 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 [49, 73]. 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. The home service serializes the committing of the memory ranges only when tasks exit.

5.4 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 in SR which programs can access through regular memory operations as shown in Figure 5.2. 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 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 service tasks' memory accesses in the persistent memory with data on local or remote disks. For efficiently locating 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.

5.5 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 [52, 85]. 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 Figure 5.3, 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

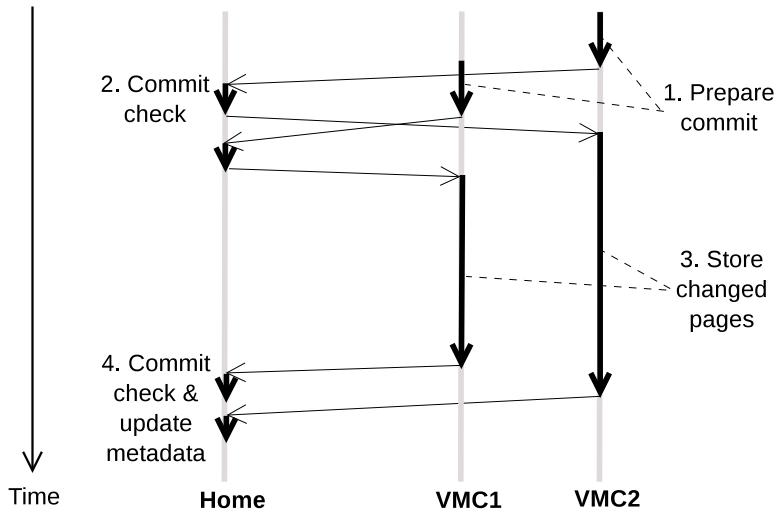


Figure 5.3: VOLUME commit protocol (two commits proceed concurrently)

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. In our current implementation, the metadata for each page consumes at most 16 bytes (8 bytes for each version) of memory at home on average since pages of a continuous range with the same metadata are managed together, and each page is 4096 bytes. That is, at most 0.78% $(2 * 16/4096)$ of the total commit operations 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 [42]. 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.

5.6 Fault tolerance

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 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.

To support programs that require high availability and throughput, VOLUME provides mechanisms that programs can use to replicate data to different VMCs. We made several design choices to give programmers a consistent experience with the overall memory system. First, as VOLUME unifies the physical memory and disk resources of many hosts in a memory space, we chose to also design the replication mechanism based on the memory abstraction. Second, the VOLUME ensures the snapshot-based consistency for the data replication. This provides programmers the same easy to reason about semantics of changes to data in the memory space no matter whether the changes are replicated or not. Third, the programs make the choice of which data to replicate and how many replicas as long as the memory system can support, while the memory system ensures the data are replicated across VMCs.

With respect to the basic choices, we design the replication mechanism that is flexible and simple to use. The overall persistent memory is partitioned to multiple replication regions which are memory ranges of the same size. How the space is partitioned is part of the system configuration. According to the configuration, the programs can include memory ranges from multiple replication regions in its snapshot and write the data that need to be replicated in ranges in a chosen number of replication regions. The VOLUME stores the changed pages

in different replication regions on different VMCs according to the system configuration rules during the commit process. With this mechanism, the program can choose the data to replicate and the number of replicas by common memory operations. As changes to all replicas are within a single snapshot, the consistency model is the same with or without data replication.

5.7 Locality and scheduling

To better control the scheduling of tasks and improve the locality, we implement and add two scheduling algorithms—the deterministic scheduling and the best-effort locality and latency (BELL) scheduling algorithms—to Layer Zero's meta-scheduler which uses a FIFO scheduling algorithm to assign tasks in the task queue to available VPCs by default. The deterministic scheduling algorithm assigns tasks to VPCs by hashing the IDs of tasks to VPC IDs. With this algorithm, the scheduling for tasks to VPCs 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 influence the scheduling decisions of the meta-scheduler. In general, it also improves the locality compared to the FIFO scheduler for jobs that run repeatedly. The more sophisticated BELL scheduling algorithm uses the delay scheduling techniques [93] and may skip some tasks from the front of the task queue and assign the task that has better locality on an available VPC/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 VPCs. If a task at the head of the queue has been skipped for a certain period of time, it is dispatched to a VPC even if the locality is poor.

5.8 Programming considerations

As shown in Section 1.3, programs are organized as tasks. Each task works on its own snapshot and commits changes atomically. Programming techniques for database systems

or transactional memory systems can also be applied for programming on VOLUME. As unsuccessful commits waste the execution of tasks, we design the commit mechanism to detect problematic commits as early as possibly by the *early abort* mechanism as discussed in Section 5.5. On the programmers' side, programmers can also tune the performance of programs following the similar rules for traditional transactional memory systems [53], such as "large transactions are preferable" and "small transactions should be used when violations are frequent".

Table 5.1: Cost for accessing N pages in the memory range of M pages

Memory access pattern	**Pages fetched through network** Bad locality	Good locality	**Size of snapshot data structure + metadata** (bytes)
Sequential	$O(N)$	$O(1)$	$\sim N/8 + 8 * N$
Clustered *	$O(N)$	$O(1)$	$\sim N/8 + 8 * N$
Random	$O(N)$	$O(N)$	$\sim M/8 + 8 * N$

$*$ Clustered memory access means accessing memory with temporal or spatial locality.

For common memory access patterns in big data processing programs, VOLUME can support the memory accesses efficiently. Table 5.1 shows the cost of the memory system for facilitating the accesses of N pages from the memory range of M pages. The number of pages fetched through the network reflects the main cost in the memory system as we will show in Section 6.4.1, and the size of the snapshot metadata reflects the cost of the metadata management. As VOLUME only fetches memory pages accessed by the task, $O(N)$ pages are fetched even if the locality was bad. For sequential and clustered memory accesses, the scheduler may schedule the tasks to execute on VMCs with good locality and avoid fetching memory pages through the network. The tasks accessing memory in a range of M pages include the M pages in the snapshots. For the sequential and clustered memory accesses, M equals to N because the programmers can include only the accessed pages in the snapshots. The snapshot data structure only includes the version numbers of pages and the metadata for the accessed pages are fetched from the home on demand. Each version number is represented by 1 bit and the metadata for each version of a page is 8 bytes. In Table 5.1, the size of the snapshot data structure and the fetched metadata for random memory accesses is

the largest one. However, the cost is relatively small with respect to the number of pages accessed. For example, for facilitating a task accessing 1GB (262,144 pages) data in a 1TB (268,435,456 pages) memory range, the snapshot data structure and metadata used are only around 34MB.

5.9 Discussion

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.

Together with the i0 and other MAZE subsystems, VOLUME provides a solid substrate for general, scalable and efficient large-scale data processing. Above the distributed virtual memory, it is easy to construct systems, implement programs or improve the performance of traditional systems such as MapReduce. In Chapter 6, we show how to construct the enhanced MapReduce system, vMR, on VOLUME, and the evaluation shows that VOLUME enables vMR to efficiently handle data.

CHAPTER 6

LAYER ZERO IMPLEMENTATION AND EVALUATION

We have implemented and tested i0, MAZE and VOLUME on several platforms. Our evaluation shows that MAZE has excellent performance and scalability and is faster than Hadoop, X10 and Spark.

6.1 Implementation

We implemented i0, VOLUME and other MAZE components on x86-64 hardware. The implementation of the MAZE comprises around 31,000 lines of C and assembly code. On each host, the MAZE runs in conjunction with a local Linux OS as the base system which manages local hardware resources including CPUs and local file systems. We emulate i0 instructions on x86-64 machines in our current implementation by using binary translation to generate x86-64 code on the fly during the execution of a i0 program.

We have implemented and run MAZE on multiple platforms as listed below.

- The CCMR research testbed. As shown in Figure 6.1(a), the CCMR testbed is constructed for cloud computing research. It is composed of 50 servers with Intel Quad-Core Xeon processors. The servers are connected by 10Gbps and 1Gbps networks.
- The Amazon Elastic Compute Cloud (EC2). To examine MAZE's portability and scalability on public IaaS platforms, we conduct part of the experiments on Amazon EC2 instances of the m1.large type in the us-east-1a zone.

(a) CCMR (b) TH-1/GZ

Figure 6.1: The CCMR and TH-1/GZ supercomputer

- The TH-1/GZ supercomputer. To examine how MAZE performs in a high-performance computing (HPC) environment, we conduct some experiments on the TH-1/GZ supercomputer (Figure 6.1(b)) which follows the same architecture of Tianhe-1A [90] ranked currently number 5 in the TOP500 list [86].

The implementations on EC2 and TH-1/GZ also show the portability of i0 in virtualized, heterogeneous and HPC environments. Although the platforms have quite different underlying hardware configurations and different Linux kernels, it proves to be straightforward to port i0 to all platforms by adding logic to match specific kernel data structures. Application programs run unchanged at all platforms without recompilation.

6.2 Fast MapReduce on VOLUME

VOLUME provides a flexible and efficient data substrate for the datacenter platform. To investigate how to use distributed virtual memory to enhance existing system designs with VOLUME, we develop vMR, which is a new MapReduce framework with enhanced performance. Existing MapReduce programs can be easily ported to vMR. Different from earlier MapReduce designs, the data flows among map and reduce tasks in vMR are handled by the memory abstraction provided by VOLUME.

The programs that run on vMR are also written in the c0 programming language. To write a vMR program, the programmer implements the `map`, `reduce`, `combine` and `vMR_finish` functions. `vMR_finish` is called by the vMR framework after one job finishes. Both the `combine` and `vMR_finish` are optional. To start a vMR job, a program calls the `vMR_run` function with the job specification. Programmers can easily design programs for iterative computation by calling `vMR_run` in the `vMR_finish` function. A default `partition` function ($hash(key)\ mod\ R$ as in MapReduce [42]) is integrated in vMR and programmers can override the partition function to control the partitioning of intermediate key/value pairs.

vMR handles a job with a set of map tasks and reduce tasks as shown in Figure 6.2, following the MapReduce model. Each task in vMR has a *status code* which is a value in a memory range that indicates whether a task is completed successfully. To start a job, vMR splits the input data and creates map tasks which parse key/value pairs out of the input data, pass them to the programmer-defined `map` function, partition the produced intermediate key/value pairs, group and pass them to the `combine` function if it is defined, and store the intermediate data in VOLUME. After all the map tasks finish, vMR creates reduce tasks. Each reduce task collects the intermediate key/value pairs for its partition, sorts and groups them by the keys (shuffle), passes the keys with lists of values to the programmer-defined `reduce` function, and stores the output in VOLUME. After the reduce tasks finish, vMR invokes the callback function `vMR_finish`. vMR distributes the tasks to a plurality of compute nodes to exploit parallelism. If there are not enough VPCs for all the tasks at the same time, the tasks will be maintained in queues and dispatched in multiple waves for execution.

6.2.1 Execution control

As there are many map and reduce tasks running in parallel, ensuring the dependence relation among them is a critical part of MapReduce systems. To make sure that tasks execute in a correct order, Hadoop and the implementation described by Dean et al. [42] adopt the centralized execution control and require a master program or master node to assign tasks to workers or slave nodes. vMR uses *depending task*s `mbarrier` and `rbarrier` to check

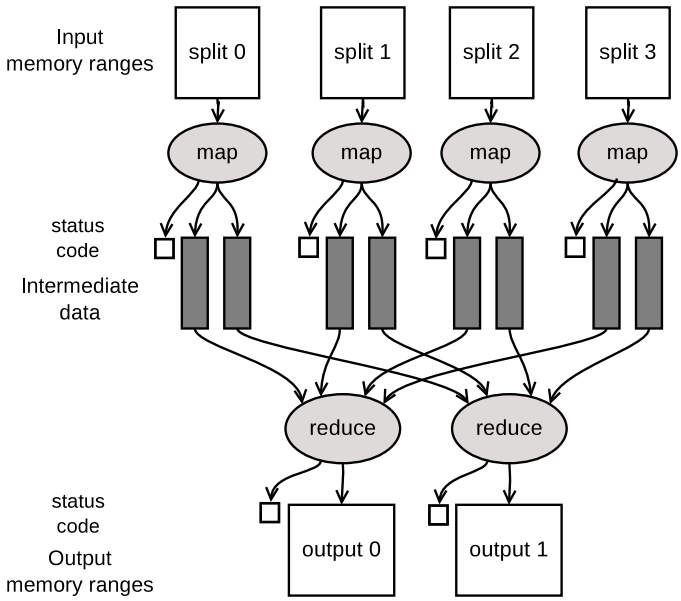


Figure 6.2: vMR tasks and data flows

map and reduce tasks' status codes and ensure the dependence relation. The scheduling and coordination function in vMR are provided through the tasks and depending tasks, and the location of data to process are efficiently and conveniently passed to the tasks as memory addresses in VOLUME. Hence, the execution control logic in vMR is distributed and vMR does not require a master program or master node.

6.2.2 Data flow

The execution of a parallel program in a system mainly consists of the task execution and data flows across compute nodes. How the data flows are managed and how data and processors are combined are critical to the system's performance. For example, researchers find that many performance problems of Hadoop can be attributed to its storage system for accessing data [24, 58, 83]. vMR handles data through VOLUME to support efficient job execution. The data flows among tasks in vMR are shown in Figure 6.2.

Input and output. Input and output are basic data flows for programs. Because many

programs require storing the input and output persistently, vMR can be configured to read the input and store the output on the persistent memory. Other programs or vMR jobs can access the data with normal memory operations.

In-job data flows. Besides the input and output, another dataset during the vMR job is the intermediate key/value pairs generated by the map tasks. MapReduce and Hadoop store the intermediate data on the nodes' local disks by I/O operations which increases the latency of accessing the data. To efficiently maintain and service the intermediate data, vMR stores the data in the non-persistent region of VOLUME and reduce tasks can access these data with the memory addresses. With memory ranges distributed across many hosts, VOLUME automatically transfers the accessed memory ranges from remote hosts to the host where the task runs. Specifically, during the execution of a job with M map tasks and R reduce tasks, a map task retrieves its split of input data from a memory range and stores the R pieces of partitioned intermediate data (some pieces may be empty) into R memory ranges, and a reduce task retrieves M pieces of intermediate data produced by the M map tasks and stores the output into a memory range. In our current implementation, the network communication for shuffling and the computation in reduce tasks on different VPC/VMCs are overlapped. An alternative method is to implement the shuffle phase as depending tasks checking the map tasks' status codes and collect the intermediate key/value pairs from completed map tasks. With the second method, the network communication for shuffling and the map task execution are overlapped.

Cross-job data flows. Besides of the input, output and intermediate data handled by MapReduce systems with the restricted programming model, many programs, particularly those with multiple iterations, need to share data among tasks and jobs. On Hadoop, programs use the files in HDFS [11] for sharing data. To support sharing data among tasks and jobs, we design the *shared area* mechanism which is flexible to use and provides strong consistency guarantee based on VOLUME. The shared area $sarea$ is a range of memory for tasks to store shared data. The programs can use it as a program-specific data structure and access structured data with low latency. The snapshot semantics of VOLUME makes the state changes of $sarea$ easy to be reasoned about: the changes to $sarea$ by one task is visible

to other tasks only after the task atomically commits successfully. Hence, a failed task's changes or a task's intermediate changes to $sarea$ will not be read by other tasks and the $sarea$ is always in a consistent state. In the implementation of k-means on vMR/VOLUME as discussed in Section 6.4, we use $sarea$ to store the centroids which are shared among tasks across iterations of jobs in memory.

6.2.3 Atomic commits

To ensure correctness, the map and reduce tasks must commit their output atomically [42]. In the design from Dean et al., the atomic commit of the output of a map task relies on the coordination of the master, and the atomicity of a reduce task output's commit is guaranteed by the atomic rename operation provided by the underlying file system [42]. vMR ensures the atomic commits of map and reduce tasks' output using the transactional semantics of VOLUME discussed in Chapter 5, requiring neither a master nor assistance from the file system. With the semantics of atomic commits, it is straightforward to implement backup execution in vMR.

The requirement of atomic operations in MapReduce-like systems is one of the reasons why VOLUME provides the transactional semantics. The atomic commits are useful or critical for many other systems and applications in datacenters, such as key/value stores and database-like systems [40, 41]. The atomicity semantics simplifies the failure handling, makes it easy for programmers to reason about programs' behavior, and enables backup executions of tasks to improve performance and reliability.

6.2.4 Fault tolerance

With the atomic commits, the semantics of VOLUME in the presence of faults discussed in Chapter 5 enables vMR to handle faults gracefully. The arrangement of vMR job executions in the presence of failures is similar to MapReduce [42]. To assist handling faults in vMR, we use the "system idle" signal mechanism which writes the current time to a memory range when the system is idle. The depending tasks in vMR include the memory range for the

signal into their memory dependence sets, and will be notified and made schedulable when the system is idle. With the signal mechanism, vMR handles the cases of faults as follows.

- The map or reduce tasks fail. The depending task (`mbarrier` or `rbarrier`) identifies the task as a failed one if it takes much more time than the other map tasks to complete and re-runs the task. The time is read from the memory range for the "system idle" signal. As vMR ensures atomic commits of map tasks, the depending task re-runs the failed task as backup execution.
- The depending tasks fail. The depending tasks depend on the "system idle" signal range to ensure that they will be eventually notified again even if one instance of the task failed.

Some intermediate data may be lost because of the failures of some nodes. The current implementation handles this fault by terminating the job and reporting the failure to the program which may re-run the whole vMR job. A possible mechanism to handle this more gracefully is that the reduce tasks requiring the lost intermediate data abort and update their status codes to indicate the failure, and, according to the status codes, the `rbarrier` re-runs the map tasks that generated the lost intermediate data and the aborted reduce tasks.

6.2.5 Implementation

vMR is implemented in c0 and compiled to an i0 program that runs on a MAZE. All the memory accesses in vMR are automatically serviced by VOLUME. As vMR is built on VOLUME and MAZE dispatches the tasks, the vMR implementation is compact, comprising 3083 lines of source code.

6.3 Computation in the big virtual machine

We measure MAZE's mechanisms with microbenchmarks, compare MAZE's performance with Hadoop and X10's, and inspect its scalability. We choose Hadoop and X10 for com-

parison because they are representatives of two mainstream approaches to datacenter computing and their implementations are relatively mature and optimized. Hadoop represents a class of MapReduce-style programming frameworks such as Phoenix [80], Mars [54], CGL-MapReduce [47] and MapReduce online [39], and has been used extensively by many organizations [10]. X10 represents a class of language-level solutions, including Chapel [31] and Fortress [23], targeting "non-uniform cluster computing" environments [34].

We use several workloads to evaluate MAZE's efficiency and scalability. We first design the *prime-checker* which uses 1000 tasks to check the primality of 1,000,000 numbers. As the prime-checker is highly parallelizable, we use this arithmetic application to evaluate the scalability of MAZE as we add more compute nodes—whether MAZE can achieve near-linear speedup. To check MAZE's performance on widely used and more complex algorithms, we implement the k-*means* clustering algorithm [71]. The k-means program iteratively clusters a large number of 4-dimensional data points into 1000 groups. This reflects known problem configurations in related work [60, 64, 97]. We run k-means on MAZE by scaling the number of compute nodes and the size of datasets to evaluate MAZE's scalability, and compare to Hadoop and X10. Additionally, to examine how MAZE scales as the overall memory usage grows, we implement and run *teragen* which produces the TeraSort [18] input datasets of various sizes.

For each technical solution in comparison, we apply the highest performance setting among the standard configurations. For example, X10 programs are pre-compiled to native executable programs, and we write the x86-64-based microbenchmark programs in comparison in the assembly language. Note that the platforms have different performance characteristics. To make the performance comparison fair, we use the same dataset and equivalent job configurations for Hadoop, X10 and MAZE.

On one physical host, the system overhead of MAZE is comparable to that of traditional VMMs. On 16 physical hosts, the MAZE runs 10 times faster than MapReduce/Hadoop and X10. On 160 compute nodes in the TH-1/GZ supercomputer, the MAZE delivers a 12.99x speedup over the computation on 10 compute nodes. The implementation of MAZE also

Table 6.1: Microbenchmark results

Benchmark	Solution	Time (second)
Arithmetic and logic operation	Native	7.13
	MAZE	7.14
	Xen	7.14
	VMware	7.35
Memory operation	Native	110.60
	MAZE	126.41
	Xen	112.43
	VMware	128.37

allows it to run above traditional VMMs, and we verify that MAZE shows linear speedup on a parallelizable workload on 256 large EC2 instances.

6.3.1 Microbenchmarks

We run microbenchmarks to measure the virtualization overhead of MAZE and compare the results with traditional virtual machine monitors. We also measure the overhead and efficiency of creating tasks, executing tasks and memory fetching. In addition, we measure the performance gains from register-based optimization.

Virtualization overhead. We design microbenchmarks to measure the overhead of CPU and memory virtualization in MAZE. To assess the overhead of CPU virtualization, we measure the arithmetic and logic operation performance by a microbenchmark program that executes 2^{34} *add* operations. Memory is another important subsystem in the MAZE, and we shall verify that the distributed memory subsystem does not incur dramatic overhead on one machine. We measure the memory operation performance with another microbenchmark program that writes 2^{29} 64-bit integers (4 GB in total) to the memory for 256 times and then read them for 256 times.

We implement the microbenchmarks in x86-64 assembly and i0, and execute the programs in Xen, VMware Player, native Linux, and a MAZE on the same physical host. To achieve the best performance, the x86-64 assembly implementations heavily use registers.

Table 6.2: Time for creating and executing a task

Operation	Operation step	Time (ms)
Create a task	Create snapshot	0.4
	Create the task	0.4
	Update VPC	0.5
	Overall	1.3
Execute a task (empty task)	Create snapshot	0.4
	Initialize VPC	0.4
	Update VPC	0.5
	Invoke the task	0.2
	Commit snapshot	0.9
	Overall	2.4

In the MAZE, the binary translator translates i0 instructions into x86-64 instructions during the execution.

Table 6.1 shows the results of the microbenchmarks. From the results, we can see that the virtualization overhead for arithmetic, logic and memory operation of MAZE is 0.1–15% over native execution on the physical host, which is comparable to traditional VMMs. In particular, MAZE exhibits even higher performance than VMware on one physical host. For the memory microbenchmark, the time on MAZE includes 7.47 seconds for creating and committing the memory snapshot for the task.

Task overhead. Task creation, scheduling, and termination are important and frequent operations in a task execution engine. To examine MAZE's performance in this aspect, we measure the overhead of creating and executing a task. We use one parent task to create 1000 new empty tasks and execute these tasks. To measure the overhead accurately, we force the tasking operation to be serial by conducting the experiment using one VPC on one compute node so that we can obtain the completion time of each operation.

Table 6.2 shows the overhead of creating and executing a task and the time used in each step. The results show that the overhead of creating and executing a task in a MAZE is only several milliseconds. The small overhead enables MAZE to handle thousands of tasks efficiently, and gives programmers the flexibility to use a large number of tasks as they want.

Table 6.3: Execution time of the reader task

Configuration	**Time / CCMR** (second)	**Time / TH-1/GZ** (second)
2 VPCs (Remote memory fetching)	51.05	36.21
1 VPC (Local memory fetching)	6.85	5.63

Memory fetching. We measure the efficiency of the memory operations across multiple physical hosts by a *writer-reader* microbenchmark. The writer-reader microbenchmark program consists of two tasks—one, the *writer*, writes 2^{30} 64-bit integers (8 GB in total) into the SR, and the other one, the *reader*, reads these integers from the SR. We run the writer-reader program on MAZEs that consist of one or two VPCs residing on one or two nodes respectively to check the performance of memory fetching from the local node and the remote node through the network.

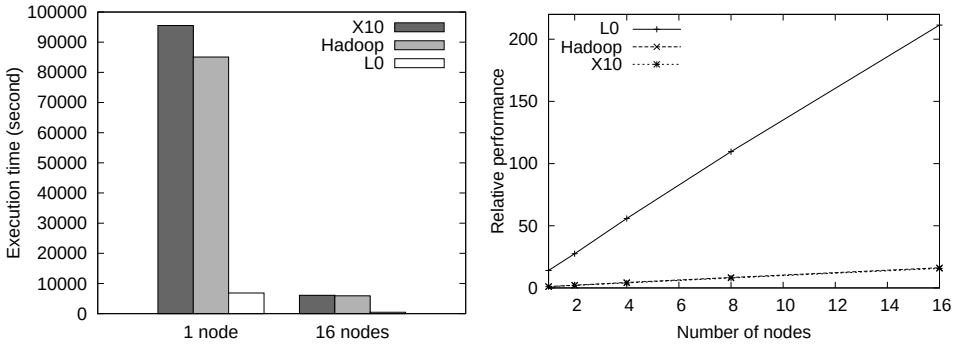
Table 6.3 shows the results of the writer-reader microbenchmark. In the 1-VPC experiment, it takes 6.85 seconds on CCMR and 5.63 seconds on TH-1/GZ for the reader to read the data. Although providing the advanced snapshot semantics and keeping two versions of the pages as discussed in Section 5.2, MAZE exhibits the page fault handling throughputs of 3.1×10^5 and 3.7×10^5 page faults handled per second on CCMR and TH-1/GZ respectively. The additional execution time in 2-VPC experiments is spent on the data transmission through the network. Applying copy-on-write may potentially further improve the performance.

Register-based optimization. To measure how much performance improvement a program can achieve by using RAMRs, we develop a *loop* microbenchmark since loops are very common in programs. The loop microbenchmark sums 4×10^{10} 64-bit integers in 4×10^{10} iterations. We implement the loop microbenchmark in i0 with and without putting the iterator and accumulator in RAMRs. We also implement and optimize it in assembly using physical registers on Linux for comparison.

Table 6.4 shows the execution time of loop on both MAZE and native Linux on the same

Table 6.4: Execution time of loop

Platform	Time / CCMR (second)	Time / TH-1/GZ (second)
MAZE, using RAMRs	31.83	27.30
MAZE, without using RAMRs	171.81	147.34
Native Linux	31.79	27.28



(a) Execution time on 1 and 16 nodes

(b) Speedups as the number of compute nodes grows

Figure 6.3: Execution time and speedups of k-means

Table 6.5: Execution time of prime-checker on Hadoop and MAZE

Number of nodes	1	2	4	8	16
Hadoop (second)	16661	8352	4169	2090	1112
MAZE (second)	15795	7885	3950	1975	986

compute node. From the results, we can see that, after using RAMRs, loop uses only 18.5% of the time of the other implementation not using RAMRs on both CCMR and TH-1/GZ. The results also show that loop on MAZE using RAMRs produces very close performance as the manually optimized one using physical registers on Linux on both CCMR and TH-1/GZ. This indicates that the binary translator can generate high-quality native code and also verifies the low virtualization overhead of MAZE.

6.3.2 Performance comparison

We run the performance evaluation programs with the same dataset on different numbers of

compute nodes on the CCMR testbed to compare MAZE's performance and scalability with X10 and Hadoop's.

Table 6.5 shows the time of prime-checker on Hadoop and MAZE. We scale the number of compute nodes to 16 physical hosts. The results show that both Hadoop and MAZE scale linearly as prime-check is easy to parallelize. For simple ALU-intensive workloads, both MAZE and Hadoop are efficient, but MAZE provides slightly higher performance in spite of its advanced instruction set features and memory model.

Figure 6.3(a) presents the execution time of k-means on Hadoop, X10 and MAZE with 1 and 16 compute nodes. The execution time on Hadoop and X10 is 11 times more than that on MAZE on 1 compute node. As shown in this evaluation, MAZE quickly exhibits superiority in efficiency as the complexity of application grows. The results on 16 compute nodes show that MAZE is at least 13 times faster than Hadoop.

Figure 6.3(a) presents that MAZE exhibits tremendous speed gains over Hadoop and X10, although Table 6.5 shows that MAZE only provides slightly better performance than Hadoop with simple ALU-intensive workloads. This observation, in fact, illustrates the advantages of the MAZE technology. When the computation is composed of one or two stages of easily parallelizable processing, the MapReduce/Hadoop frameworks can already accomplish algorithmically optimal performance. The MAZE can still provide slightly higher performance due to its efficient instruction-level abstraction. Moreover, the MAZE quickly outperforms existing technologies when the computation becomes more complex—i.e., containing iterations, more dependence, or non-trivial logic flows, which are common cases in programs. The system design of MAZE aims to provide strong support for general programs, and the speed gains of MAZE shown in Figure 6.3(a) reflect i0's generality, efficiency and scalability.

One may argue that the comparison between programs written in C-like languages on MAZE and Java-like languages on Hadoop is unfair. Indeed, the Java-like languages introduce overhead and there are efforts to make use of native code to improve Hadoop's performance such as Hadoop C++ Extension (HCE) [8] which implements data processing

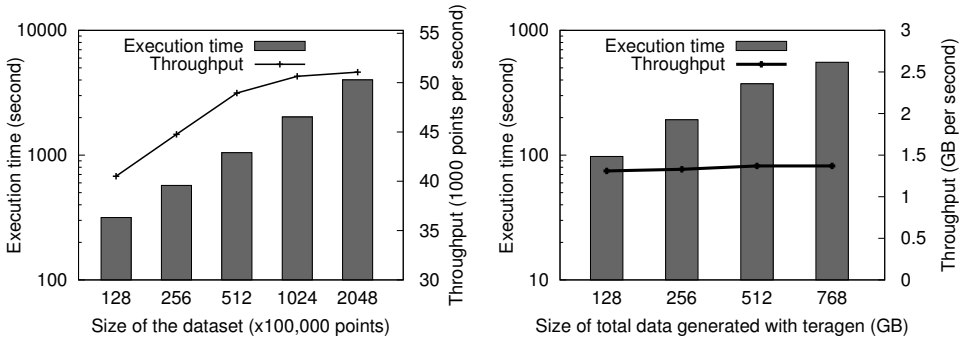
stages of Hadoop in C++. However, HCE only improves Hadoop's data processing performance of MapReduce jobs by 20-41%. Therefore, the difference in linguistic features does not decisively make a MAZE much faster.

It may also appear that the MAZE's memory-based data processing is the only reason for its superiority in performance. This is, in fact, an over-simplified view of the technical design space. X10 also handles program data through memory with PGAS (Partitioned Global Address Space) [34]. As shown in Figure 6.3(a), however, the MAZE is one order of magnitude faster than both Hadoop and X10. In fact, disk I/O does not necessarily outweigh other factors, such as network bandwidth and CPU capacity, in a datacenter computing environment. In our evaluation, we have implemented the same algorithm in all the three technologies and applied salient optimizations for each of them—e.g., compiling X10 programs to native x86-64 instructions—to ensure the workloads are comparable with the three implementations. It is the holistic design and the simple yet effective mechanisms in the MAZE that lead to the significant speedup over existing technologies, and it is the ISA-level abstraction that enables such a design and the mechanisms. As the result, MAZE performs at least as well as other technologies on all workloads, and delivers much better performance than others for complex programs.

Figure 6.3(b) shows the speedups for three technologies as we scale the number of compute nodes. All speedups are calculated with respect to the X10 performance on one compute node for each workload. The speedup of the k-means computation on MAZE is near-linear to the number of compute nodes used, which highlights the good scalability of the design and implementation of MAZE. We can also observe that the overhead of MAZE is small, taking into consideration the near-linear speedup and the fact that the execution time on MAZE in the 16-node experiment is very short compared to the time on Hadoop or X10.

6.3.3 Scalability

A scalable system may be required to scale with the data size, the processor count, the user population, or a combination of them. As user-scalability is not a goal of MAZE, our



(a) k-means (b) teragen

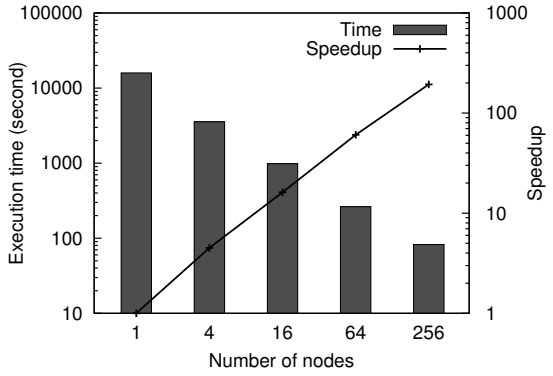
Figure 6.4: Scalability of MAZE with the data

evaluation focuses on the scalability with the data size and the number of compute nodes.

We run k-means on MAZE with 50 compute nodes on the CCMR testbed to evaluate MAZE's scalability with the dataset size by increasing the dataset from 12,800,000 points to 204,800,000 points. Figure 6.13(a) presents the execution time and throughput which is calculated by dividing the number of points by the execution time. The result shows that, on the same number of compute nodes, the execution time grows more slowly than linear growth with the dataset size while the throughput increases. Hence, MAZE scales well with the dataset size.

To examine how MAZE performs when the total memory usage grows, we run *teragen* on 32 compute nodes to generate data in the SR with increasing sizes. We stress the memory subsystem using physical memory only by generating data of up to 768GB which reaches the capacity of the 32 compute nodes' overall available physical memory. Figure 6.13(b) shows the execution time and throughput of teragen. The throughput is approximately the same as the total data generated increases from 128GB to 768GB, indicating MAZE's memory subsystem scales well with increasing memory usage.

To evaluate MAZE's scalability and also examine MAZE's performance in a public cloud, we run the prime-checker on a MAZE running on Amazon EC2 with up to 256 virtual machine instances. Figure 6.5(a) shows the execution time and speedup with respect to the performance on one instance. The results show that the MAZE scales very well to 256 com-



(a) prime-checker on EC2

Platform and # of nodes	Execution time (second)	Speedup
TH-1/GZ		
160	480	12.99
80	822	7.58
40	1578	3.95
20	3128	1.99
10	6233	1.00
CCMR		
50	4011	1.55

(b) k-means on TH-1/GZ

Figure 6.5: Scalability with the number of compute nodes

pute nodes—the speedup is near-linear as we increase the number of instances. Our ability to perform experiments with larger-scale and more sophisticated workloads was limited by the concurrent instance cap stipulated by EC2. In the mean time, the current result on 256 instances already shows that, although prime-checker is a relatively easy-to-parallelize program, the i0 architecture and the MAZE design easily scale to hundreds of compute nodes without showing signs of slowdown or obvious system bottlenecks. The experiment on EC2 also proves the good portability of MAZE in a virtualized environment, and that the MAZE can be used independently of or in combination with traditional virtual machine monitors.

6.3.4 Performance on the TH-1/GZ supercomputer

To examine the performance and scalability of MAZE in high-performance computing environments, we run k-means on the TH-1/GZ supercomputer with 10 to 160 compute nodes. The input dataset consists of 204,800,000 points.

Figure 6.5(b) shows the execution time and speedup of k-means on TH-1/GZ. The speedup is calculated with respect to the execution time on 10 compute nodes on TH-1/GZ. For comparison, we also show the execution time and speedup of k-means on CCMR with 50 compute nodes in Figure 6.5(b). The performance of the MAZE on 50 compute nodes on CCMR is between the performance of MAZE on 10 and 20 nodes on TH-1/GZ. Although these two

platforms have different performance characteristics, the results verify that MAZE can deliver efficient performance in different environments. From the results, we can see that when we scale the compute nodes from 10 to 160, the speedup of MAZE also increases to 12.99, which indicates that MAZE scales well in the supercomputing environment.

The experiments of MAZE on TH-1/GZ show that, although originally designed for datacenter environments, MAZE is portable to run on HPC platforms, conducting computation efficiently and scaling well. These results also suggest a potential of using cloud system software for high-performance computing.

6.4 Improved data substrate

We also 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 [42, 47, 57, 80, 91, 96], k*-means* clustering, a data mining workload [38, 98], *adjacency list*, a graph processing workload [48], and *sort*, a benchmark for data processing systems [19, 42, 75, 92]. Following identical algorithmic designs suitable for the programming models, we implement the benchmark programs or use standard implementations on vMR/VOLUME, Hadoop and Spark.

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 [42]. Although word count can fit into the MapReduce programming model well, Yoo et al. find that the scalability of word count on the shared-memory MapReduce on a single machine depends on the scalability of the memory management service in the OS [91]. As vMR is also built on a shared-memory model, we use word count to evaluate the performance and scalability of vMR and VOLUME. We implement word count following the implementation described by Dean et al. and the one in Hadoop distribution [2, 42]. The dataset for the *word count* is the 15GB text content of the static HTML dump of English Wikipedia in June of 2008 [20].

K-means clustering is an algorithm commonly used for data mining to iteratively partition a dataset into k clusters [71], which is one example of the set of algorithms that require iterative computation. Our implementations of k-means clustering mainly follow that of Mahout [3], and we adopt relevant optimizations used in related work [38, 98]. The centroids shared by tasks across iterations are stored in the $sarea$ on VOLUME. The datasets are a large number of 4-dimensional data points which are randomly generated by the standard *gensort* record generator for TeraSort [18]. 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. Adjacency list can prepare the graph to be used by PageRank-like computation [63]. Adjacency list 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 recommended in the related work [48], and contain around 100 million (30GB) and 500 million (150GB) vertices with an average out-degree of 7.2. We implement adjacency list on vMR following the implementation on Hadoop by Ahmad et al. [48].

Sorting is an important part of many computing tasks. Our implementation of the *sort* program on vMR/VOLUME follows the implementation on MapReduce [42]. The implementation on Hadoop for comparison is the *terasort* program from the Hadoop release. 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 [18].

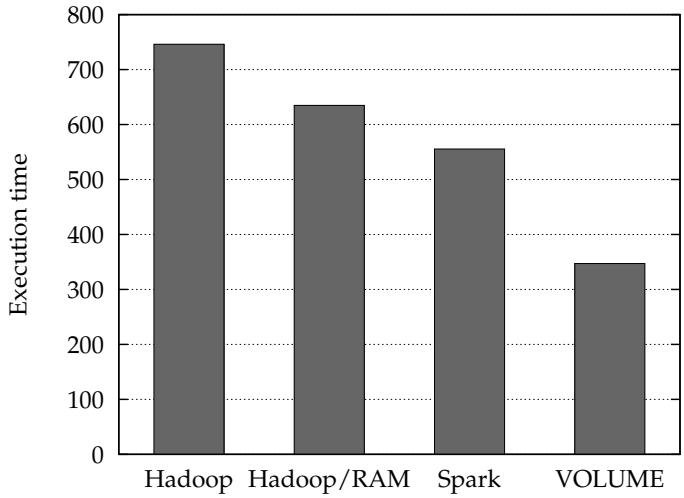
Although some of the workloads on vMR we used here are "embarrassingly parallel", the design and implementation of vMR itself shows that a relatively sophisticated program with barriers, atomic commits, data shuffling, cross-job data sharing and backup execution can be easily and efficiently implemented using the mechanisms of MAZE. As i0 can express any computational logic, more sophisticated programs can be implemented on MAZE. To achieve best performance, programmers should design "good" programs to make best use of i0 and VOLUME's mechanisms as discussed in Section 5.8.

During the evaluation, we configure the HDFS to store only one copy of data to avoid

additional cost of data replication for Hadoop. This configuration minimizes the I/O contention to which Hadoop is more sensitive. We allocate 1 map task slot and 1 reduce task slot on each node for Hadoop which requires static slot allocation for map and reduce tasks on each compute node, while we run only 1 task on each compute node for VOLUME and Spark.

6.4.1 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.



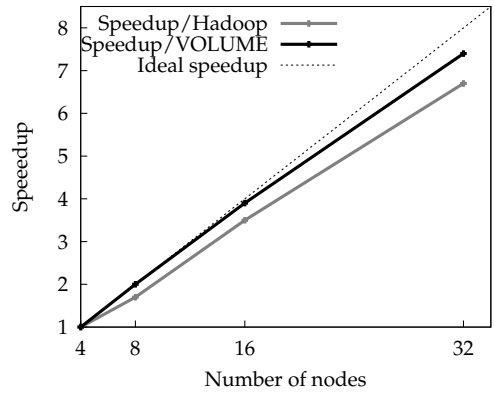
(a) Execution time

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

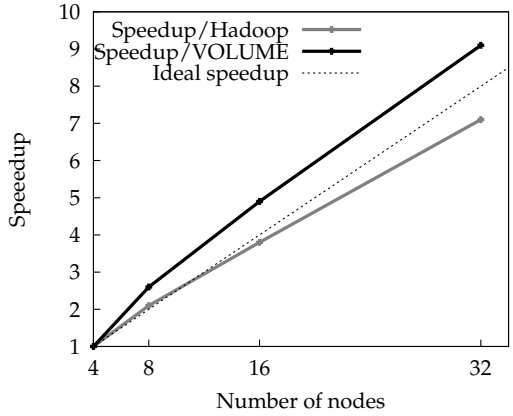
$*$ 14% of shuffle phase included.

(b) Time breakdown

Figure 6.6: Execution time and breakdown of sorting 50GB data on 16 nodes



(a) Word count



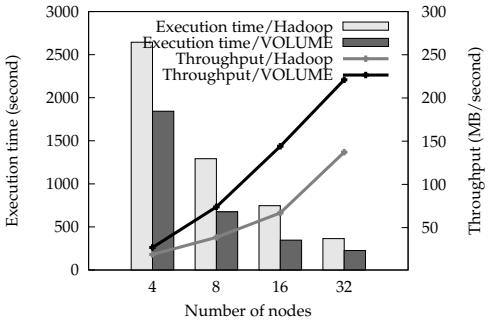
(b) Adjacency list

Figure 6.7: Speedups on VOLUME and Hadoop

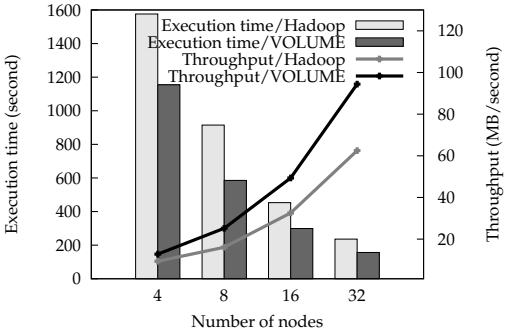
We first run *sort* on 50GB input using 16 nodes to evaluate VOLUME's performance when the intermediate dataset can be processed in physical memory. We also run *sort* on both Hadoop and Spark for comparison. The input is on disks managed by HDFS for Hadoop and Spark or the persistent memory for VOLUME, 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 Figure 6.6(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 Figure 6.6(b). We also run k-*means*, *word count* and *adjacency list* on the CCMR to evaluate the scalability and

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

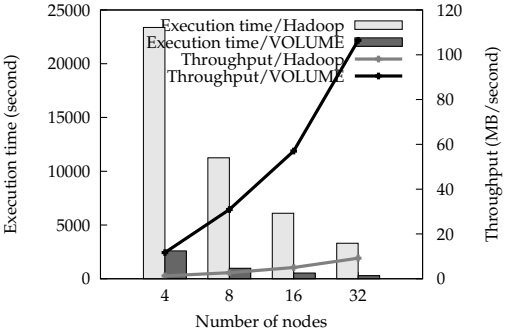
(a) Execution time of k-means clustering on 30 million points



(b) Execution time of sorting 50GB data



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

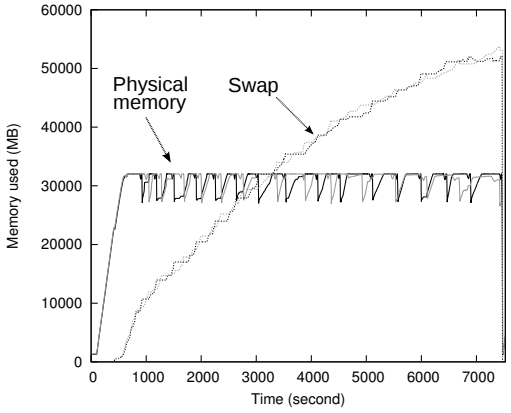


(d) Execution time and throughput of adjacency list on 30GB input

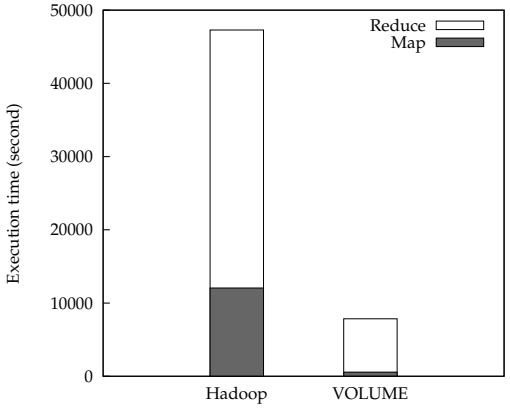
Figure 6.8: Performance of VOLUME on datasets of moderate sizes

performance of VOLUME on datasets of moderate sizes. Figure 6.7 and Figure 6.8 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.

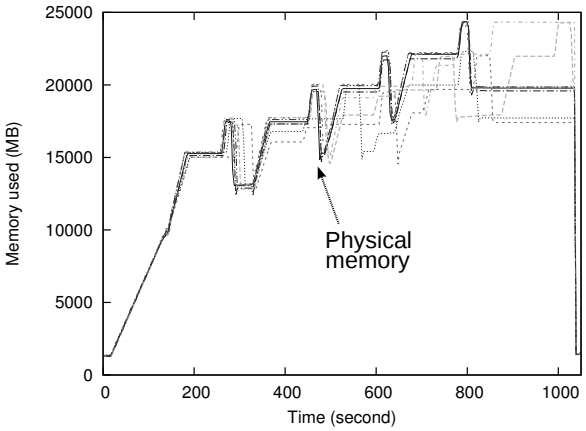
Understanding the performance. VOLUME accelerates the computation in several aspects: low-latency operations on the intermediate data and small system overhead, efficient sharing of data among tasks and jobs and efficient support to input and output. We find two of those are most effective. First, VOLUME transparently helps the program save part of the input and intermediate data generated by map tasks in memory and significantly accelerates data exchange in the shuffle phase. For *sort*, VOLUME and Spark/RDD that store intermediate data in memory [94] 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 through-



(a) Adjacency list on 2 nodes



(b) Execution time of adjacency list on 2 nodes

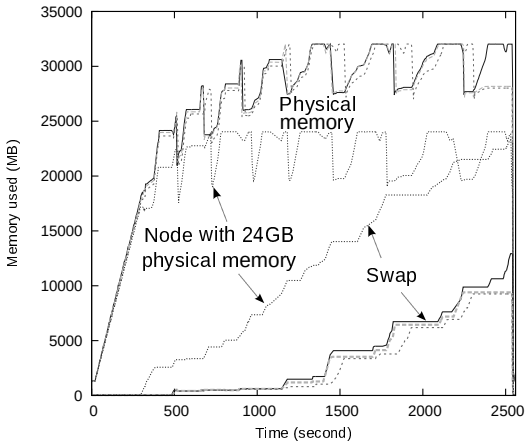


(c) Adjacency list on 8 nodes

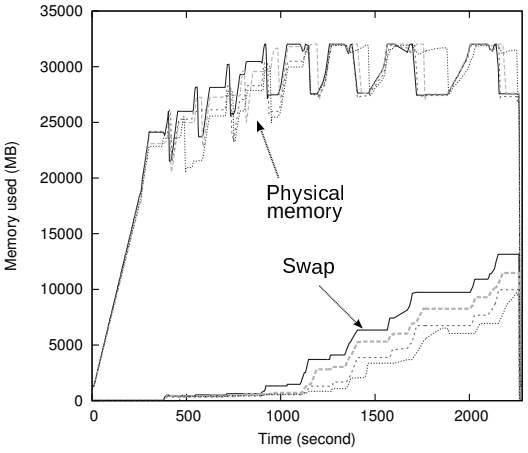
Figure 6.9: VOLUME memory usage

put. 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 Figure 6.6(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 that are much larger than the input data, and also improves the performance of workloads that are compute or disk I/O intensive, such as *word count* and *sort*.



(a) One node has 24GB physical memory and the other 3 have 32GB physical memory



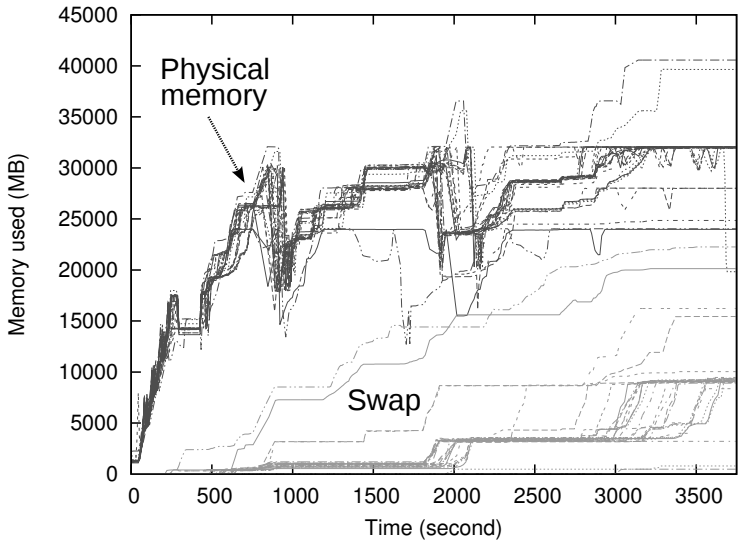
(b) All 4 nodes have 32GB physical memory

Figure 6.10: VOLUME memory usage of adjacency list on 4 nodes

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. Figure 6.9(a) shows the VOLUME usage on 2 nodes. We can see

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

(a) Sort on 32 nodes



(b) Sort 400GB data on 32 nodes

Task order	Execution time (second)	Page fault handling time (second)
0	37.21	6.29
1	38.29	4.03
2	42.51	4.38
3	48.26	4.98
4	0.05	0.048
5	449.72	310.20
6	961.31	823.40
7	1048.15	875.72

(c) Task execution time and page fault handling time on one node for sorting 400GB data

Figure 6.11: Execution time, throughput and memory usage of sorting

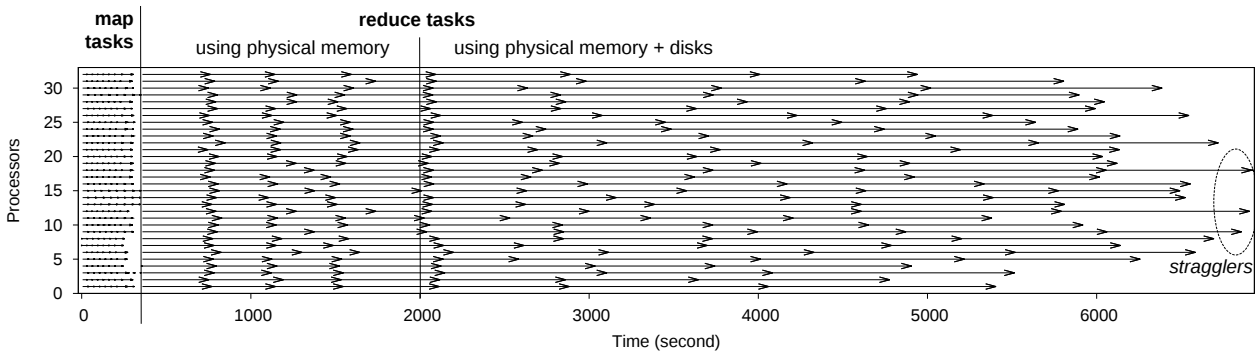
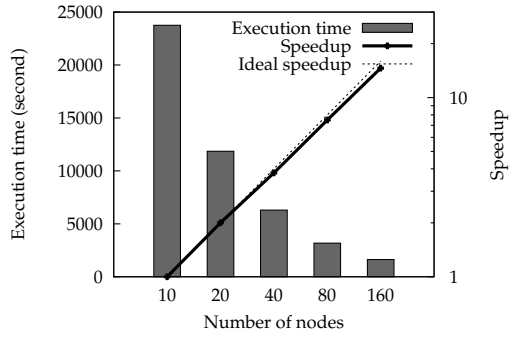


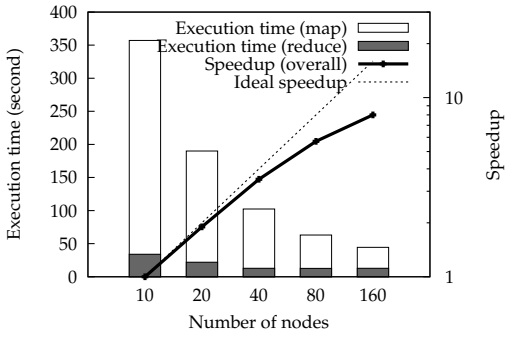
Figure 6.12: Progress of sorting 512GB data

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. Figure 6.9(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 Figure 6.9(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 Figure 6.7(b). Figure 6.9 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.

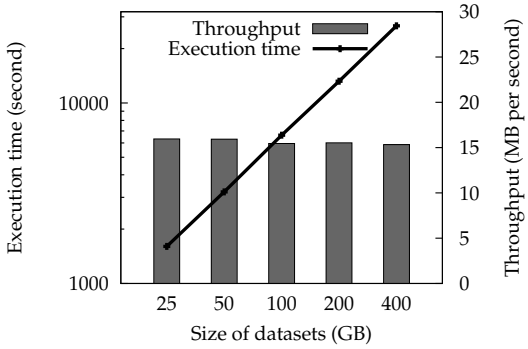
In addition to making use of disks to store data when the overall physical memory is not sufficient, VOLUME can also handle situations where the cluster has non-uniform physical memory capacity distribution among nodes. To check how VOLUME performs when the cluster has a heterogeneous physical memory capacity distribution, we run adjacency



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



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



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

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

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

Figure 6.13: Scalability of VOLUME on large datasets and many compute nodes

list on one 4-node cluster where one node has 24GB physical memory and the other three have 32GB physical memory and another 4-node cluster where all nodes have 32GB physical memory. Figure 6.10 shows the memory usage of adjacency list on these 2 clusters. Comparing these 2 memory usage patterns, we can see that VOLUME can adapt to different physical memory distribution well and makes use of more disks to handle data on nodes that have smaller physical memory capacity.

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. Figure 6.11(a) shows the execution time and throughput. As the dataset scales up, the throughput decreases because the percentage of data handled by disks increases and the virtual memory access speed in VOL-

UME decreases. We measure the VOLUME usage during running *sort* on a 400GB dataset as shown in Figure 6.11(b). 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. We plot the task execution time and page fault handling time on one node for sorting 400GB data in Figure 6.11(c). Task 0 to 3 are map tasks which read around 12.5GB of input from the persistent memory and store the intermediate data in the memory on the local VMCs. Hence, page fault handling which mainly reads data from disks to physical memory takes around 12% of the overall execution time of these tasks. Task 4 is a `mbarrier` depending task instance which checks the status of the map tasks in memory ranges fetched from local and remote VMCs. Task 4 shows that the barrier task implemented as depending task is very efficient although it checks every map task's status. Task 5 to 7 are the reduce tasks which fetch around 12.5 GB of intermediate data from other VMCs. Hence, the page fault handling which mainly fetches pages from physical memory on remote VMCs to local physical memory through the network takes around 82% of the overall execution time of these tasks. Task 6 takes around 510 more seconds than task 5 and the additional time is spent on page fault handling, while task 7 takes slightly more time than task 6 takes. The result shows that the physical memory is already used up (after around 600 seconds in Figure 6.11(b)) before the execution of task 6 and VOLUME uses disks for handling new data at a lower speed.

To look into how the usage of disks affect tasks' execution time, we plot the execution time of tasks during sorting 512GB data in Figure 6.12. 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 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. Figure 6.12 also shows that there are stragglers. Starting backup executions of them as discussed in Section 5.5 may further improve the overall performance.

6.4.2 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. Figure 6.13(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. Figure 6.13(b) 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 configured to 30 since the data in the reduce phase are relatively small for the datasets we used 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 Figure 6.13(c), 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. Figure 6.13(d) shows the execution time of *adjacency list* in the CCMR and TH-1/GZ. We can see that the per-node throughput on TH-1/GZ is higher than that on the CCMR, thanks to the high-performance hardware.

6.4.3 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 6.6: Time for writing persistent data and recovering the home service

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

TABLE 6.6 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.

CHAPTER 7

CONCLUSION

In this thesis, we first investigate the limitation of traditional disk-based systems. We analyze the limitations of MapReduce using parallel compilation as a probing case handling moderate-size data with dependences among numerous computational steps, and investigate the performance and challenges of in-memory data processing.

To support general large-scale data processing in datacenters, we organize the in-memory data processing in many compute nodes by presenting programmers an illusion of a big virtual machine. We design a new instruction set architecture, i0, to unify myriads of compute nodes to form the big virtual machine called MAZE, and present programmers the view of a single computer where thousands of tasks run concurrently in a large, unified, and snapshotted memory space. The MAZE provides a simple yet scalable programming model and mitigates the scalability bottleneck of traditional distributed shared memory systems.

We construct VOLUME, a distributed virtual memory as a system-wide data substrate and the memory subsystem of MAZE. VOLUME provides a general memory-based abstraction with transactional semantics, takes advantage of DRAM in the system to accelerate computation, and, transparently to programmers, scales the system to handle large datasets by swapping data to disks and remote servers for general-purpose computation in datacenters. Along with the efficient execution engine, VOLUME enables the MAZE to scale up to support efficient computation on large datasets.

We have implemented and evaluated MAZE on various platforms and environments. On VOLUME, we implemented vMR with enhanced functionality and high performance. Our evaluation shows that MAZE has excellent performance and scalability.

REFERENCES

[1] Amazon EC2. *http://aws.amazon.com/ec2/*.

[2] Apache Hadoop. *http://hadoop.apache.org*.

[3] Apache Mahout. *http://mahout.apache.org/*.

[4] C0 Manual. *http://lazero.net/doc/c0.pdf*.

[5] C0 Tutorials. *http://lazero.net/get-started.htm*.

[6] cc0. *http://sourceforge.net/p/cc0/code/ci/master/tree/*.

[7] Google Docs. *http://docs.google.com*.

[8] Hadoop C++ Extension. *https://issues.apache.org/jira/browse/MAPREDUCE-1270*.

[9] Hadoop Streaming. *http://hadoop.apache.org/common/docs/r1.2.1/streaming.html*.

[10] Hadoop users list. *http://wiki.apache.org/hadoop/PoweredBy*.

[11] HDFS. *http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html*.

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

[13] libi0. *https://github.com/layerzero/libi0*.

[14] Memcached. *http://memcached.org/*.

[15] mrcc. *http://www.ericzma.com/projects/mrcc/*.

[16] Rackspace. *http://www.rackspace.com/*.

[17] Salesforce.com. *http://www.salesforce.com*.

[18] Sort Benchmark. *http://sortbenchmark.org/*.

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

[20] Wikimedia Downloads. *http://dumps.wikimedia.org/*.

[21] Windows Azure. *http://www.microsoft.com/windowsazure/*.

[22] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.

[23] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al. The Fortress language specification. *http://research.sun.com/projects/plrg/fortress.pdf*, 2008.

[24] K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proc. of the 2011 Intl. Conf. on Management of data*, pages 1165–1176. ACM, 2011.

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM symposium on Operating Systems Principles*, pages 164–177, 2003.

[26] L. Barroso, J. Dean, and U. Hoelzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[27] L. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[28] 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*, pages 1–10, 1995.

[29] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[30] R. Buyya, T. Cortes, and H. Jin. Single system image. *Intl. Journal of High Performance Computing Applications*, 15(2):124, 2001.

[31] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Intl. Journal of High Performance Computing Applications*, 21(3):291, 2007.

[32] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

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

[34] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing, 2005.

[35] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *Proc. of the 17th annual intl. symposium on Computer Architecture*, pages 239–248, 1990.

[36] S. Chen and S. W. Schlosser. Map-Reduce meets wider varieties of applications. Technical Report IRP-TR-08-05, Intel Research Pittsburgh, May 2008.

[37] Y. Chen, D. Pavlov, and J. F. Canny. Large-scale behavioral targeting. In *SIGKDD'09*, pages 209–218, 2009.

[38] 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*, pages 281–288, 2007.

[39] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *Proc. of the 7th USENIX conf. on networked systems design and implementation*, pages 21–21, 2010.

[40] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[41] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally-distributed database. *Proceedings of OSDI*, page 1, 2012.

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

[43] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing. U.S. Patent 7,650,331, 2010.

[44] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, 2007.

[45] D. DeWitt and M. Stonebraker. MapReduce: a major step backwards. *http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/*.

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

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

[48] M. T. Faraz Ahmad, Seyong Lee and T. N. Vijaykumar. MapReduce benchmarks. *https://sites.google.com/site/farazahmad/pumabenchmarks*.

[49] 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*, pages 201–212, 1995.

[50] M. P. I. Forum. MPI: A message-passing interface standard. *http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf*, 2009.

[51] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP'03*, pages 29–43, 2003.

[52] R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[53] 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*, volume 32, page 102, 2004.

[54] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *PACT'08*, pages 260–269, 2008.

[55] B. Hedlund. Inverse virtualization for internet scale applications. *http://bradhedlund.com/2011/03/16/inverse-virtualization-for-internet-scale-applications/*.

[56] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, 1993.

[57] 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*, pages 41 –51, march 2010.

[58] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi. Evaluating MapReduce on virtual machines: The Hadoop case. In *CloudCom'09*, pages 519–528, 2009.

[59] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys'07*, pages 59–72, 2007.

[60] H. Jégou, M. Douze, and C. Schmid. Improving bag-of-features for large scale image search. *Intl. Journal of Computer Vision*, 87(3):316–336, 2010.

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

[62] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Proc. of the Linux Symposium*, volume 1, pages 225–230, 2007.

[63] A. Langville and C. Meyer. Deeper inside PageRank. *Internet Mathematics*, 1(3):335–380, 2004.

[64] D. Lee, S. Baek, and K. Sung. Modified k-means algorithm for vector quantizer design. *Signal Processing Letters, IEEE*, 4(1):2–4, 1997.

[65] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[66] H. Lu, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proc. of the IEEE/ACM Supercomputing 95 Conf.*, page 37, 1995.

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

[68] Z. Ma, K. Hong, and L. Gu. VOLUME: Enable large-scale in-memory computation on commodity clusters. In *CloudCom'13*, pages 56–63, 2013.

[69] Z. Ma, Z. Sheng, and L. Gu. DVM: A big virtual machine for cloud computing. *IEEE Transactions on Computers*, (99), April 2013.

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

[71] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, page 14, 1967.

[72] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[73] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[74] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proc. of the 9th IEEE/ACM Intl. Symposium on Cluster Computing and the Grid*, pages 124–131, 2009.

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

[76] 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*, SOSP '11, pages 29–41, 2011.

[77] 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.*, 43:92–105, January 2010.

[78] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[79] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI'10*, pages 1–14, 2010.

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

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

[82] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25:1363–1369, 2009.

[83] J. Shafer, S. Rixner, and A. Cox. The Hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE Intl. Symposium on*, pages 122 –133, march 2010.

[84] P. Snyder. tmpfs: A virtual memory file system. In *In Proc. of the Autumn 1990 European UNIX Users Group Conf.*, pages 241–248, 1990.

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

[86] TOP500 Supercomputer Sites. TOP500 List. *http://www.top500.org/list/2012/06/100*.

[87] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *ACM SIGPLAN Notices*, volume 30, pages 144–155, 1995.

[88] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[89] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD'07*, pages 1029–1040, 2007.

[90] 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*, 26(3):344–351, 2011.

[91] 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*, pages 198 –207, 2009.

[92] 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*, pages 1–14, 2008.

[93] 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*, pages 265–278, 2010.

[94] 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*.

[95] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, University of California, Berkeley, May 2010.

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

[97] R. Zhang and A. Rudnicky. A large scale clustering scheme for kernel k-means. *Pattern Recognition*, 4:40289, 2002.

[98] W. Zhao, H. Ma, and Q. He. Parallel k-means clustering based on MapReduce. In *CloudCom'09*, pages 674–679, 2009.