University of Alberta

Enhancing Cloud Environments with Inter-Virtual Machine Shared Memory

by

Adam Wolfe Gordon

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2011
Edmonton, Alberta

Abstract

Cloud computing has emerged as a popular mechanism for deploying services and running applications. A key feature of many cloud environments is the use of virtual machines (VMs) as the unit of resource allocation. Multiple virtual machines running the same application may be co-located on a single physical host, by either chance or design, in which case we can share memory between them, providing fast inter-process communication to applications. It is desirable to use inter-VM shared memory at the framework and service level, providing its advantages to applications transparently.

We present two use cases for inter-VM shared memory. Elastic Phoenix, a MapReduce framework allowing for malleable jobs, demonstrates how inter-VM shared memory allows the easy addition of features to existing frameworks, making them more suitable for cloud computing. Nahanni Memcached, a caching server for web applications, demonstrates how existing services can use inter-VM shared memory to improve performance.

Acknowledgements

Although the work presented in this thesis is my own, I must acknowledge and thank the many others who made it possible.

Thank you to my supervisor, Paul Lu, who provided valuable insights and observations, asked me the right questions, and kept me on track while giving me a great degree of freedom in my research. Without both the guidance and the freedom I would not have been able to complete this undertaking.

Thank you to all the members of the Trellis research group, especially my lab-mates Cam Macdonell, Xiaodi Ke, and Jeremy Nickurak, who were always willing to answer questions and have interesting discussions. Our group paper discussions were an extremely valuable part of my graduate school experience.

Thank you to Nelson Amaral, who gave me many opportunities and responsibilities as a teaching assistant in his course. Without a doubt, teaching was my favourite part of being a graduate student; for this I am grateful to Nelson, Paul, and the many CMPUT 229 and CMPUT 379 students who were a joy to teach.

Thank you to my wife, Ellen, who made my home life happy when school was easy and when school was hard. The support of Ellen, my parents, and the rest of my family and friends has been vital to my success.

Contents

List of Tables

List of Figures

List of Abbreviations

API	application programming interface
ATRL	average total read latency
EC2	Amazon Elastic Compute Cloud
GAE	Google App Engine
IaaS	Infrastructure as a Service
IPC	inter-process communication
KVM	Linux Kernel Virtual Machine
LAMP	Linux, Apache, MySQL, and PHP
MPI	Message Passing Interface
PaaS	Platform as a Service
VDE	Virtual Distributed Ethernet
VM	virtual machine
YCSB	Yahoo Cloud Serving Benchmark

Chapter 1

Introduction

Cloud computing has become a popular model for deploying applications and services. Cloud environments typically provide scalability and fault tolerance by allowing a user to deploy multiple virtual machines (VMs). Thus, applications and services designed for cloud deployment tend to have loosely-coupled components. Such a decoupled design helps make applications scalable because it is easy to add more resources as an application runs, for example by deploying additional virtual machines. However, in order to share data between nodes or components, the application must marshal the data, send it over the network, and unmarshal it; as such, communication overheads in cloud computing environments are higher than in traditional environments.

The technologies that drive cloud computing are not new. Full-system virtualization goes back to the 1970s [36], although virtualization on the now ubiquitous Intel x86 platform was uncommon and inconvenient until paravirtualization was popularized by Xen [19] and hardware virtualization extensions were introduced by both AMD and Intel in 2006 [15]. Multi-tenancy in data centres, allowing more efficient use of resources such as network infrastructure, has occurred for decades. However, cloud computing differs from traditional virtualization scenarios and multi-tenant data centres in two important ways. First, cloud computing environments offer a high degree of automation, allowing virtual machines to be provisioned with a few keystrokes and avoiding the need to purchase additional physical machines. Second, cloud computing provides computing power as a commodity, pricing it in a usage-based scheme. In other words, cloud computing makes computing resources available inexpensively and on demand.

In some cloud environments, there is a chance that two or more VMs running the same application will be co-located on the same physical host. VM co-location is particularly likely in private clouds, run by a single organization for internal users, which tend to en-

compass fewer physical machines than public clouds such as Amazon Elastic Compute Cloud (EC2) . Co-located VMs should be able to communicate much faster than VMs running on different hosts, as they have access to shared physical resources such as memory. Indeed, using the virtualized network technologies common in modern virtualization solutions, network communication between co-located VMs can be much faster than communication over the physical network. However, virtualized network technologies still incur domain-crossing and data copying overheads.

Suppose, however, that co-located VMs are able to share some memory between them, and potentially with applications running natively on the host machine. When VMs are co-located, certain data structures can be accessed directly by applications rather than being marshalled, sent over the network, and unmarshalled. Direct access should, intuitively, be faster than using the network. Additionally, this would allow applications designed for shared memory to be easily ported to cloud environments by making use of inter-VM shared memory instead of standard shared memory. Of course, shared memory is not a new idea, going back to at least the early 1990s [20], and shared memory has been used to optimize network communication in systems such as fbufs [31].

While inter-VM shared memory has great potential for both porting applications to cloud environments and improving the performance of cloud applications when VM co-location occurs, doing either for an arbitrary application requires significant development effort. Fortunately, many cloud applications are built using frameworks that abstract away concerns such as communication. We show how such frameworks can be modified to use shared memory, achieving either of the aforementioned goals — porting to cloud environments or performance improvement — with little to no added effort on the part of the application developer.

1.1 Porting a Shared-Memory MapReduce Framework

MapReduce [29], originally introduced by Google, has become a popular programming model for cloud computing applications. This popularity follows at least in part from the high level of abstraction provided by the model. The framework takes care of parallelization and communication, leaving the application developer to develop only application logic. The MapReduce model, though originally developed for large distributed systems, has grown so popular that MapReduce frameworks have been developed for many different environments. In particular, there are MapReduce frameworks for shared-memory

systems [41, 64].

Suppose a developer has built a MapReduce application for a shared-memory system, but has reached the limit of the shared-memory machine's resources. A natural suggestion would be to move the application to a cloud computing environment, where nearly arbitrary resources are available at low cost. However, cloud providers typically offer VMs with limited resources, expecting that users will provision multiple VMs to meet their total resource needs; this does not work for a shared-memory application.

By converting the MapReduce framework to use inter-VM shared memory, the user can deploy existing MapReduce applications to a cloud environment with few changes. For a single application, the effort required to convert the framework to inter-VM shared memory is likely to be similar to that required to convert the application directly. But, it is common to have multiple applications built on the same framework, in which case converting the framework saves significant effort. For example, a user might have one MapReduce application that extracts links from HTML pages, and a second MapReduce application that produces a reverse index from this output.

By modifying the underlying MapReduce framework to use inter-VM shared memory, the entire pipeline of MapReduce jobs can be moved to the cloud with little or no effort expended directly on each application. We develop this idea in Chapter 3.

1.2 Improving Web Application Performance

Web applications often have unpredictable load patterns, as the popularity of a website can grow, or shrink, rapidly, and the difference in traffic between high and low periods can be great. This unpredictability has made web applications a common application for cloud computing. The ability to dynamically add and remove machines makes it easy and cost-effective to scale an application up and down as demand changes. Additionally, web applications parallelize naturally, since user sessions are normally independent of one another.

Most web applications are built using multiple libraries and services. Often, a database stores user and application information, which is fetched to generate webpages and cached with a caching service for better performance. The caching service, which probably already keeps data in memory, is a good candidate for conversion to inter-VM shared memory. Using inter-VM shared memory for caching allows co-located VMs to access cached data with reduced latency, and reduces load on the caching server.

For example, a social networking website might be built in Python and keep its data in a MySQL database, which it uses a library to access. To reduce latency and improve database scalability, it might cache database results using memcached. In a typical configuration, the website will run on multiple web servers, with a load balancer, and memcached will also run on each of these servers such that they take advantage of the aggregate memory pool of all the servers.

Suppose that, in order to get automatic scalability, the website developers decide to host it with a cloud provider. Now, the individual web servers are actually virtual machines, and some of them may be co-located. These servers could share memory with the memcached server, allowing them to access some cached values with reduced latency. We develop this idea further in Chapter 4.

1.3 Contributions

We demonstrate the advantages of using inter-VM shared memory in application frameworks. Specifically, we show how inter-VM shared memory can be used to easily add features to existing frameworks, and how it can be used to reduce application latency.

1. We show how inter-VM shared memory can be used to add features to existing cloud programming frameworks. To this end, we present Elastic Phoenix, a MapReduce framework for shared-memory systems and co-located virtual machines, which allows for malleable jobs in which resources can be added and removed as the job runs. Elastic Phoenix extends the existing Phoenix [64] MapReduce framework with malleability, as well as the ability to run in VMs; these features are enabled by inter-VM shared memory. Using Elastic Phoenix, we show that adding processor resources to a MapReduce job as they become available can improve the response time of a workload by 29%, and that adding malleability induces minimal (6% to 8%) overhead for some classes of applications [62].

2. We show how inter-VM shared memory can be used to reduce the latency of cloud services, as presented in a recent refereed workshop paper [63]. To demonstrate, we present Nahanni memcached, a version of the popular memcached [7] object caching server that uses inter-VM shared memory to significantly reduce the latency of caching operations for applications running in co-located VMs. Using the Yahoo Cloud Serving Benchmark [27], we show that using inter-VM shared memory can

reduce the latency of cache reads by up to 86%, reducing the total latency of read-related operations in a workload by up to 45%.

1.4 Concluding Remarks

Through the scenarios in Sections 1.1 and 1.2, we have described where inter-VM shared memory can potentially be useful. Before presenting our contributions in full, we must describe some background. In particular, we will describe the technologies and mechanisms — cloud computing, inter-VM shared memory, data-centric programming models, and web frameworks — on which our contributions build.

Chapter 2

Background

Our work is built upon a number of existing mechansims, technologies, and ideas, which we will discuss here in order to lay a foundation for our contributions. We will also try to give some historical perspective, showing how we fit into the larger body of work in cloud computing.

2.1 Cloud Computing

Cloud computing describes environments in which computational resources are provided as a service [18], usually by a third party. Typically, cloud computing providers allow provisioning of resources on demand, and charge for resources based on usage. Cloud providers can make efficient use of their physical resources by sharing them among many users, thus allowing them to sell these resources inexpensively [38]. The cloud computing model allows developers of applications and services to scale up and down on demand. Because cloud resources can be sold inexpensively, developers of applications and services can deploy services with low up-front costs. Cloud computing eliminates the need for developers to buy hardware that will sit idle initially and then be outgrown as a service's popularity grows.

Under the umbrella of cloud computing there are various models, differentiated primarily by the level of abstraction provided atop computational resources. Infrastructure as a Service (IaaS) clouds (e.g. Amazon Elastic Compute Cloud (EC2) [1], Eucalyptus [48], OpenStack [10]) give users access to full virtual machines, a low level of abstraction. In contrast, Platform as a Service (PaaS) clouds (e.g. Google App Engine [4], AppScale [24], Microsoft Windows Azure [14]) provide an application programming interface (API) developers can use to write applications that will be deployed on third-party infrastructure and scaled automatically.

The popularity of *public cloud* services such as Amazon EC2 and Google App Engine has led to the development of *private clouds*, cloud environments run by a single organization for internal users. Private clouds provide the advantages of public clouds (on-demand resources, high utilization) while mitigating the security concerns [43, 53] associated with offloading storage and computation to a third party. *Hybrid clouds* combine public and private resources into unified systems [57].

2.2 Cloud Resource Allocation and VM Co-Location

The basic unit of resource allocation in IaaS clouds is a virtual machine. The typical method of scaling a service that runs in the cloud is to provision more VMs as demand for the service grows. This method of resource allocation requires that services intended for the cloud have loosely-coupled components capable of running in separate virtual machines, likely located on separate physical hosts, and potentially in geographically disparate locations.

This architecture affects the performance of cloud applications. In particular, network performance between VMs running an application may be unpredictable, depending on where the VMs are placed. There is currently active research on network-aware placement and migration of virtual machines [45, 47, 50]. Placement approaches focus on determining which VMs in a cloud environment are likely to communicate heavily, either through simulation or by requiring developers to provide some hints. With such information, the cloud scheduler can run heavily interacting VMs on physical nodes connected by fast networks. Migration approaches determine which VMs communicate heavily as an application runs, and migrate VMs that communicate heavily to be logically near each other.

2.3 Nahanni Inter-VM Shared Memory

Nahanni [9,46] is a mechanism in QEMU/KVM that allows a POSIX shared memory region on a host to be mapped into the address space of one or more VMs running on the same host. Consequently, the co-located VMs and the host can share data and perform interprocess communication (IPC) through the shared memory region. Nahanni is implemented as a virtual PCI device with an on-board memory region that can be mapped using a special driver in the guest operating system. The programming interface of Nahanni is essentially identical to that of POSIX shared memory, with slight differences only in how the memory region is initialized at application startup.

Of course, VMs can only take advantage of Nahanni if they are co-located. We do not

include an analysis of co-location probabilities here. However, we argue that private cloud deployments, increasing in popularity, will tend to encompass fewer resources than public clouds, and that as the core counts of commodity servers continue to increase, these resources will be concentrated in fewer and fewer individual servers. Thus, it will be increasingly common for multiple VMs running an application in such a cloud to be co-located on a single physical host. While multiple servers will still be required for many scales, and will absolutely be required for high availability, those that do happen to be co-located can make use of Nahanni.

2.4 Programming Environments for Cloud Computing

Managing distributed resources, as cloud environments require, is a difficult task for programmers. For optimal performance it is important that computation take place near where data is stored, reducing communication overhead. Data parallel programming models have thus become popular, as they allow programmers to write distributed applications without reasoning much or at all about where data will be located or how to parallelize computation.

Perhaps the best-known of these models is MapReduce [29]. Inspired by the standard functional programming functions map and reduce, MapReduce allows application developers to provide just the application logic, with the framework taking care of dividing up data and placing computations near their input. While Google's MapReduce and the well-known Hadoop [3] implementation target distributed environments, the MapReduce model has been implemented for shared-memory systems as well [41, 64]. A number of higher-level languages have been built on top of Hadoop MapReduce. These include Hive [59], an SQL-like query language and Pig [49], a dataflow language.

Dryad [40] is a data parallel programming environment developed by Microsoft. More general than MapReduce, it specifies programs as directed acyclic graphs, with each vertex as a computation and edges specifying data dependencies. It has been integrated into the Microsoft .NET language suite in the form of DryadLINQ [65]. Comet [39] is a stream processing framework that integrates with DryadLINQ.

Spark [66] introduces a new programming model based on using parallel operations to operate on distributed datasets. It provides something of a middle ground between the limited model of MapReduce and the general model of Dryad. Piccolo [52] provides a programming model based on partitioned tables and moving computation to data.

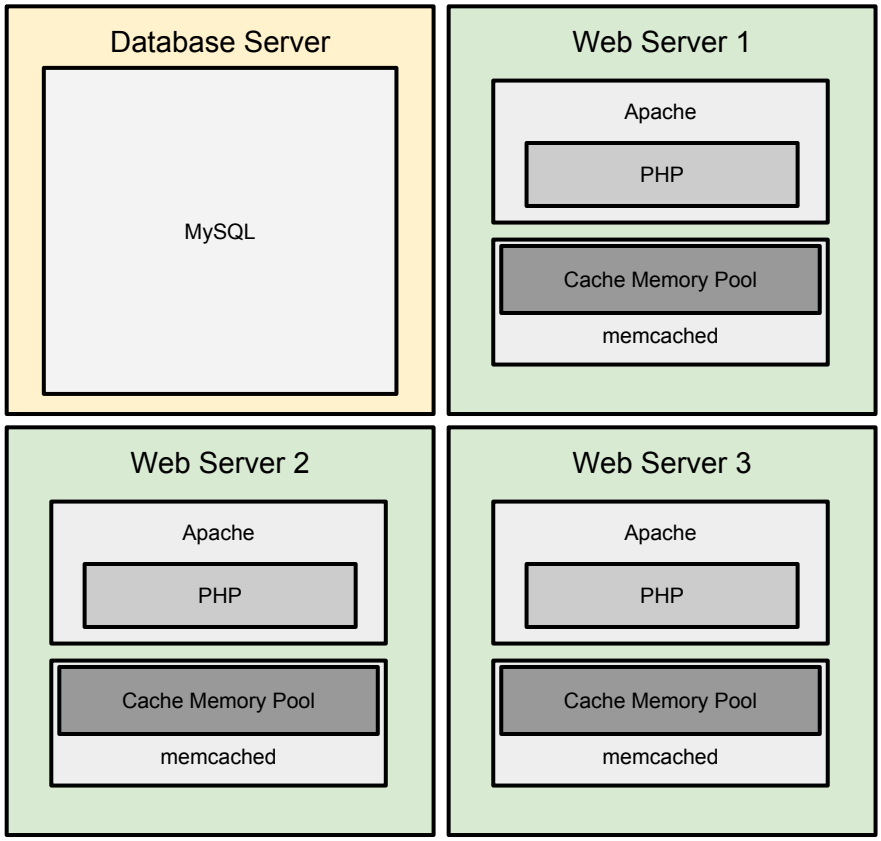


Figure 2.1: A Typical Non-Cloud Web Application Architecture

2.5 Cloud-Based Web Applications and Data Management

Before cloud computing, many web applications were built using the popular Linux, Apache, MySQL, and PHP (LAMP) software stack, and deployed as illustrated in Figure 2.1. In this model, applications are built in the PHP language, which runs as a subprocess in the Apache web server, usually on a Linux host. As each user session to the web server is independent, the web application can be scaled by adding more web servers, and load balancing over them, using simple mechanisms such as a DNS round-robin. On the other hand, MySQL scales up (to a more powerful server) better than it scales out (to multiple servers), and thus it usually runs on its own, more powerful server. To take load off MySQL, it is common to run a caching service such as memcached [7], which takes advantage of any extra memory on each web server to provide a distributed cache of common database queries.

LAMP does not translate well into cloud environments. In particular, as we noted above, MySQL scales out poorly [22], making it ill-suited to the multi-VM model of cloud computing. Cloud web applications are written in a wide variety of languages, and typically

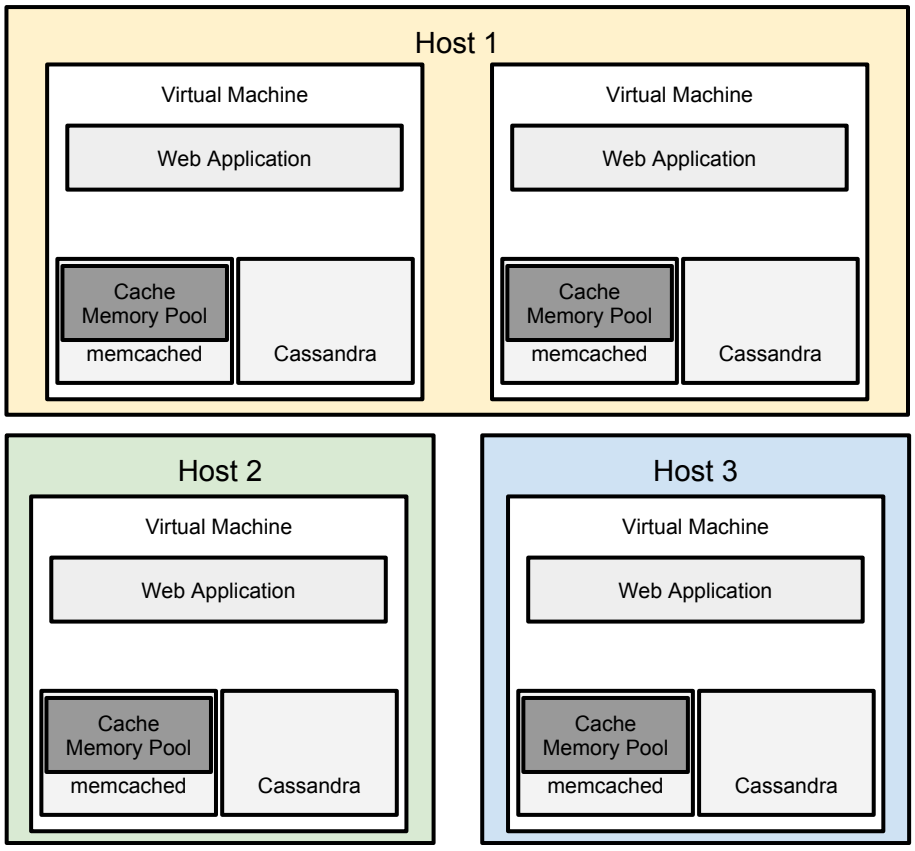


Figure 2.2: A Typical Cloud Web Application Architecture

operate as illustrated in Figure 2.2. The cloud-based model focuses on making applications scale out, rather than up. The developer builds a virtual machine containing all the necessary components for the web application, and deploys as many instances of it as needed to service demand. In IaaS clouds such as Amazon EC2, such deployment can be done automatically according to load. As shown in Figure 2.2, the VMs hosting a web application are deployed to multiple physical servers, potentially being co-located together. In typical IaaS environments, the developer deploying the application does not control, or care, on which physical servers the VMs run.

Datastores offering simpler, usually non-relational and non-transactional, semantics scale out better than relational databases such as MySQL, and have become standard in cloud computing environments. Collectively, these datastores are known as NoSQL.

Google's BigTable [23] is one such distributed datastore, with a table model. Data is distributed across machines using row blocks and column families. Unlike traditional relational databases, BigTable does not provide support for multi-row transactions. It also versions data, appending instead of overwriting existing rows. Open source datastores mod-

eled after BigTable include HBase [5] and HyperTable [6].

Amazon's Dynamo [30] is a distributed key-value store, designed for high availability. It provides very simple key-value semantics and weak consistency guarantees in order to improve scalability and availability. Scalaris [12] and Voldemort [11] are open source key-value stores similar to Dynamo.

BigTable and Dynamo are at two ends of a feature spectrum, with other datastores falling in between. Amazon's SimpleDB [2] provides database semantics without transactions. Cassandra [44], an open-source datastore originally developed by Facebook, provides BigTable-style tables with Dynamo's peer-to-peer model. MongoDB [8] is a "document-oriented" datastore, providing key-value like semantics with value query support.

Web applications for the cloud are often built using frameworks, abstracting away data management concerns. Some frameworks, such as Google App Engine [4] and Microsoft Azure [14] are intended to abstract away scaling concerns, adding resources to applications automatically as demand grows. These frameworks provide developers access to proprietary resources through an API. AppScale [24] is an open source framework that is API compatible with Google App Engine.

2.6 Concluding Remarks

Having explained the mechanisms we use and given some perspective for our contributions, we can now describe our new work. First, we will discuss Elastic Phoenix, a malleable MapReduce framework for shared-memory systems and co-located virtual machines. Then, we will describe Nahanni memcached, a caching service for cloud computing environments.

Chapter 3

Elastic Phoenix: Malleable MapReduce

The most intuitive advantage of inter-VM shared memory is its ability to provide reduced latency and increased bandwidth to applications. However, shared memory can also be used to add additional functionality to programming frameworks. In cloud computing environments, resources are provisioned in units of virtual machines, meaning that the only way to scale up an application is to add more VM nodes. Programming systems designed for shared-memory systems are hard to scale in such an environment.

One of the major advantages of cloud computing, also available in other environments where resources are shared, is the ability to add and remove resources from a computation as they become available or are needed. In scheduling, jobs with this ability are called *malleable* [34], and in cloud computing, this is often what is meant by *elastic*. Adding resources to an application as it runs can reduce the *response time* of the job. If the application is part of a workload made up of multiple jobs, malleability can improve the response time of the whole workload.

While malleability is desirable, it is difficult for a framework to transparently provide the ability to add and remove resources. There are often application-specific constraints regarding when resources may be added or removed, and it is hard to guarantee consistency of internal state when the resources available change. Because of this, most frameworks that do offer malleability require the programmer to explicitly reason about the joining and leaving of worker processes. Constrained programming models such as MapReduce, however, abstract the management of resources to allow for automatic parallelization. Because the management of worker processes is handled by the framework, malleability is a natural

A version of this chapter has been accepted for publication. Adam Wolfe Gordon and Paul Lu. Elastic Phoenix: Malleable MapReduce for Shared-Memory Systems. In *8th IFIP International Conference on Network and Parallel Computing (NPC 2011)*. Springer, 2011

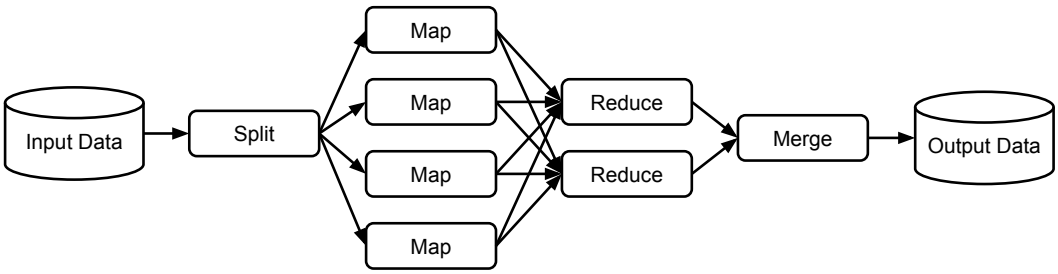


Figure 3.1: Stages of a MapReduce Job

extension for such systems.

The Phoenix [64] MapReduce framework is designed to run in a single process on a shared-memory system, and is thus not well-suited to the cloud environments in which MapReduce has become popular. It also does not support malleable jobs, since the number of threads to use for a computation must be specified when a job is started. One of the most appealing aspects of MapReduce is the separation of the application code from the management of processes and data, making it a natural fit for transparent malleability.

We have used Nahanni shared memory to develop a new framework based on Phoenix, which we call Elastic Phoenix. Elastic Phoenix adds malleability to Phoenix applications, while retaining the programming model and making few changes to the Phoenix API. In addition, Elastic Phoenix allows Phoenix applications to run with worker processes in multiple co-located virtual machines. This makes it suitable for cloud environments where resources are allocated in VM units.

Elastic Phoenix demonstrates how inter-VM shared memory can be used to port a shared-memory programming environment to be suitable for cloud computing, with few changes to the programming model and API.

3.1 Background

3.1.1 The MapReduce Programming Model

MapReduce is a programming framework originated by Google [29]. MapReduce requires programmers to write applications as map and reduce functions, inspired by the functional programming primitives of the same names. The execution of a MapReduce job proceeds in stages, as illustrated in Figure 3.1. In the split stage, a function divides the input data into chunks. The map function processes a chunk of input data into key-value pairs, which are consumed by the reduce function to produce final output data. The map function is parallelized and the input data split among the map tasks. The reduce function may be

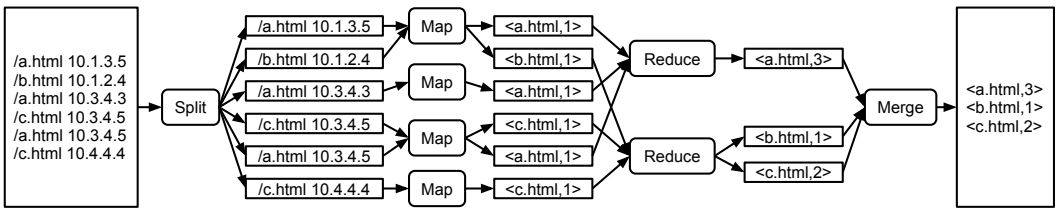


Figure 3.2: A MapReduce Job for Log Analytics

parallelized as well, with each reduce task receiving a portion of the intermediate key-value pairs. In Google's MapReduce implementation, and in the open-source Hadoop [3] implementation, each reduce task produces a separate output file. In other frameworks, such as Phoenix, a parallel merge stage merges the final data into a single set of output.

The MapReduce model is designed to be bandwidth-bound, rather than latency-bound [54]. One consequence of this design choice is that in general we are more interested in the throughput of MapReduce on a given workload than in the latency with which it finishes a particular job. In contrast, web applications using memcached, which we discuss in Chapter 4, are designed for interactive use, where latency is an important metric.

A common use for MapReduce is log analytics: extracting the most popular URLs from server logs. This common task is similar to the first stage of web indexing, as performed by search engines such as Google and Yahoo. Such a job is illustrated in Figure 3.2. The input data is a server log file containing accessed URLs on the server, and who accessed them (e.g. the line "/a.html 10.1.3.5" indicates that a user from IP address 10.1.3.5 visited the page a.html). The input is split into lines by the split phase. The map function extracts a URL (e.g. "a.html") from a line and emits a key-value pair with the URL as the key and 1 as the value (e.g. "<a.html,1>"). MapReduce partitions intermediate data based on its key, meaning that all intermediate key-value pairs with a given key are sent to the same reduce task. The reduce task in this example sums the values for each key, giving a total access count for each URL (e.g. "<a.html,3>"). In Phoenix the output from the reduce stage is merged, as illustrated.

MapReduce is one example of a class of programming models known as data parallel. In data-parallel programming models, computation is parallelized based on splitting the input data into pieces that can be worked on independently, as the map and reduce tasks in MapReduce can. Data-parallel programs scale well because individual tasks are completely independent allowing them to be distributed arbitrarily and run in parallel with no communication overhead.

The programmer may provide a number of other functions that control the execution of a MapReduce job. The partitioner determines which reduce task will process each intermediate key-value pair. Some implementations allow an analogous splitter function for splitting the map input. A combiner function allows intermediate data to be aggregated before it is sent to the reduce tasks. Often, the combiner is the same as the reduce function, when the reduce function is associative and commutative.

In addition to being widely in industry and academia, MapReduce has (as described in Section 2.4) inspired a number of other programming models allowing automatic parallelization including Microsoft Dryad [40], Comet [39], Spark [66], Piccolo [52], and Boom [16].

3.1.2 The Phoenix Framework

Phoenix is a MapReduce framework designed for shared-memory systems. Unlike other MapReduce frameworks, such as Google's MapReduce and Hadoop, Phoenix is not distributed: a Phoenix application runs entirely in one address space on a single host. A Phoenix application is written in C and linked with the Phoenix library into a single executable. A Phoenix job runs in a single process, within which there are multiple worker threads and a single master thread. As in other MapReduce frameworks, execution proceeds in four stages: split, map, reduce, and merge. Input data is usually stored in a file on the local filesystem. The split stage is performed by the master thread, generating map tasks. The map tasks are executed by the worker threads, generating intermediate key-value pairs in memory. This intermediate data is consumed by reduce tasks also executed by the worker threads. Final data emitted by the reduce tasks is merged in multiple rounds by the worker threads. The merged data can be written to a file or processed further by the master thread.

In the log analytics example from Section 3.1.1, the split stage would split a log file into lines. The map stage would extract a URL from each line and emit the intermediate key-value pair with the URL as the key and 1 as the value. The reduce stage would then sum the values for each URL in the intermediate data, emitting a final key-value pair with the URL as the key and the sum as the value. Finally, the merge phase would combine the results of all the reduce tasks, sorting the final output data by key (URL). A subsequent MapReduce job could re-sort the data by value, to obtain a list of the most popular URLs.

The `main` function of a Phoenix application is provided by the application, not by the framework. An example of a Phoenix `main` function is shown in Figure 3.3. The applica-

```
int main(int argc, char *argv[]) {
  final_data_t wc_vals;
  int i, fd, disp_num;
  struct stat finfo;
  char *fname, *disp_num_str, *fdata;

  fname = argv[1];
  disp_num_str = argv[2];

  printf("Wordcount: Running...\n");

  // Read in the file
  CHECK_ERROR((fd = open(fname, O_RDONLY)) < 0);
  // Get the file info (for file length)
  CHECK_ERROR(fstat(fd, &finfo) < 0);
  // Memory map the file
  CHECK_ERROR((fdata = mmap(0, finfo.st_size + 1,
                            PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0)) == NULL);

  // Get the number of results to display
  CHECK_ERROR((disp_num = (disp_num_str == NULL) ?
    DEFAULT_DISP_NUM : atoi(disp_num_str)) <= 0);

  // Setup splitter args
  wc_data_t wc_data;
  wc_data.unit_size = 5; // approx 5 bytes per word
  wc_data.fpos = 0;
  wc_data.flen = finfo.st_size;
  wc_data.fdata = fdata;

  CHECK_ERROR (map_reduce_init ());

  // Setup map reduce args
  map_reduce_args_t map_reduce_args;
  memset(&map_reduce_args, 0, sizeof(map_reduce_args_t));
  map_reduce_args.task_data = &wc_data;
  map_reduce_args.map = wordcount_map;
  map_reduce_args.reduce = wordcount_reduce;
  map_reduce_args.combiner = wordcount_combiner;
  map_reduce_args.splitter = wordcount_splitter;
  map_reduce_args.locator = wordcount_locator;
  map_reduce_args.key_cmp = mystrcmp;
  map_reduce_args.unit_size = wc_data.unit_size;
  map_reduce_args.result = &wc_vals;
  map_reduce_args.data_size = finfo.st_size;

  printf("Wordcount: Calling MapReduce Scheduler Wordcount\n");

  CHECK_ERROR(map_reduce (&map_reduce_args) < 0);

  printf("Wordcount: MapReduce Completed\n");
  dprintf("\nWordcount: Results (first %d):\n", disp_num);
  for (i = 0; i < disp_num && i < wc_vals.length; i++) {
    keyval_t * curr = &((keyval_t *)wc_vals.data)[i];
    dprintf("%15s - %" PRIdPTR "\n", (char *)curr->key, (intptr_t)curr->val);
  }

  free(wc_vals.data);

  CHECK_ERROR(munmap(fdata, finfo.st_size + 1) < 0);
  CHECK_ERROR(close(fd) < 0);

  return 0;
}
```

Figure 3.3: The `main` Function of A Phoenix Application: Word Count

```
void wordcount_map(map_args_t *args) {
  char *curr_start, curr_ltr;
  int state = NOT_IN_WORD;
  int i;

  char *data = (char *)args->data;
  curr_start = data;

  for (i = 0; i < args->length; i++) {
    curr_ltr = toupper(data[i]);
    switch (state) {
    case IN_WORD:
      data[i] = curr_ltr;
      if ((curr_ltr < 'A' || curr_ltr > 'Z') && curr_ltr != '\'') {
        data[i] = 0;
        emit_intermediate(curr_start, (void *)1, &data[i] - curr_start + 1);
        state = NOT_IN_WORD;
      }
      break;

    default:
    case NOT_IN_WORD:
      if (curr_ltr >= 'A' && curr_ltr <= 'Z') {
        curr_start = &data[i];
        data[i] = curr_ltr;
        state = IN_WORD;
      }
      break;
    }
  }

  // Add the last word
  if (state == IN_WORD) {
    data[args->length] = 0;
    emit_intermediate(curr_start, (void *)1, &data[i] - curr_start + 1);
  }
}
```

Figure 3.4: The `map` Function of A Phoenix Application: Word Count

```
void wordcount_reduce(void *key_in, iterator_t *itr) {
    char *key = (char *)key_in;
    void *val;
    intptr_t sum = 0;

    while (iter_next (itr, &val)) {
        sum += (intptr_t)val;
    }

    emit(key, (void *)sum);
}
```

Figure 3.5: The `reduce` Function of A Phoenix Application: Word Count

tion configures a MapReduce job by filling in a data structure (lines 34–45) which is passed to the framework (line 49). The application must provide a map function (e.g. Figure 3.4). Most applications provide a reduce function (e.g. Figure 3.5), although an identity reduce function is provided by the framework. The application can optionally provide a splitter function for dividing input data (the default splitter treats the input data as an uninterpreted byte array and splits it into even parts); a combiner that merges intermediate values emitted by a map thread; a partitioner that divides the intermediate data among reduce tasks; and a locator function that helps the framework assign each input split to an appropriate map thread based on the location of the data in memory. The application must also provide a comparator for the application-defined intermediate data keys.

Phoenix creates a worker thread pool, which is used for map, reduce, and merge tasks during execution of a job. Tasks are placed in a set of queues, one for each worker thread, by the master thread. The worker threads in the pool get work by pulling tasks from their queues. If a worker's queue runs out of tasks, it will steal work from other threads to avoid sitting idle.

Map tasks emit intermediate key-value pairs into a set of arrays. Each map thread has an output array for each reduce task (the number of reduce tasks is defined by the application). A reduce task processes the data produced for it by each map thread, emitting final data into another array of key-value pairs. These final data arrays are merged pairwise until a single, sorted output array is produced.

To avoid confusion, we will henceforth refer to Phoenix as "original Phoenix," in contrast to our Elastic Phoenix framework. We use the name Phoenix on its own only for discussion applicable to both frameworks.

3.2 Related Work

3.2.1 Malleable Programming Environments

Malleable applications are supported by the Message Passing Interface (MPI) through its dynamic process management capability [37]. However, this technique is only applicable to applications developed directly on top of MPI, which offers a low level of abstraction.

At a higher level, support for malleability has been implemented in the Process Checkpointing and Migration (PCM) library, which is built using MPI [32, 33]. PCM is similar to Elastic Phoenix in that it adds malleability to an existing API. However, unlike Elastic Phoenix, the PCM implementation uses explicit splitting and merging of MPI processes;

that is, the application developer is required to reason about malleability in order to support it.

An unrelated project called Phoenix [58] provides a message-passing API that supports malleability by allowing nodes in a cluster to join and leave a computation while it runs. Unlike Elastic Phoenix, it is not intended for shared memory systems.

3.2.2 MapReduce Improvements

Other research has reduced the response time of MapReduce jobs by modifying the framework or semantics of the programming model. The Hadoop Online Prototype (HOP) [26] improves response time of jobs in the Hadoop framework by modifying the flow of data during execution. In HOP, intermediate key-value pairs are sent directly from map workers to reduce workers, before a map task is completed. This reduces the amount of work that must be done between stages, improving parallelism. The framework can also produce partial results by running the reducer on partial map output, though the reducer must be executed over all the map output to produce a final result.

Verma *et al.* [61] present a Hadoop-based MapReduce framework in which reduce tasks not only receive but consume map output as it is produced, eliminating the effective barrier between the map and reduce stages. This technique changes the semantics of the MapReduce model, since the reducer no longer has guarantees about the ordering of its input data. Application developers are thus required to store and sort the data if necessary. Evaluation shows that if careful and efficient techniques are used to handle intermediate data, this modification to MapReduce can improve response time for certain classes of problems.

Chohan *et al.* [25] explore the advantages of using Amazon EC2 Spot Instances (SIs), inexpensive but transient virtual machines, to accelerate Hadoop jobs. This is similar to our work in that it provides a type of malleability to a MapReduce framework: a job can be accelerated by adding worker nodes in SIs. However, because of the nature of Hadoop, termination of a spot instance is expensive, since completed work often needs to be re-executed. Elastic Phoenix does not encounter this expense in reducing the number of workers, since the data produced by all completed tasks is in shared memory and will not be lost. We note, though, that our current implementation is not suitable for the semantics of SIs, in which the worker node is terminated without warning, since an Elastic Phoenix worker requires a signal and a little bit of extra time to clean up when it is removed.

The Intermediate Storage System (ISS) [42] for Hadoop improves the response time of MapReduce jobs when faults occur. Hadoop normally stores map output data on local disk

until it is required by a reduce worker, requiring the map task that produced the data to be re-executed if a fault occurs on the worker node. The ISS asynchronously replicates the intermediate data, saving the re-execution time if only one replica incurs a fault. However, this technique induces overhead in the absence of faults.

3.3 Design Goals

Our design goals for Elastic Phoenix were:

1. **Elasticity.** The user should be able to add worker threads to a running MapReduce job in order to accelerate it. The framework should not make assumptions about the stage of computation during which workers will be added.

2. **Design Flexibility.** Elastic Phoenix applications should be able to run either with workers running on a shared-memory system using POSIX shared memory, or with workers running in multiple virtual machines using Nahanni shared memory. Therefore, the design should assume as little as possible about the underlying data sharing mechanism.

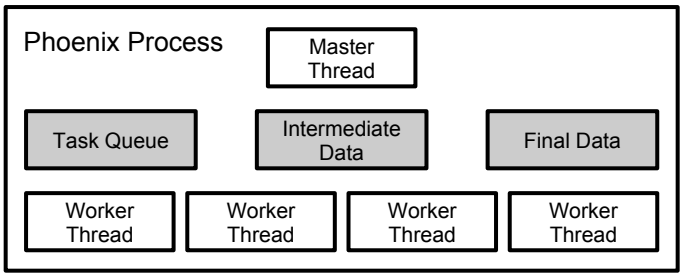
3. **API Compatibility.** It should take minimal effort to make an original Phoenix application work with Elastic Phoenix.

These design goals are sometimes conflicting. In particular, maintaining the API of the original Phoenix framework, which was not designed for elasticity, made it more difficult to achieve the other goals.

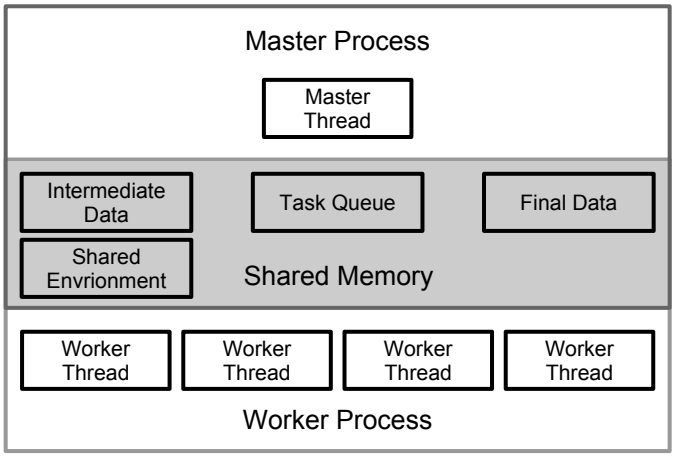
3.4 Implementation

The basic design change that enables Elastic Phoenix is separation of the master thread and worker threads into different processes, as illustrated in Figure 3.6. This presents a number of challenges, since original Phoenix was designed with a shared address space in mind. In order to support Nahanni shared memory, our implementation does not assume that all worker processes share a kernel or filesystem. These assumptions would also make it easier to port Elastic Phoenix to a distributed shared memory or message passing system.

The shared memory region in Figure 3.6(b) can be either a POSIX shared memory region (when running a job natively on a host) or a Nahanni shared memory region (when running a job in VMs).



(a)



(b)

Figure 3.6: High-Level Architecture of (a) Original Phoenix and (b) Elastic Phoenix

3.4.1 Sharing Data

One challenge in designing Elastic Phoenix was how to get input data to the worker threads. The master thread splits the input data, but the worker threads then need to access this data for the map phase. A common idiom in original Phoenix applications is to `mmap` an input file, then have the splitter calculate offsets into the mapped file. Since Elastic Phoenix threads do not share an address space, this clearly will not work. In the interest of flexibility, we also assume that the worker processes cannot access the input file, as may be the case when running Elastic Phoenix in a set of VMs.

Our initial design, which made no changes to the original Phoenix API, required splitter functions to return pointers to the actual input data. The framework then copied this data into shared memory. This was a simple requirement, but one that did not exist in original Phoenix; some sample applications did not meet it, returning from their splitters pointers to structures describing the input data. This design allowed applications to continue using the `mmap` method described above, with most applications requiring no changes to the splitter code.

However, copying the input data into shared memory after it has been split is a performance bottleneck. Most application-provided splitters must examine at least some portion of the data in order to generate sane splits, so with copying the data ends up being read in from disk during the split, then copied into the shared memory region, possibly being read off disk again if the buffer cache has been recycled.

Our final design changes the API for splitter functions by adding an argument: a structure containing pointers to memory allocation functions, which we require the splitter use to allocate space for the input data. The functions we pass to the splitter are custom memory allocation functions that allocate space in the shared memory region, keeping all allocation metadata in the region itself. The common design for splitters using this API is to allocate memory for a split, read data directly into this memory, then reduce the size of the split as necessary to reflect application-specific restrictions on the split data.

In addition to the input data, the master and the workers must share the intermediate and final data arrays. The intermediate data is straightforward to move into the shared memory region, using the previously mentioned memory allocators to dynamically allocate the arrays. The only complication is that there is a set of intermediate data arrays for each worker thread, and in Elastic Phoenix the total number of worker threads is not known at initialization time. We deal with this by setting an upper bound on the number of map

threads and pre-allocating space for the necessary pointers.

There is one additional issue with the final data array: it is returned to the application when the MapReduce job finishes. This causes a problem because many applications `free` this array when they are done processing the results it contains. This works in original Phoenix because the framework allocates the final data array with `malloc`. In Elastic Phoenix this array resides in the shared memory region and is allocated with our custom allocator. We deal with this by adding a function to the API: `map_reduce_cleanup`. The cleanup function mirrors the framework's `map_reduce_init` function, and is called by the application when it is finished with the results of the MapReduce job.

In order to retain the structure of the pointer-based data structures used throughout original Phoenix, we map the shared memory to the same address in all processes. This avoids the need for pointer swizzling or other tricks, even when running worker processes in separate virtual machines.

3.4.2 Assigning Tasks

In original Phoenix, there is one task queue for each worker thread. The framework does allow threads to steal work from other queues when their own queue runs out, so pre-allocating a task queue for each of the maximum number of worker threads is one potential solution. However, given that Elastic Phoenix will rarely have the maximum number of worker threads, this would lead to the threads exhausting their own queues, then all sharing the other queues until they are exhausted. Given this, we opted for a simpler design with a single task queue. The task queue is pre-allocated in shared memory with a maximum number of tasks, and all worker threads take work from the queue. Since the task queue in original Phoenix was already thread-safe for dequeue operations, we re-used nearly all of its code.

The pre-allocation design causes an issue for the splitter: there is a maximum number of map tasks, so the splitter can only create so many splits. Elastic Phoenix uses the application-provided unit size estimate to request an appropriate number of data units for each split such that the task queue is filled but not exceeded. However, an application's estimated unit size will, in some cases, be inaccurate because the data is not uniform and not known beforehand. In this case, the framework may run out of space for map tasks before all the data has been split. We deal with this situation by requiring that splitters be "resettable." When the splitter is passed a special argument, it should thereafter operate as if it has not yet been called. When we run out of space for map tasks, we reset the split-

ter and re-split the data, requesting more units for each task. This is a rare case, as many applications provide good unit size estimates. We have observed that when re-splitting is required it often takes only one try, which tends to be very fast due to filesystem caching.

3.4.3 Coordinating Workers

An Elastic Phoenix job is started by first running the master process, then running one or more worker processes. Coordinating the worker threads is somewhat different from original Phoenix, since the master does not create the worker threads.

We use two basic mechanisms for coordination in Elastic Phoenix: a worker counter and a barrier, both stored in shared memory. When the master starts, it executes the split phase, then waits for workers to join the computation. The first thing a worker process does when it starts is acquire a worker ID using the counter in shared memory. This is done using atomic operations to avoid the need for a lock.

Because the workers may run in separate virtual machines, and thus do not necessarily share a kernel, we cannot use a blocking barrier for synchronization between computation phases. Instead, we have implemented a barrier using spinlocks and polling. The barrier is used after each stage to synchronize the master and worker threads. The master thread reaches the barrier first, since it does no work in the map, reduce, and merge phases. To make the barrier code simpler, we give each worker process a constant number of threads; in our current implementation the default is four. In addition, we use a shared spinlock to prevent new workers from joining the computation at inconvenient times, such as after the first worker thread reaches the barrier (since this would change the number of threads expected, complicating the barrier code) or before the master has finished setting up the necessary shared environment.

Elastic Phoenix keeps track of the current phase of execution using a flag in shared memory. This allows workers to join during any stage of execution: split, map, reduce, or merge. A worker that joins during a later phase skips the previous phases entirely.

An additional coordination problem in Elastic Phoenix is that because we have attempted to preserve the original Phoenix API as much as possible, the application code is not intended to be executed in multiple processes. Many original Phoenix applications contain code that should be executed only once as part of initialization or cleanup. We add two optional application-provided functions to the API to deal with this: `prep` and `cleanup`. If an application provides these functions, Elastic Phoenix guarantees that they will be executed only in the master process. The `prep` function is called before any other

Description	Original Phoenix	Elastic Phoenix
Splitter function signature	`int splitter(void *, int, map_args_t *)`	`int splitter(void *, int, map_args_t *, splitter_mem_ops_t *)`

Splitter uses provided allocator	`free(out->data);`
	`out->data =`
	`  malloc(out->length);`
		`mem->free(out->data);`
		`out->data =`
		`  mem->alloc(out->length);`

Splitter can be reset	N/A	`if(req_units < 0) {`
		`  lseek(data->fd, 0, SEEK_SET);`
		`  data->fpos = 0;`
		`}`

New `cleanup` API function	`printf("Num. values: %d\n",`
	`  mr_args.result->length);`
	`return(0);`
		`printf("Number of values: %d\n",`
		`  mr_args.result->length);`
		`map_reduce_cleanup(&mr_args);`
		`return(0);`

New application `prep` function	`int main(int argc,`
	`  char **argv) {`
	`...`
	`  struct stat s;`
	`  stat("in", &s);`
	`  data.fd = open("in");`
	`  mr_args.data_size =`
	`  s.st_size;`
	`...`
		`int prep(void *data,`
		`  map_reduce_args_t *args) {`
		`  struct stat s;`
		`  stat("in", &s);`
		`  ((app_data_t*)data)->fd =`
		`  open("in");`
		`  args->data_size =`
		`  s.st_size;`
		`}`

New application `cleanup` function	`int main(int argc,`
	`  char **argv) {`
	`...`
	`  close(data.fd);`
	`...`
		`int cleanup(void *data) {`
		`  app_data_t *ad =`
		`  (app_data_t *)data;`
		`  close(ad->fd);`
		`}`

API `init` modifies `argc` and `argv`	`int nargs = argc;`
	`char *fname = argv[1];`
	`map_reduce_init();`
		`map_reduce_init(&argc, &argv);`
		`int nargs = argc;`
		`char *fname = argv[1];`

Table 3.1: API Changes From Original Phoenix to Elastic Phoenix

work is done, and can be used to, for example, open the input file and set up data for the splitter. The `cleanup` function is called by the `map_reduce_cleanup` API function described earlier. These functions reduce the need for applications to explicitly manage concurrency, and allow our implementation to handle initialization and cleanup differently in future versions.

3.5 Porting Phoenix Applications

One of our design goals, as described in Section 3.3, was to allow original Phoenix applications to be easily ported to Elastic Phoenix. In this section, we describe our experiences porting applications, and try to quantify the amount of change required. Table 3.1 lists the complete set of API changes.

The biggest change that is required in every application is to the splitter function. Since Elastic Phoenix requires that the splitter use a framework-provided memory allocator for allocating input splits, it has an additional argument. In some splitters, this is simply a

Application	Original	Elastic	Lines altered
Histogram	245	285	110
Word Count	234	202	114
Linear Regression	230	243	97
Matrix Multiply	168	185	48

Table 3.2: Porting MapReduce Applications: Original to Elastic Phoenix (lines of code)

matter of replacing calls to `malloc` with calls to the provided allocator, but most need to be restructured to read in a file incrementally instead of using `mmap`, as described in Section 3.4.1.

In Phoenix applications, most of the code in `main` serves one of four purposes: setting up the splitter; setting up the MapReduce job; processing the results; and cleaning up. The first can usually be moved verbatim into the `prep` function. The second is usually idempotent, operating only on local data structures, and can remain intact without harm. The third and fourth can usually be moved to the `cleanup` function to ensure they execute only in the master process.

The original Phoenix framework includes seven sample applications: histogram, k-means, linear regression, matrix multiply, PCA, string match, and word count. Of these, we have ported histogram, word count, and linear regression. PCA and k-means require multiple MapReduce jobs, a feature not currently supported in Elastic Phoenix. The matrix multiply included in original Phoenix is not written in the MapReduce style: the map tasks calculate the resultant matrix directly into a known output location, rather than emitting the calculated values as intermediate data. We wrote a new matrix multiply for original Phoenix that uses the MapReduce style, emitting a key-value pair for each entry in the resultant matrix, then ported it to Elastic Phoenix. String match also does not emit any intermediate or final data; in fact, it produces no results at all.

To quantify the amount of work required to convert an application to Elastic Phoenix, we analyzed the applications we converted. First, we calculated the lines of code for each application before and after conversion using SLOCCount [13]. Then we estimated how many lines were altered by taking a `diff` of the two source files and counting the number of altered lines in the Elastic Phoenix version. This does not entirely account for the fact that many of the changes simply involved moving code from one function to another. The results of this analysis are displayed in Table 3.2.

Word count was the first sample application we ported, and we modified it incrementally during the development of Elastic Phoenix. Therefore, it underwent more change than

```c
int main(int argc, char *argv[]) {
  int i;
  wc_data_t wc_data;

  i = map_reduce_init (&argc, &argv);
  CHECK_ERROR(i < 0);

  wc_data.fname = argv[1];

  printf("Wordcount: Running...\n");

  // Setup splitter args
  wc_data.unit_size = 5; // approx 5 bytes per word
  wc_data.fpos = 0;

  // Setup map reduce args
  map_reduce_args_t map_reduce_args;
  memset(&map_reduce_args, 0, sizeof(map_reduce_args_t));
  map_reduce_args.task_data = &wc_data;
  map_reduce_args.task_data_size = sizeof(wc_data_t);

  map_reduce_args.prep = wc_prep;
  map_reduce_args.cleanup = wc_cleanup;
  map_reduce_args.map = wordcount_map;
  map_reduce_args.reduce = wordcount_reduce;
  map_reduce_args.combiner = wordcount_combiner;
  map_reduce_args.splitter = wordcount_splitter;
  map_reduce_args.key_cmp = mystrcmp;

  map_reduce_args.unit_size = wc_data.unit_size;
  map_reduce_args.partition = NULL; // use default
  map_reduce_args.result = &wc_data.wc_vals;

  printf("Wordcount: Calling MapReduce Scheduler Wordcount\n");

  CHECK_ERROR(map_reduce (&map_reduce_args) < 0);

  map_reduce_cleanup(&map_reduce_args);
  CHECK_ERROR (map_reduce_finalize ());

  return 0;
}
```

Figure 3.7: The `main` Function of An Elastic Phoenix Application: Word Count

```c
int wc_prep(void *data_in, map_reduce_args_t *args) {
  struct stat finfo;
  wc_data_t *data = (wc_data_t *)data_in;

  // Read in the file
  CHECK_ERROR((data->fd = open(data->fname, O_RDONLY)) < 0);
  // Get the file info (for file length)
  CHECK_ERROR(fstat(data->fd, &finfo) < 0);
  args->data_size = finfo.st_size;
  data->fsize = finfo.st_size;

  return(0);
}
```

Figure 3.8: The `prep` Function of An Elastic Phoenix Application: Word Count

```
int wc_cleanup(void *data_in) {
  wc_data_t *data = (wc_data_t *)data_in;
  int i;

  qsort(data->wc_vals.data, data->wc_vals.length, sizeof(keyval_t), mykeyvalcmp);

  dprintf("\nWordcount: Results (TOP %d):\n", DEFAULT_DISP_NUM);
  for (i = 0; i < DEFAULT_DISP_NUM && i < data->wc_vals.length; i++) {
    keyval_t * curr = &((keyval_t *)data->wc_vals.data)[i];
    dprintf("%15s - %" PRIdPTR "\n", (char *)curr->key, (intptr_t)curr->val);
  }

  return(close(data->fd));
}
```

Figure 3.9: The `cleanup` Function of An Elastic Phoenix Application: Word Count

would an application being ported directly to the final version of Elastic Phoenix. For comparison with Figure 3.3, we present the `main` function for the Elastic Phoenix version of word count in Figure 3.7. Note that much of the code that has been removed from `main` was moved into the `prep` and `cleanup` functions, displayed in Figures 3.8 and 3.9, respectively.

We ported histogram and linear regression to one intermediate version of Elastic Phoenix, then to the final version, so they display less change than word count. We developed matrix multiply for original Phoenix, then ported it directly to the final version of Elastic Phoenix. It therefore provides the best indication of the effort needed to directly port an application. Anecdotally, a single developer wrote the new matrix multiply application for original Phoenix in several hours, and took only a few minutes to port it to Elastic Phoenix.

3.6 Evaluation

We evaluated Elastic Phoenix in two ways. First, we evaluated the overhead of using Elastic Phoenix by running our ported applications with both frameworks. Then, we explored the advantages of Elastic Phoenix by developing a workload of repeated MapReduce jobs on a system where the availability of processors varies over time. We performed these experiments running the master and worker processes directly on a server, as well as in virtual machines. All experiments were performed on an Intel Xeon X5550 (2.67 GHz) server with two sockets, eight cores (16 hyperthreads), and 48 GB of RAM. We compiled original Phoenix with its default tuning parameters, and used the default four worker threads per process for Elastic Phoenix. For the VM experiments, we used three VMs, each with 4 GB of RAM and four CPUs. We ran at most one worker process in each VM, with a worker sharing a VM with the master process when necessary. Note that we ran only up to

Application	Description	Input Size
Histogram	Counts the frequency of each colour value over pixels in a bitmap image	1.4 GB
Word Count	Counts the frequency of each word in a text file	10 MB
Linear Regression	Generates a linear approximation of a set of points	500 MB
Matrix Multiply	Parallel matrix multiplication using row and column blocks	1000x1000

Table 3.3: Applications and Inputs for Evaluation

12 total threads in the VMs; this was due to time constraints, not any technical limitations of Elastic Phoenix.

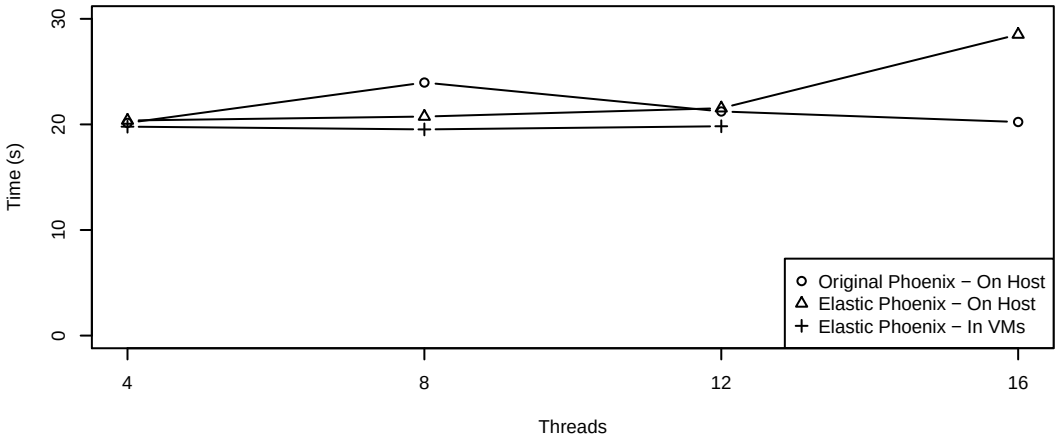
3.6.1 Overhead

The performance advantage of malleability is that MapReduce jobs can be accelerated as they run by adding threads to the computation. If the overhead of Elastic Phoenix is so large that, for example, using four threads in original Phoenix is faster than using eight threads in Elastic Phoenix, then this advantage disappears. In this section we evaluate the overhead of using Elastic Phoenix, showing that for some applications it is possible to increase performance by adding threads.

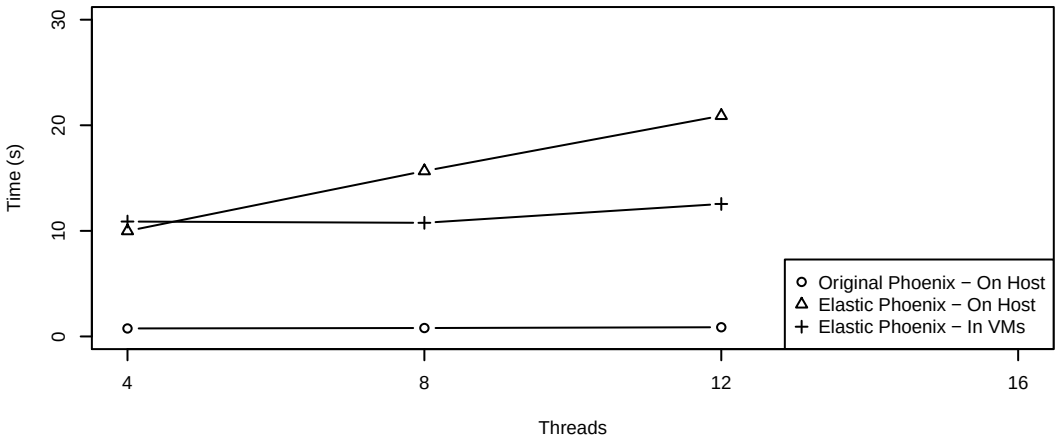
To evaluate the overhead of our design, we ran four applications under both original and Elastic Phoenix. For each application, we used the largest input data we could given 4 GB of shared memory. The applications we used and their input sizes are shown in Table 3.3. All times are averages over five runs, with cold filesystem caches.

We divide these applications into two classes, which we will discuss separately: I/O-bound and CPU-bound. Histogram, word count, and linear regression are I/O-bound. This is due to the fact that the computation performed in the map and reduce stages is quite trivial, even with large input. The bottleneck, therefore, is reading and splitting the input data, which must be done serially in both original and Elastic Phoenix. Our matrix multiply application is CPU-bound because its input is generated in the splitter function, avoiding any file I/O.

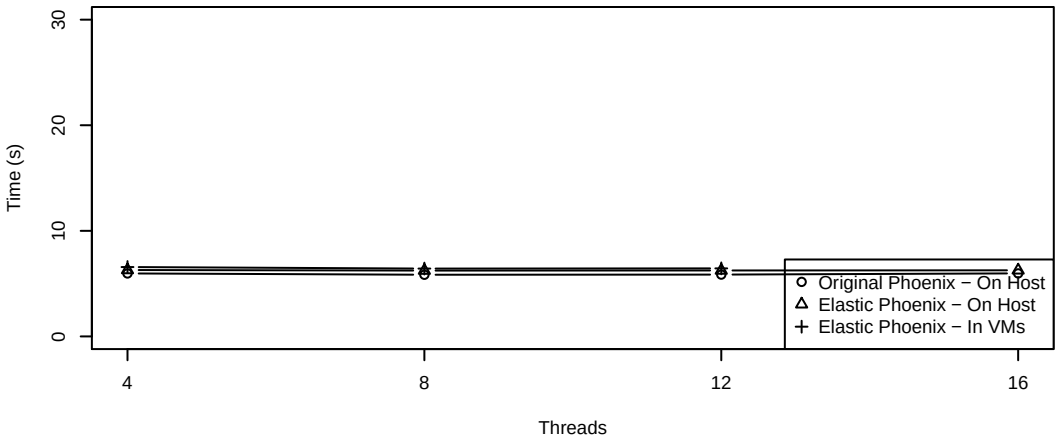
Benchmark results for the I/O-bound applications are presented in Figure 3.10. Being I/O-bound, we do not expect any speedup as we add threads. We observe that Elastic Phoenix has little overhead compared to original Phoenix for the histogram and linear regression applications (8% and 6%, respectively, on average), showing some slowdown as



(a) Histogram



(b) Word Count



(c) Linear Regression

Figure 3.10: Overhead Experiment Results for I/O-bound Applications

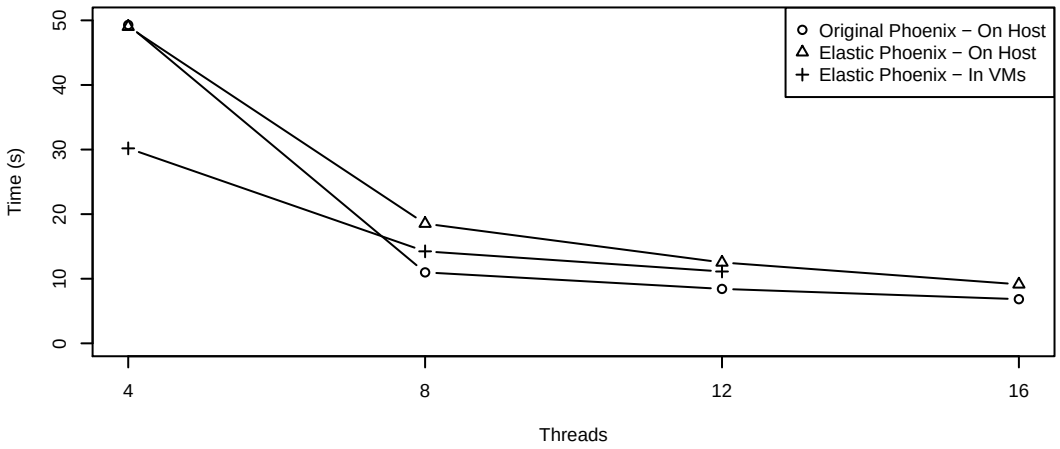


Figure 3.11: Overhead Experiment Results for the CPU-Bound Matrix Multiply Application

we add more threads due to the additional coordination overhead. These applications have large input data, and generate intermediate data with a fixed number of keys.

In contrast, we observe significant overhead (1,792% average) in the word count application. This overhead likely stems from the fact that the input size is small, and that the application produces a large amount of intermediate data with very diverse keys, since many more words appear infrequently in the input than appear frequently. This causes a large amount of extra shared memory to be allocated but never used, as the framework allocates space for multiple values (10, by default) with the same key when the first value for a given key is emitted. In fact, word count, with only 10 MB of input data, can be run with at most 12 worker threads in Elastic Phoenix; with more threads, the small amount of additional per-thread allocation in shared memory causes it to exceed the 4 GB limit. Our shared memory allocator is simplistic, and does not match the efficiency of the system's `malloc`, which is used in original Phoenix.

We observe that the benchmark results for I/O-bound applications run in virtual machines are consistent with those run on the host. In some cases, the applications are faster in virtual machines, which we suspect is due to scheduling effects. The four Elastic Phoenix worker threads are the only active processes in each VM, and each VM has four CPU cores, potentially allowing the kernel to make better scheduling decisions than on the host machine. An in-depth analysis of VM scheduling interactions is beyond the scope of this thesis.

Benchmark results for our CPU-bound application, matrix multiply, are shown in Figure 3.11. Here we observe significant speedup from four to eight threads, then moderate

speedup as we increase to 12 and 16 threads. We theorize that the extreme speedup from four to eight threads is caused by the fact that our CPUs are quad-core, and with eight threads Phoenix uses both sockets. Original Phoenix pins threads to cores, spreading work out as much as possible, but we have configured it to use the same number of processors as threads, so the four threads are pinned onto one CPU. Using both sockets doubles the total usable cache size and the total memory bandwidth available. The decline in speedup beyond eight threads is likely due to the fact that we have only eight real CPU cores, and the hyperthreaded cores offer little advantage in an application such as matrix multiply with regular memory access patterns, few branches, and no disk I/O.

With four threads (a single worker process), Elastic Phoenix adds no overhead compared to original Phoenix. Elastic Phoenix adds a moderate amount of overhead when more than one process is used, due to the added overhead of coordinating threads in multiple processes, such as increased contention on spinlocks. Overall, the overhead averages 38%. In virtual machines, we again observe results similar to running on the host. The surprising performance win for VMs with four worker threads might again be due to the scheduling effects described above.

The important result from our overhead experiments is that Elastic Phoenix with, for example, eight threads can be significantly faster than original Phoenix with, for example, four threads, suggesting that the ability to add additional workers as a job runs can improve performance in some cases.

3.6.2 Elasticity Advantages

Elastic Phoenix can improve throughput by allowing idle processors to be used immediately by a running MapReduce job. A performance improvement is possible when the overhead of using Elastic Phoenix does not exceed the speedup provided by the additional threads. As we showed earlier, this is the case for some applications. In this section, we evaluate how much performance can be gained in such a situation using a synthetic benchmark. For this benchmark, job latency can be reduced by 29% by adding threads as processors become available.

Consider a scenario in which a sequence of MapReduce jobs is run repeatedly, starting again each time it finishes. For example, the sequence may consist of a job that creates a link graph from a set of hypertext pages, then a job that computes the most popular page. Assume the jobs run on a system where CPU resources become available unpredictably, as is typical in a multi-user, batch-scheduled environment. Original Phoenix always uses the

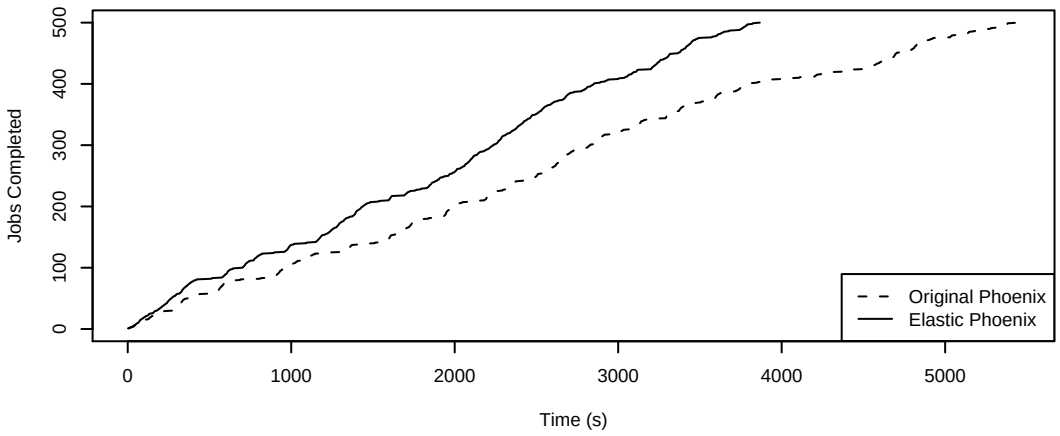


Figure 3.12: Task Completion Over Time in the Elasticity Benchmark

number of CPUs that are available when it starts, while in Elastic Phoenix CPUs are added to the computation as they become available.

Our benchmark uses alternating linear regression and matrix multiply jobs with random-sized inputs of 250 MB to 500 MB for linear regression and 500x500 to 1000x1000 for matrix multiply. This sequence represents a typical workload of an I/O-bound job that generates input for a compute-bound job, although we do not actually use the linear regression output as input for the matrix multiply. With one worker thread, the jobs take between 1 second and 5 minutes each. For this experiment, we compiled Elastic Phoenix with one thread per worker process, so that it can take advantage of a single processor becoming available.

We generated a set of random CPU availability changes, where every 5 seconds the number of CPUs available can change, with at least 2 and at most 8 CPUs always being available, since our host has 8 CPU cores (we do not employ the 8 hyperthreaded cores). The workload we generated contains 500 jobs in total: 250 linear regression jobs interleaved with 250 matrix multiply jobs. We executed the benchmark only on the host, not in VMs. However, we observe that running linear regression and matrix multiply in VMs (Figures 3.10(c) and 3.11) has little to no overhead compared to running them on the host machine, and also that in VMs running the applications with eight threads is faster than running them with four threads. Therefore, we would expect similar results from running our elasticity benchmark in VMs.

The results of the experiment are displayed in Figure 3.12. Original Phoenix completed the jobs in 5,451 seconds and Elastic Phoenix completed the jobs in 3,866 seconds, a 29% improvement. We see from these results that adding processors as they become available

can drastically speed up a workload of this type.

3.7 Limitations of Elastic Phoenix

The design of Elastic Phoenix introduces a number of limitations not present in original Phoenix. Some of these are inherent to the design we have chosen; others are due to our current implementation and could be eliminated in the future without a significant design change. There are also some ways in which Elastic Phoenix could be improved, orthogonal to these limitations.

3.7.1 Job Size

The size of the shared memory region used with Elastic Phoenix limits the amount of input data that can be used. This is because the input data must all fit in shared memory, and will share the space with some of the intermediate data (input data is freed immediately after being used by a map task, but intermediate values are stored to shared memory as they are produced). Similarly, the intermediate data must share the space with some final data, so this data is also limited in size.

The input size limit is not the same for all applications and cannot be calculated in advance. As mentioned in Section 3.6.1, the number of intermediate key-value pairs, as well as the diversity of the keys, affects the maximum input size. This is because Elastic Phoenix allocates space for multiple values for each intermediate data key, since in most applications keys are repeated. In an application with very diverse intermediate keys, much of this allocated space goes unused as many keys will have very few values. An obvious way to allow more intermediate data in applications with high key diversity would be to allocate space for fewer values by default – perhaps only a single value. Unfortunately, this involves a tradeoff, and finding an optimal solution would be quite difficult.

Our shared memory allocator works at the granularity of fixed-size blocks. A key and a small number of values do not use much of a block, but from the allocator's perspective the entire block becomes unavailable. Smaller blocks would allow for less fragmentation, but would make allocation slower and require the allocator to use more of the shared memory for accounting purposes. The optimal block size and number of values allocated depends on the application, but it may be possible to classify different types of applications and provide default values appropriate for each class. The most desirable solution would be to develop a better memory allocator.

The task queue in Elastic Phoenix is of a fixed size (32,770 tasks by default). Thus, a limited number of input splits can be produced by the splitter. This means that even with small input data, an application which must split its input into many units cannot run under Elastic Phoenix. As described in Section 3.4.1, we re-split data if it fills the job queue, requesting more units from the splitter. If an application ignores the number of requested units in its splitter, then porting it to Elastic Phoenix will require modifying the splitter to honor this argument.

3.7.2 Combiners

The original Phoenix framework allows applications to provide a combiner function, which performs a partial reduce on the map output of a single worker thread. This can speed up the reduce stage, and reduce the amount of space used for intermediate data. In particular, it helps applications that have a large number of repeated intermediate keys.

Elastic Phoenix currently does not support combiners. The application may specify a combiner, to keep the API consistent with original Phoenix, but it will not be used by the framework. This is a limitation of the current implementation and could be changed in the future. It is possible that this modification would help with the job size limitation by compacting intermediate data.

3.7.3 Multiple MapReduce Jobs

A common idiom in MapReduce applications, including Phoenix applications, is to run multiple MapReduce jobs in sequence. For example, one might run a job to do a large computation, then another job to sort the results by value. In original Phoenix, multiple jobs can be performed without writing data to disk: the output data from one job becomes the input data to another job. That is, multiple calls are made to the Phoenix MapReduce scheduler within a single executable application.

The current Elastic Phoenix implementation does not support running multiple MapReduce jobs in one execution, due to the way in which Elastic Phoenix initializes data that will be shared among the master and worker processes. This limitation could be eliminated through some further design.

3.7.4 Number of Workers

Elastic Phoenix pre-allocates a number of data structures that in original Phoenix are allocated based on the number of threads being used. In particular, there is one set of interme-

diate data queues and one final data queue for each worker thread, and the pointers for these are pre-allocated. They could be allocated dynamically as workers join the computation, but this would require using a linked list or similar structure, making lookups slower when inserting data.

In order to pre-allocate this data, we set a fixed limit on the number of worker tasks that can join a MapReduce job. The default in our current implementation is 32 threads, which allows for eight worker processes with four threads each. On any given system, the number of worker threads is naturally limited by the number of processors, so this is a minor limitation.

3.8 Shared Memory Hadoop: A Cautionary Note

In this chapter, we have described our Elastic Phoenix system, which uses inter-VM shared memory to add features to the existing Phoenix MapReduce framework. However, Phoenix was not the first MapReduce environment we tried using with inter-VM shared memory, and adding features was not our original goal. We first tried using inter-VM shared memory to improve the performance of Hadoop. Here we describe the work we did on Hadoop, to show that a system using inter-VM shared memory must be designed carefully to gain significant performance benefits.

Hadoop is a popular distributed MapReduce framework written in Java. Its architecture closely follows that of the Google MapReduce system. Input and output data are typically stored in HDFS, a distributed file system modeled after GFS [35].

Like any MapReduce framework, Hadoop must transfer intermediate data produced by map tasks to reduce tasks. The map tasks and the reduce tasks are, in a distributed MapReduce system, not guaranteed to run on the same host. This presents something of a bottleneck, since this shuffle step must be completed before the reduce computation can begin. In a cloud environment, where some virtual machines used as Hadoop nodes may be located on the same physical host, it is possible to avoid the network for intermediate data transfer. We modified Hadoop to use Nahanni shared memory for transferring intermediate data.

The general lesson from our experience with Hadoop is that using inter-VM shared memory to speed up only a single component of a complex application will not yield significant performance improvements. A specific lesson from Hadoop is that it is not effective to use inter-VM shared memory as a circular buffer: the overhead of copying data in and

Figure 3.13: Dataflow of a Hadoop MapReduce Job

out of shared memory is almost as large as copying data through the network stack. That is, inter-VM shared memory is more effective when it is used to store pointer-based data structures, as in Elastic Phoenix.

3.8.1 Hadoop Dataflow

Hadoop executes map and reduce tasks on worker nodes. Each worker node is allocated a certain number of map slots and a certain number of reduce slots, and thus may be assigned multiple map tasks and multiple reduce tasks. Key-value pairs emitted by a map task are initially stored in memory, but are flushed to local disk on completion or as needed.

Each reduce task has three phases: shuffle, sort, and reduce. In the shuffle phase, intermediate values are fetched from each map host using HTTP and stored to local disk. These key-value pairs are then sorted by key, using an external sort if necessary. Once sorted, they are used as input for the reduce function. The dataflow for a Hadoop MapReduce job is shown in Figure 3.13. Hadoop attempts to place tasks such that some or all of a reduce task's input data is available on a local file system. This reduces the amount of data transferred over the network during the shuffle phase. The receiver attempts to keep all the data it fetches during the shuffle phase in memory, spilling to local disk if necessary.

3.8.2 Shared Memory Streaming

Because Hadoop transfers intermediate values over the network, we first built a streaming interface, using the shared memory region as a circular buffer. A reduce task in the shuffle phase determines for each map output it needs to fetch whether the map task ran on the same physical host as the reducer. Currently, this is done using a list of hostnames maintained in shared memory. If the host is found in the list, then the map output can be fetched using shared memory. The dataflow for a Hadoop MapReduce job using shared memory streaming is shown in Figure 3.14.

To initiate a file transfer, the client first finds a free block and writes its index into a designated location in shared memory. It writes an indicator of what it would like to fetch into a second designated location, then uses a Nahanni-provided semaphore to signal the

Figure 3.14: Dataflow of a Hadoop MapReduce Job with Shared Memory Streaming

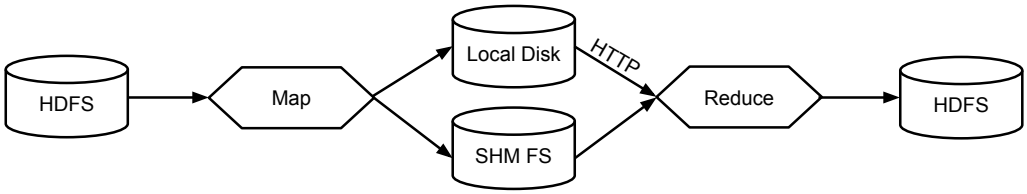


Figure 3.15: Dataflow of a Hadoop MapReduce Job with Shared Memory Filesystem

server. Upon receiving the signal, the server begins writing the requested data into the memory chunks.

3.8.3 Shared Memory Filesystem

Streaming intermediate data through shared memory still requires copying data into and out of shared memory in order to transfer it. This is a performance bottleneck that reduces the performance of the shared memory streaming to near that of virtualized networking. To reduce this performance effect, we implemented a second design: a shared memory filesystem. This comprises a simple filesystem residing in shared memory, and a Java interface to it conforming to Hadoop's filesystem interface. This is much simpler than a full POSIX filesystem implementation, since the Hadoop interface contains few operations.

The dataflow for this design is shown in Figure 3.15. In this design, a map task attempts to write its output to the shared memory filesystem. If the filesystem is full then data is written to local disk as in standard Hadoop. The reduce task looks for its input data in the shared memory filesystem, then requests it from the map host if it is not found. This fallback uses HTTP to transfer, as in the standard Hadoop dataflow.

3.8.4 Performance

To isolate the performance of the shuffle phase, we wrote a Hadoop MapReduce application whose map phase reads a single byte of input data and produces a set amount of intermediate data, and whose reduce phase reads the intermediate data and produces a single byte of output data. We ran the Hadoop MapReduce framework on a virtualized cluster in which the master node is the physical host and the worker nodes are three virtual machines. The

Hadoop Distributed Filesystem (HDFS) used to store input and output data was configured similarly, with the master on the host and the workers in VMs. A 1 GB Nahanni shared memory region was used, shared among all the VMs.

The results of our microbenchmark are displayed in Figure 3.16. With small intermediate data (512 MB), neither of our shared memory optimizations provides a performance improvement. With larger intermediate data sizes, shared memory streaming provides a small benefit. This is likely because network contention is not high enough for shared memory to make a difference in the small data case. The improvement is about 15% with 2 GB of intermediate data.

The shared memory filesystem optimization displays the same performance as stock Hadoop with small intermediate data, and lower performance with large intermediate data. There are multiple reasons for this. For one, the local disk used for intermediate data in our stock Hadoop setup is tmpfs, an in-memory filesystem that takes advantage of the Linux kernel's buffer cache. This in-memory filesystem may be faster than our simple shmfs. Because our shmfs resides entirely in Nahanni shared memory, it must use spinlocks to synchronize write accesses. With large intermediate data, there are more writes, increasing contention for these spinlocks and reducing performance. Hadoop still copies the data into memory from the shared memory region, so while copying in and out of the network stack is avoided, there is still copying.

3.8.5 Lessons from Hadoop

One lesson from our experience with Hadoop is that speeding up a small component (i.e. intermediate data transfer) of a complex application is unlikely to provide significant performance improvements. When applying optimization such as inter-VM shared memory, it is important that the optimization be applicable throughout the computation, removing a bottleneck end-to-end. In contrast, Nahanni memcached, which we discuss in Chapter 4, uses inter-VM shared memory for the common case of read operations, optimizing a large portion of the application's operations.

Our experience with Hadoop shows that frameworks using inter-VM shared memory must be carefully designed to maximize the benefit shared memory can provide. A specific lesson is that marshaling and unmarshaling data is a significant overhead that can negate any advantage of shared memory compared to virtualized networks.

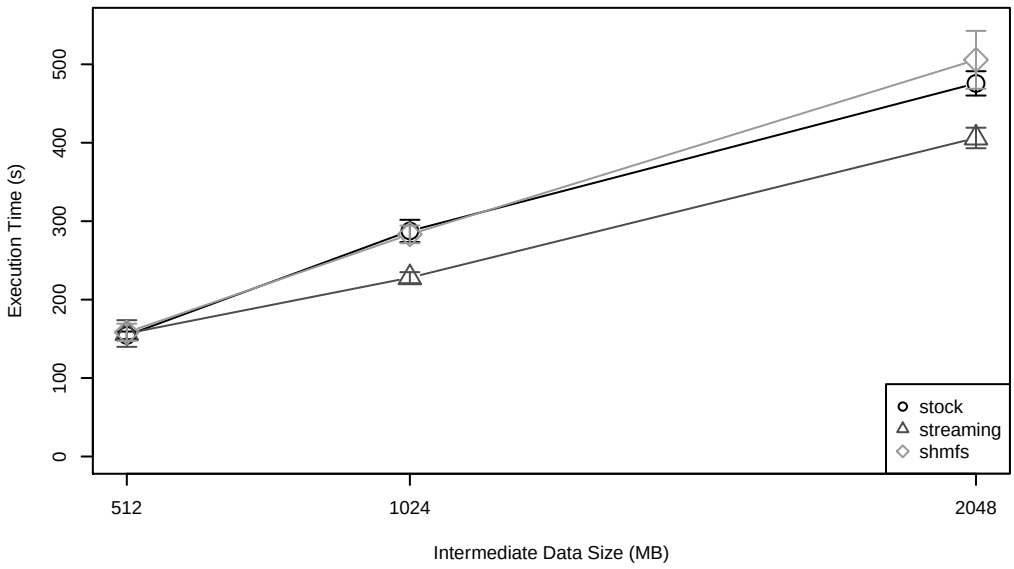


Figure 3.16: Hadoop Microbenchmark Results

3.9 Concluding Remarks

We have shown that shared memory is useful for adding new features to existing shared-memory applications and frameworks, and for making them more suitable for cloud computing. As an example of one such framework, we have developed Elastic Phoenix, a MapReduce framework based on the original Phoenix framework for shared-memory systems.

Our first conclusion from our work on Elastic Phoenix is that inter-VM shared memory allows shared-memory frameworks to be made more suitable for cloud computing environments with only minor API changes. This greatly reduces the effort of porting existing applications. Our experiences porting original Phoenix applications to Elastic Phoenix (Table 3.2) bear out this conclusion.

Our second conclusion from Elastic Phoenix is that the performance overhead incurred by converting a shared-memory framework to distributed operation using inter-VM shared memory is small for compute-intensive tasks. This means that it is feasible to use such converted frameworks and ported applications for performance-intensive tasks. Our experimental results (Figures 3.10 and 3.11) show that for many applications the overhead induced by porting Phoenix applications to Elastic Phoenix is 6% to 8%.

Our final conclusion from Elastic Phoenix is that elasticity can provide an advantage for

certain workloads running on systems with unpredictable resource availability. This will often be the case in cloud computing environments, where resources are usually shared and can be allocated on the fly. Our experimental results (Figure 3.12) show that for certain workloads, taking advantage of additional processors as they become available reduces response time by 29%.

Chapter 4

Shared Memory Memcached

Web applications often have unpredictable load patterns, and cloud computing allows developers to automatically scale their services by allocating resources on demand as traffic grows. In infrastructure-as-a-service (IaaS) systems such as Amazon Elastic Compute Cloud (EC2) and private clouds built using Eucalyptus [48] or OpenStack [10], resources are allocated in the form of virtual machines (VMs). A number of platform-as-a-service (PaaS) offerings (e.g., Google App Engine (GAE) [4], Microsoft Azure [14], AppScale [24]) provide application programming interfaces (APIs) that allow developers to build and deploy web applications that will be automatically scaled based on demand.

Furthermore, many web applications are backed by a datastore, such as a database or a cloud-scale storage engine. For example, Amazon offers SimpleDB, and GAE includes a component called `Datastore` that is backed by BigTable [23]. Because datastore access can be expensive, contributing significantly to web application latency, it is common to use an in-memory cache.

In IaaS environments, application-level caching is usually the responsibility of the application developer, who may choose to deploy a caching service such as memcached [7] using the virtual machines in which their web application is deployed. In PaaS environments, a caching mechanism may be provided in the API (e.g., the `memcache` API in GAE and AppScale). This API is could be backed by memcached (as it is in AppScale) or a similar system.

We have developed a modified version of the memcached server that uses inter-VM shared memory to store cached data, as well as a memcached client for C, Java, and Python that can take advantage of this shared memory design to retrieve cached items without using

A version of this chapter has been published. Adam Wolfe Gordon and Paul Lu. Low-Latency Caching for Cloud-Based Web Applications. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB '11)*, Athens, Greece, 2011

the network. We show that this design can drastically reduce the latency of cache reads, and thus improve web application performance. We also show that our shared-memory-based technique can be faster than sophisticated paravirtualized network devices, such as virtio and vhost in Linux KVM.

4.1 Memcached

Memcached is a popular distributed in-memory cache designed for use in web applications. The system is a simple key-value store, where both keys and values are arbitrary byte sequences. Clients communicate with one or more servers, which keep data entirely in memory. The basic operations are get and set, though atomic operations such as increment and append are also supported. By design, the server has few features: if the user requires replication or distribution, they must be done by the client. Memcached client libraries are available for a variety of programming languages; most support replication and distribution.

A web application uses memcached by storing database or derived data the first time it is fetched or generated. The data may be given a timeout to prevent staleness. Before fetching or generating data, the application checks memcached for a recent version of the data.

Memcached can provide two performance benefits: reduced latency when data can be found in the cache instead of being re-generated, and improved database scalability due to reduced request volume.

Any performance improvement is only possible if data is re-used frequently. For latency reduction, the cost of generating the data again must be greater than the overhead of writing the data to memcached and looking it up again. The benefit of database scaling is only possible if the database server itself is the bottleneck, rather than the network.

Memcached manages a fixed, pre-allocated pool of memory using a slab memory allocator. Memory for cached items is allocated from this pool, and items are linked into a set of global linked lists. Items are separately inserted into a hashtable based on their key. Memcached uses lazy expiration logic to avoid the need to frequently scan the item list. Items are removed from the cache only when they are requested after expiry or when the available memory pool gets low.

There are two memcached protocols: a text-based protocol suitable for manual interaction with a memcached server, and a more efficient binary protocol. Using the binary protocol, a read requires at least two network operations: one to request the item and re-

ceive a header, and a second one to receive the value or error message after determining the request status and the size of the value or message to be received. Memcached supports quiet operations for which no responses are sent until a non-quiet request is used.

4.2 Related Work

The idea of using shared-memory IPC is not new. For example, the fbufs mechanism goes back to 1993 and, more recently, a variety of Xen-based projects have used shared memory to speed up sockets-based IPC (e.g., XenSocket [67]). The Nahanni memcached approach is different in that it shares data across VMs (not just intra-VM domains, like fbufs) and the pointer-based hash table of memcached can be placed in shared memory and directly accessed by the clients, avoiding the potential bottleneck of the memcached server as well as network latency.

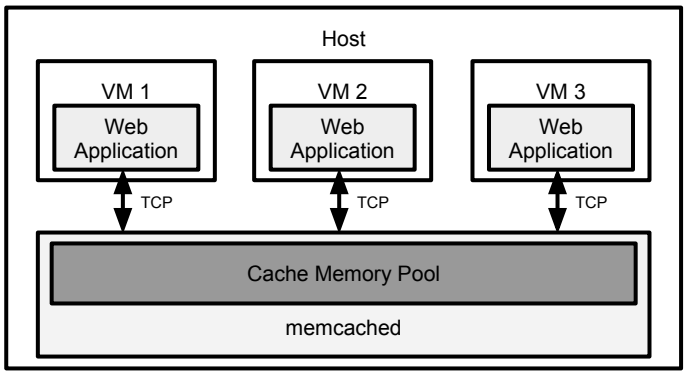
Nahanni memcached shares many similarities with the global shared buffer cache in Disco [21]. However, whereas in Disco the shared memory region was used to implicitly cache filesystem buffers for homogeneous operating systems, memcached allows applications running on any operating system to explicitly cache arbitrary data.

Automatic, consistency-preserving caching of datastore results [60] and transactionally-consistent caching [51] are active areas of research. The goal in such consistency-preserving caching is to enable automatic caching, particularly for applications where transactional consistency is vital. However, work on cache consistency does not address the latency of caching, nor is it aimed at memcached applications, which often gain performance by tolerating some degree of staleness. Work on memcached performance [56] has focused primarily on scaling memcached servers for large deployments, and does not take advantage of the structure of cloud environments to reduce latency.

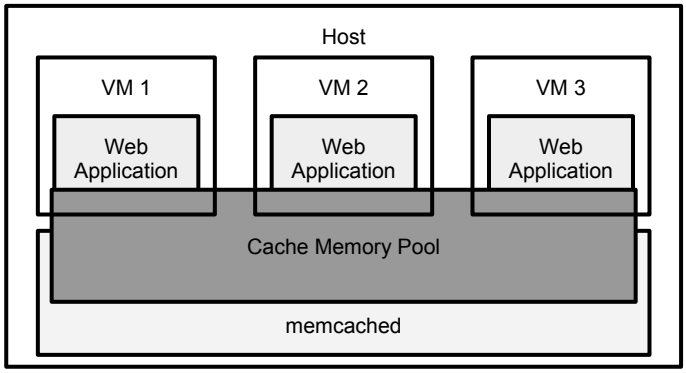
Something similar to Nahanni memcached was implemented for a cloud environment on the IBM Blue Gene/P (BG/P) [17]. Unlike Nahanni memcached, the BG/P hardware has physically distributed memory and a fast RDMA mechanism. But, like Nahanni memcached, the BG/P work attempts to reduce the latency of cache look-ups using RDMA instead of normal network mechanisms.

4.3 Nahanni Memcached

We have modified the memcached server to keep cached data in a shared memory region. The server can run either on a physical host, or in a VM using the Nahanni mechanism.



(a)



(b)

Figure 4.1: Cloud-Based Web Application Using (a) Standard Memcached and (b) Nahanni Memcached

To a standard memcached client, the server acts exactly like a standard memcached server. However, a Nahanni-aware memcached client, which we have implemented for C, Java, and Python, can fetch items directly from shared memory. This design avoids the latency of requesting and receiving cached data over the network, and significantly reduces load on the memcached server.

Figure 4.1 shows the architecture of a web application running in virtual machines and communicating with a co-located memcached server. In Figure 4.1(a), we show the standard memcached server, which communicates with the application using a TCP connection over the virtualized network. Figure 4.1(b) shows Nahanni memcached, where the client shares memory with the server. Omitted in both scenarios are connections to the memcached server from other clients — web applications running in VMs on other hosts, which use the network to contact either version of the server.

4.3.1 Item Storage and Lookup

Because the standard memcached server allocates a pool of memory, then handles item allocation internally, minimal changes were required to have memcached store cached data in shared memory. Instead of `malloc`ing a large region of memory for the cache, the server `mmap`s the Nahanni or POSIX shared memory into its address space. Having the cached items in shared memory only solves half the problem: the client still needs the ability to find them.

Our first design allowed clients to access the cached data using a pointer into shared memory, but still required a network roundtrip for each get operation. This was a simple modification, since the memcached server allocates memory for cached data using its own memory allocator. The allocator was modified to use the shared memory region as its memory pool. We also modified the memcached protocol, adding a new data type. In unmodified memcached, all requests have a data type of `RAW_BYTES`. We added a data type `SHM_PTR`, indicating that the client would like to receive a pointer to the cached data. This relies on the fact that the shared memory region is mapped to the same address in the memcached server and all the clients. The pointer is returned as an integer in the body of the response packet.

This initial design can offer improved performance by eliminating the need to send values over the network. However, given that the data stored in memcached is at most 1 MB, the latency of the network roundtrip dominates the time to read a value from the cache. Because of this, the performance benefits of using shared-memory memcached in this way are minimal.

Our second design uses the shared memory region for the hash tables memcached uses to keep track of key-value pairs, in addition to the data itself. Implementing this involved moving memcached's internal hash table into shared memory. To simplify implementation, we make the hash table a fixed size, limiting the number of items that can be cached without eviction. Since the number of cached items is limited by the size of the cache anyway, we can minimize the effect of this design decision by carefully choosing the size of the hash table based on the expected data size. This design allows clients to look up values in the hash table themselves instead of contacting the server over the network. Even on a fast virtualized network, this saves significant latency. We call this design "direct lookup."

Because our client accesses cached data directly in the server's memory, it is possible that an item will be removed (due to expiry or capacity) between when the client acquires a

pointer to it and when it reads the data. To avoid this race condition, we take advantage of the fact that memcached uses reference counting to enable its multithreaded design. When it finds an item, the client increments the reference count, copies the item's data into a client-local buffer that will be returned to the application, then decrements the reference count.

4.3.2 Item Expiration

Memcached's lazy expiration causes a complication in our design. Because the client library looks up items directly in the server's memory, it will find expired items that the server would not return over the network. However, memcached does not store items with absolute expiry time; instead, it uses a relative time based on when the memcached server was started. In order for the client to behave properly, it must be able to recognize expired items in the cache.

To accomplish this, we add a message to the one-time handshake between the client and the server that occurs when a connection is established. The client sends a special request message to the server, which responds with the real time at which the server was started. The client records this time, which it can use to convert the relative expiry times stored with the cached data into real expiry times.

4.3.3 Locality Discovery

In a simple environment involving only one host, memcached clients may be able to assume that they are running on the same physical host as the memcached server, and thus can use the Nahanni shared memory mechanism to fetch values. However, most cloud environments will employ multiple servers, each potentially running a memcached server, a datastore node, and a number of application VMs. In this case, the memcached client must discover whether each memcached server is co-located with it. We call this process *locality discovery*.

We have designed and implemented a simple mechanism for locality discovery in Nahanni memcached. On startup, the server writes a random 64-bit value called the *cookie* into shared memory. After connecting to the server, the client sends a special locality discovery message. A standard memcached server will reply to this message with an error, but the Nahanni memcached server responds with a packet containing the cookie's address and value. The client checks whether the received value matches the cookie stored at the received address to determine whether the server is local.

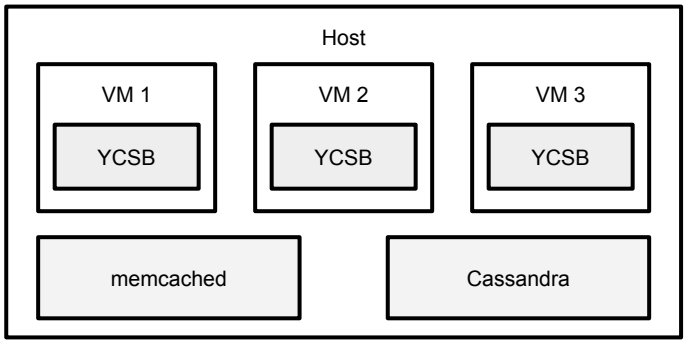


Figure 4.2: YCSB Setup for Experiments

4.4 Latency Evaluation

We evaluated the performance of Nahanni memcached using the Yahoo Cloud Serving Benchmark (YCSB) [27]. YCSB is intended to benchmark cloud-scale datastore systems by providing configurable workloads that can model the loads provided by web applications. The *core workload* provided by YCSB can be configured with a target throughput; the distribution of operations: read, insert, update, and scan; and the statistical distributions to use for selecting which record to read or update and how many records to scan. The YCSB framework reports latency statistics for each operation.

We have implemented a YCSB workload (the *cache workload*) that caches the values it reads. The cache workload allows the user to specify one or more memcached servers; the validity time for cached data; and whether cached data should be invalidated on updates. The cache workload reports latency statistics for cache reads and cache writes, and the hit rate of the cache, in addition to the datastore latency statistics reported by YCSB.

In order to evaluate the latency advantages provided by caching in general, and Nahanni memcached specifically, we needed a measurement that would account for the overheads of caching (such as writing new values to the cache and the cache misses incurred before each datastore read) as well as the latency of the successful cache operations and datastore reads. For this purpose, we define the *average total read latency (ATRL)* as the total time spent on read-related operations, divided by the total number of read operations performed. Most reads will be much faster (on a cache hit) or much slower (on a cache miss) than the ATRL, but it is useful in comparing the latency of reads without caching to the latency of reads with caching, and for comparing different caching systems.

For our benchmarks, we used an Intel Xeon X5550 (2.67 GHz) server with two sockets, eight cores (16 hyperthreads), and 48 GB of RAM. We ran the Cassandra [44] datastore and

Workload	Operations	Record Selection	Application Scenario
Read only	100% Read	Zipfian	Static, externally-generated user information
Read heavy	95% Read, 5% Update	Zipfian	Photo tagging; adding a tag is an update, but most accesses are reads
Read latest	95% Read, 5% Insert	Latest	Status updates; users mostly view the newest statuses

Table 4.1: Workload Parameters for Evaluation

memcached natively (not in a VM) and ran YCSB in virtual machines. Each VM has four virtual CPUs and 4 GB of RAM. Cassandra was chosen because it is a typical example of a cloud-scale datastore, and is open-source. We use a 4 GB memory pool for memcached in both the standard and Nahanni configurations. Except where otherwise specified, the VMs were connected to the host and each other by a Virtual Distributed Ethernet (VDE) [28], and use the paravirtual virtio [55] network interface provided by QEMU/KVM. This setup is illustrated in Figure 4.2.

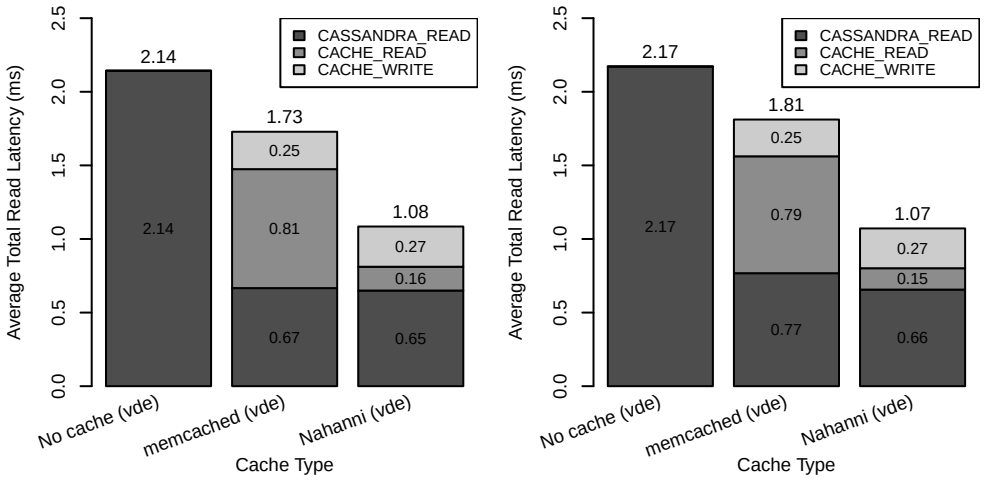
While this testbed does not accurately represent a cloud environment with multiple servers, it does provide something of a worst case. Cassandra is fast because it is using only a single node, and the virtual network the client uses to connect to memcached and Cassandra has low latency. This means that in order to beat the non-caching setup and the standard memcached setup, Nahanni memcached must provide extremely low overall latency.

We used three different sets of parameters with our cache workload in YCSB. For brevity, we refer to each set of parameters as a workload, even though they use the same workload component in YCSB. Our workloads are summarized in Table 4.1. We used the same sets of parameters the YCSB authors [27] used to evaluate datastores, choosing the three workloads that best reflect scenarios in which web applications would use memcached.

We loaded the YCSB datastore with one million records, and ran each workload with three YCSB clients running at 2000 operations per second each, for a total of 6000 operations per second. Each client executed 12 million operations, for a total of 36 million operations.

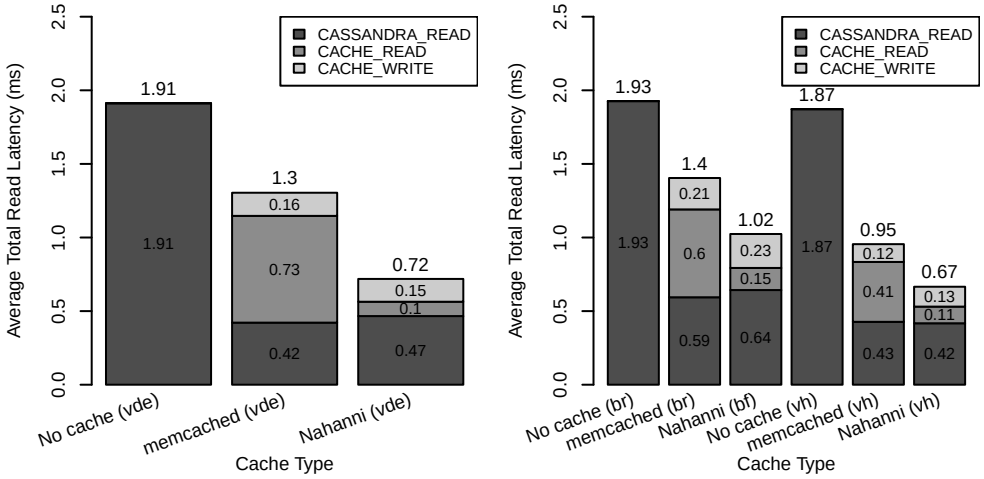
4.4.1 Results With VDE Networking

We present the results for the three workloads in Figure 4.3. The bars represent the ATRL, with the segments representing the contribution of datastore read (CASSANDRA_READ),



(a) Read only workload with VDE

(b) Read heavy workload with VDE



(c) Read latest workload with VDE

(d) Read heavy with bridging (br) and vhost (vh)

Figure 4.3: YCSB Benchmark: Breakdown of Average Total Read Latency (ATRL). Bar segments represent the contribution of Cassandra read, memcached cache read, and memcached cache write to the ATRL. "No cache" uses only Cassandra. "Nahanni" uses the network for Cassandra reads and cache writes.

cache read, and cache write to the total.

Read Only Workload

The cache hit rate for the read only workload (Figure 4.3(a)) was 71%. For this workload, the standard memcached setup improved the ATRL by a modest 19% (i.e., 1.73 vs. 2.14) compared to the non-caching scenario. Nahanni memcached improves the cache read latency by 80% (i.e., 0.16 vs. 0.81), leading to a 38% improvement in ATRL (i.e., 1.08 vs. 1.73) compared to the standard memcached setup.

Read Heavy Workload

For the read heavy workload (Figure 4.3(b)) standard memcached provides a 17% improvement (i.e., 1.81 vs. 2.17) over the non-caching case (i.e., using only Cassandra) with a 71% cache hit rate. Nahanni memcached further improves the ATRL by 41% (i.e., 1.07 vs. 1.81) compared to standard memcached, improving the cache read latency by 81% (i.e., 0.15 vs. 0.79).

Note that the read heavy workload involves update operations. Although our YCSB cache workload supports cache invalidation, we left it disabled. Applications that use memcached for mutable data are usually tolerant to some degree of staleness, since cache invalidation must be handled by the application. Also, the current implementation of Nahanni memcached still uses sockets-based messages to send updates to the memcached, which is why the CACHE_WRITE portions of the graphs (i.e., 0.27 vs. 0.25 per read) are similar. Tackling the latency of cache writes is future work.

Read Latest Workload

The cache hit rate for the read latest workload (Figure 4.3(c)) was 81%, which is higher than for the read heavy workload (71%) because the latest distribution favors the newest records heavily, more than the Zipfian distribution. This also allows Cassandra's internal cache to be more effective, thus decreasing the average datastore read latency.

The high cache hit rate allows standard memcached to improve the ATRL by 32% (i.e., 1.3 vs. 1.91). Nahanni memcached further improves the ATRL by 45% (i.e., 0.72 vs. 1.3) compared to standard memcached, due to an 86% (i.e., 0.1 vs. 0.73) reduction in cache read latency.

4.4.2 Paravirtualized Networks

The experiments described in Section 4.4.1 used VDE to provide the virtual network. VDE has the advantage that a non-root user can start new VM instances with no administrator intervention. However, if one is willing to provide root or sudo permissions to accounts that start up VM instances, higher-performance network setups are possible. We are wary of such privilege levels, but not everyone in the Linux KVM community is similarly concerned, and a common practice is the use of selective `sudo` privileges to enable normal users to run VMs with such setups.

One standard mechanism for virtualized networking is to use a bridged tap device for each VM. This requires per-VM setup from the administrator or sudo permissions for the user running the VM. The experimental vhost mechanism in Linux KVM extends the bridging mechanism and further improves performance by reducing the number of protection domain crossings required for network I/O. We display results from the read heavy workload using bridged (br) and vhost (vh) network setups in Figure 4.3(d).

Our main conclusion is that the Nahanni memcached configurations still have the lowest latencies, although the specific amount of improvement might be more modest: using bridging without vhost ("br" in Figure 4.3(d)), Nahanni is 27% faster (i.e., 1.02 vs. 1.4) than using standard memcached. And, using both bridging and vhost ("vh" in Figure 4.3(d)), Nahanni is 29% faster (0.67 vs. 0.95). After all, bridged networking and vhost *also* improve the performance of Nahanni memcached since sockets are still being used for updates and for IPC with Cassandra when we miss in the cache.

4.5 Concluding Remarks

We have presented the design and implementation of Nahanni memcached, showing how an existing server can be converted to use inter-VM shared memory for client communication in a cloud environment. We evaluated our implementation using a microbenchmark (YCSB), as well as with a full application (AppScale).

Using the Yahoo Cloud Serving Benchmark (YCSB), we have empirically shown that Nahanni memcached, used with VDE networking, can improve the overall read latency for applications by up to 45% (i.e., read latest workload) compared to standard memcached, resulting from a reduction in cache read latency of 81% or more. Even when combined with state-of-the-art paravirtualized network mechanisms, such as vhost, Nahanni memcached can still reduce the latency by up to 29%.

Chapter 5

Concluding Remarks

Cloud computing is now a popular mechanism for deploying software, and a number of programming models, software services, and frameworks have been built to facilitate the development of cloud applications. Cloud computing allows for the easy scaling of services and efficient use of resources by sharing them among multiple customers.

We argue that as private and hybrid clouds become more common, and the core counts of commodity servers continue to grow, it will become more common for multiple virtual machines used for a cloud service to be co-located on a single physical host. In such circumstances, it will become possible in some cases to use mechanisms such as inter-VM shared memory in production applications. But, inter-VM shared memory is easiest to take advantage of if it is integrated into a framework or service such that the developer can use it in existing applications.

We have modified an existing framework and an existing service to take advantage of inter-VM shared memory. In doing so, we have demonstrated the following contributions.

1. In Chapter 3, we showed how inter-VM shared memory can be used to add features to existing cloud programming frameworks. To this end, we presented Elastic Phoenix, a MapReduce framework for shared-memory systems and co-located virtual machines, which allows for malleable jobs in which resources can be added and removed as the job runs. Elastic Phoenix extends the existing Phoenix MapReduce framework with malleability, as well as the ability to run in VMs, features enabled by inter-VM shared memory. Using Elastic Phoenix, we showed that the overhead of malleability is low (6% to 8%) for some classes of applications, and that taking advantage of malleability can improve the response time of a workload by 29%.

2. In Chapter 4, we showed how inter-VM shared memory can be used to reduce the latency of cloud services. To demonstrate, we presented Nahanni memcached, a

version of the popular memcached object caching server that uses inter-VM shared memory to significantly reduce the latency of caching operations for applications running in co-located VMs. We used the Yahoo Cloud Serving Benchmark to show that inter-VM shared memory can reduce the total latency of read-related operations in a workload by up to 45%, resulting from a reduction in cache read latency of up to 86%.

We have presented two use cases for inter-VM shared memory in which programmers can take advantage of the new mechanism with existing code. We accomplished this by modifying frameworks and services that are used in existing applications, providing advantages (malleability and reduced latency) transparently.

5.1 Future Work

Although we have developed working implementations of Elastic Phoenix and Nahanni memcached, there are a number of areas in which each can be improved and extended. Our evaluation could also be extended in certain ways. We describe specific areas of future work for Elastic Phoenix and Nahanni memcached separately in the following sections.

5.1.1 Elastic Phoenix

In Section 3.7, we described a number of limitations of our current Elastic Phoenix implementation. Some of these limitations are inherent to the design of Elastic Phoenix, but most are the result of implementation decisions, and many can be eliminated. Here we describe those that could be dealt with through further implementation work.

Job Size

The input size of a Phoenix job is limited, as we described in Section 3.7.1. The basic source of this limitation is that the amount of inter-VM shared memory available to Elastic Phoenix is limited and set before the job starts. This limitation is inherent to the design of Elastic Phoenix, but one area of future work is to reduce the impact of this limitation by allowing larger jobs to run successfully. One way to do this is to develop a better memory allocator; the simple design of the current allocator wastes quite a bit of memory. Implementation of combiners, described below, would also help allow for larger jobs, by reducing the number of intermediate key-value pairs produced.

Combiners

As described in Section 3.7.2, Elastic Phoenix does not support map-side combiners for intermediate data, which is a standard feature in other MapReduce frameworks, including the original Phoenix framework. Adding support for combiners in Elastic Phoenix would be a straight-forward modification, as it is likely possible to copy the combiner handling code directly from original Phoenix.

Multiple MapReduce Jobs

As described in Section 3.7.3, Elastic Phoenix supports only one MapReduce job per application, a limitation not present in the original Phoenix framework. One problem in making Elastic Phoenix support multiple jobs is how newly-initialized workers can know which job is currently running. In the current design, a new worker joining a computation need only know what stage of the computation is currently taking place: map, reduce, or merge. With multiple jobs, there are multiple map, reduce, and merge stages, and a new worker executing the wrong code will corrupt the job's data. This could be dealt with by counting the number of MapReduce jobs that are started by the master, or by requiring the application to give a unique name to each job.

5.1.2 Nahanni Memcached

Our current implementation of Nahanni memcached supports all the features of standard memcached. However, there are still some areas of future work, such as further evaluation and performance improvements for operations other than cache reads.

Platform Integration and Real-Application Evaluation

Nahanni memcached is currently suitable for use in an IaaS environment with Nahanni shared memory support. That is, it could deployed by a developer in VMs along with a web application. However, many applications are deployed in PaaS environments where memcached is provided as a service.

One area of future work is to integrate Nahanni memcached into a PaaS environment so that existing applications using a particular API can take advantage of the reduced latency provided by shared-memory caching. We plan to integrate Nahanni memcached into AppScale, an open-source framework that emulates the GAE API. This will allow us to evaluate the benefits of shared-memory caching in a real web application. In preparation for this integration, we have already implemented Nahanni-aware memcached client libraries for Java

and Python, the languages supported by AppScale.

Direct Cache Writes

In effective uses of memcached the application looks up each item in the cache before reading it from the datastore, and experiences a high cache hit rate. In such an application, cache reads are much more common than other operations, since cache writes are only required on cache misses. Therefore, we targeted read latency in our design and our current implementation requires the memcached client to communicate with the server over the network for all operations other than reads. However, since the client has direct access to the cached data, it is possible to implement direct writes in the client.

Allowing client-side modifications to the cache will help solve another problem: expiration. In our current implementation, the client recognizes and ignores expired cache data, but does not remove it from the cache. This leads to the server performing an expensive cleanup operation when the cache gets full. With client-side writes implemented, it would be a natural extension to allow the client to remove expired items from the cache when they are found, as the server does.

Scalability Evaluation

Intuitively, allowing direct access to cached data should improve the scalability of the memcached server by removing the bottleneck of handling client connections. However, memcached uses coarse-grained locking to protect the cache against concurrent modifications, a bottleneck Nahanni memcached does not avoid. An area of future work is to evaluate the scalability bottlenecks of standard memcached, and determine whether Nahanni memcached helps avoid them.

Bibliography

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[3] Apache Hadoop. http://hadoop.apache.org/.

[4] Google App Engine. http://code.google.com/appengine/.

[5] HBase. http://hbase.apache.org/.

[6] Hypertable: An open source, high performance, scalable database. http://hypertable.org.

[7] Memcached - a distributed memory object caching system. http://memcached.org/.

[8] MongoDB. http://mongodb.org/.

[9] Nahanni. http://gitorious.org/nahanni.

[10] OpenStack open source cloud computing software. http://openstack.org/.

[11] Project Voldemort. http://project-voldemort.com/.

[12] scalaris — distributed transactional key-value store. http://code.google.com/p/scalaris/.

[13] SLOCCount. http://sourceforge.net/projects/sloccount/.

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

[15] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGPLAN Notices*, 41(11):2, October 2006.

[16] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys '10: Proceedings of the 5th European Conference on Computer Systems*, pages 223–236, Paris, France, 2010. ACM.

[17] Jonathan Appavoo, Amos Waterland, Dilma Da Silva, Volkmar Uhlig, Bryan Rosenburg, Eric Van Hensbergen, Jan Stoess, Robert Wisniewski, and Udo Steinberg. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 385–394, Chicago, Ill, 2010. ACM Press.

[18] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, April 2010.

[19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, 2003. ACM.

[20] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[21] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM SIGOPS Operating Systems Review*, 31(5):143–156, December 1997.

[22] Chris Bunch, Navraj Chohan, Chandra Krintz, Jovan Chohan, Jonathan Kupferman, Puneet Lakhina, Yiming Li, and Yoshihide Nomura. An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 305–312. IEEE, July 2010.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008.

[24] Navraj Chohan, Chris Bunch, and Sydney Pang. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *First International Conference on Cloud Computing*, Munich, 2009.

[25] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Chandra Krintz. See spot run: using spot instances for mapreduce workflows. In *HotCloud'10: Proceedings of the 2nd USENIX Conference on Hot topics in Cloud Computing*, Boston, MA, June 2010. USENIX Association.

[26] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *NSDI '10: Proceedings of the 7th Conference On Networked Systems Design And Implementation*, pages 21–21, San Jose, California, 2010. USENIX Association.

[27] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10: Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, Indianapolis, Indiana, 2010. ACM.

[28] R. Davoli. VDE: Virtual Distributed Ethernet. In *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 213–220. IEEE, 2005.

[29] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, 2004. USENIX Association.

[30] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07: Proceedings of the twenty-first ACM Symposium on Operating Systems Principles*, pages 205–220, Stevenson, Washington, 2007. ACM.

[31] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *SOSP '93: Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 189–202, Asheville, North Carolina, 1993. ACM.

[32] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. Dynamic Malleability in Iterative MPI Applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 591–598. IEEE, May 2007.

[33] Kaoutar El Maghraoui, Boleslaw K. Szymanski, and Carlos A. Varela. An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. *Parallel Processing and Applied Mathematics*, 3911:258–271, 2006.

[34] Dror Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer Berlin / Heidelberg, 1997.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Operating Systems Review*, 37(5):29–43, October 2003.

[36] R P Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974.

[37] W. Gropp and E. Lusk. Dynamic process management in an MPI setting. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 530–533, San Antonio, Texas, October 1995. IEEE.

[38] R.L. Grossman. The Case for Cloud Computing. *IT Professional*, 11(2):23–27, March 2009.

[39] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: batched stream processing for data intensive distributed computing. In *SIGMOD '10: Proceedings of the 2010 International Conference on Management of Data*, pages 63–74, Indianapolis, Indiana, 2010. ACM.

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, Lisbon, Portugal, 2007. ACM.

[41] Frans Kaashoek, Robert Morris, and Yandong Mao. Optimizing MapReduce for Multicore Architectures. Technical report, Massachusetts Institute of Technology, May 2010.

[42] Steven Y Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. Making cloud intermediate data fault-tolerant. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 181–192, Indianapolis, Indiana, 2010. ACM.

[43] F. John Krautheim. Private virtual infrastructure for cloud computing. In *Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, San Diego, California, June 2009. USENIX Association.

[44] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35, April 2010.

[45] Gunho Lee, Niraj Tolia, Parthasarathy Ranganathan, and Randy H. Katz. Topology-aware resource allocation for data-intensive workloads. In *APSys '10: Proceedings of the first ACM Asia-Pacific Workshop on Systems*, New Delhi, India, August 2010. ACM.

[46] Cam Macdonell. *Fast Shared Memory-based Inter-Virtual Machine Communications*. PhD thesis, University of Alberta, 2011.

[47] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *2010 Proceedings IEEE INFOCOM*, San Diego, California, March 2010. IEEE.

[48] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE, May 2009.

[49] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1110, Vancouver, British Columbia, Canada, 2008. ACM.

[50] Jing Tai Piao and Jun Yan. A Network-aware Virtual Machine Placement and Migration Approach in Cloud Computing. In *2010 Ninth International Conference on Grid and Cloud Computing*, pages 87–92. IEEE, November 2010.

[51] Dan R.K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI '10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, British Columbia, Canada, 2010. USENIX Association.

[52] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI '10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, British Columbia, Canada, 2010. USENIX Association.

[53] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pages 199–212, Chicago, Ill, November 2009. ACM Press.

[54] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. It's Time for Low Latency. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, California, 2011. USENIX Association.

[55] Rusty Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[56] Paul Saab. Scaling memcached at Facebook. `http://www.facebook.com/note.php?note_id=39391378919`, 2008.

[57] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, September 2009.

[58] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. *ACM SIGPLAN Notices*, 38(10):216, October 2003.

[59] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, August 2009.

[60] Niraj Tolia and M. Satyanarayanan. Consistency-preserving caching of dynamic database content. In *International World Wide Web Conference*, pages 311–320, Banff, Alberta, 2007.

[61] Abhishek Verma, Nicolas Zea, Brian Cho, Indranil Gupta, and R.H. Campbell. Breaking the MapReduce Stage Barrier. In *IEEE International Conference on Cluster Computing*, pages 235–244, Heraklion, Greece, 2010.

[62] Adam Wolfe Gordon and Paul Lu. Elastic Phoenix: Malleable MapReduce for Shared-Memory Systems. In *8th IFIP International Conference on Network and Parallel Computing (NPC 2011)*. Springer, 2011.

[63] Adam Wolfe Gordon and Paul Lu. Low-Latency Caching for Cloud-Based Web Applications. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB '11)*, Athens, Greece, 2011.

[64] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207. IEEE, October 2009.

[65] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar, and Gunda Jon. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, California, 2008. USENIX Association.

[66] Matei Zaharia, M Chowdhury, M Franklin, and S. Spark: Cluster computing with working sets. Technical report, UC Berkeley, 2010.

[67] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. Xensocket: a high-throughput interdomain transport for virtual machines. In *MIDDLEWARE2007: Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, pages 184–203, Berlin, Heidelberg, 2007. Springer-Verlag.