Karlsruher Institut für Technologie

Enabling fast VM deployment and migration in tightly interconnected HPC cloud environments

Studienarbeit
von

Manuel Gutekunst

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter:	Dr. Jan Stoess

Bearbeitungszeit: 30. October 2011 – 31. January 2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 29. Januar 2012

Abstract

Cloud computing environments play an important role in the today's computer era. Therefore computing clouds need to be able to deploy several hundreds to even thousands of virtual machines in a scalable way and to provide adequate migration and backup strategies. To achieve this, cloud providers need to use efficient mechanisms to deploy and migrate VMs and need to avoid potential bottlenecks like a central storage system. In this thesis we present our approach to enable fast deployment and migration in an HPC environment that exploits the tight interconnects of a Blue Gene/P supercomputer. Our approach uses main-memory-based distributed storage and runs directly inside the hypervisor that allows our approach to directly access the VM's memory. By using the remote DMA capabilities of BG/P's interconnects we are able to minimize load at the storage nodes and reduce the potential of a bottleneck they pose. Although the main purpose of our approach is to provide snapshotting, it is able to copy any data into the servers.

Acknowledgments

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

Contents

Chapter 1

Introduction

In the modern computer age, cloud computing becomes more and more important. Customers have to employ only the resources needed and to pay only the computation time used. They also do not need to administrate their own server farms. Using a cloud provider like Amazon [4], customers can significantly reduce costs but get an equivalent deployment as when they would purchase dedicated hardware. Also customers benefit from the enhanced flexibility a cloud environment provides. They can, for example, use an operating systems of their choice. Also a customer has access to the system-level of his operating system of choice but does not endanger other customers if he uses buggy or even malicious software [10].

This flexibility can be achieved by giving the user complete control over the virtual machines he leased. The drawback of this approach is that it can put a great burden on the cloud provider since his environment has to quickly serve customers requests. These requests could include the need for additional resources or the wish to regularly backup his VMs. Having many customers, the need for additional resources can result in the necessity to load several hundreds to even thousands of VMs at the same time which also puts heavy load on the network interconnect and the backing storage.

There has been some research to satisfy these challenges. One attempt is to quickly deploy lots of virtual machines at the same time to satisfy the sudden resource hunger of some customers. SnowFlock [17] or the Potemkin Honeyfarm [31] are popular concepts to solve this issue, as they can significantly reduce VM start-up time. But reducing start-up time does not only increase the reactivity of the customer's VMs. It can reduce cost of the customer as well, since the time a customer uses the cloud is often accounted in his bill.

Once all VMs are running, another challenge occurs. That of snapshotting the VM for checkpointing or to perform migration without impeding the VM. Clark et. al. [8] explored this challenge and present an approach that tries to minimize the downtime of a VM while it is migrated. Migration can also be

useful to balance the load among several CPUs to increase availability and to reduce cooling costs [21].

We try to solve the problem of fast deployment and migration by using the advantages a high performance computer (HPC) provides us with. Normally a supercomputer is, for example, used by meteorologists for weather forecasts [28] or to do very complex computation in physics like to calculate and simulate the events that happened immediately after the Big Bang [3]. Therefore supercomputers offer several thousand CPUs, numerous Gigabytes of main memory and fast network interconnects and hence resemble today's cloud environments in many aspects.

Computing clouds as well as supercomputers are developed to allow access of multiple users who only use a part of the system for their jobs. However cloud providers often use commodity hardware, whereas supercomputers tend to use special processing elements and interconnects to enable high performance. Additionally supercomputer software is often specially adapted to exploit hardware features and can therefore only run on a particular architecture. Nevertheless a HPC can be a considerable advantage for a cloud environment. For example, by exploiting the fast network interconnect of an HPC we can reduce the time it takes to start a VM in comparison to Amazon's EC2 and satisfy one of the requirements of a cloud customer. We also believe that snapshotting can benefit from the HPC environment.

In the remainder of this thesis we present a our approach to enable fast VM deployment and migration on an HPC. We begin by giving a brief overview of related work, continue with our design and describe some of the implementation issues we experienced. Finally, we will evaluate and conclude our work.

Chapter 2

Related Work

In this section we will give a brief overview of the work that is done on migration and deployment of virtual machines in cloud environments.

2.1 Deployment

On the topic of fast VM deployment some research exists. Snowflock [17] presents VM fork, a mechanism to rapidly clone a running virtual machine, similar to the Unix fork system call. VM fork uses memory-on-demand, a lazy fetch algorithm, lazy state replication and avoidance heuristics. However, Snowflock is limited to cloning running virtual machines, no static storing of a VM state is provided. Also, each VM initially originates from one node, which could become a bottleneck if the number of child VMs becomes large enough.

The Potemkin Virtual Honeyfarm [31] is another approach that allows fast booting of virtual honeyfarms. A honeyfarm is a system that is extensively monitored to detect and analyze intrusion. For this purpose, it is often left unprotected. The Potemkin Virtual Honeyfarm supposes that different virtual machines often use the same unmodified content. Following this assumption, Potemkin uses a copy-on-write semantic to reduce data transfer costs on the creation of new virtual machines. While this might be true on virtual honeyfarms it is not necessarily the case on a more general workload. Additionally, Potemkin needs one running VM that can be used for copy-on-write and because copy-on-write does not work between several servers, it effectively limits each physical server to one operating system and one application which renders Potemkin's approach even more special.

The European Organization for Nuclear Research (CERN) uses a BitTorrent based algorithm to distribute a 10 GB image among 400 physical nodes within 30 minutes [32]. While this is impressive if the nodes are spread all over the

world, it is definitely too long to satisfy our criteria of fast deployment in a cloud environment. During their research, two methods for image distribution were tested: a secure copy (SCP) based copy algorithm [1] and a BitTorrent based one. The SCP based algorithm starts with a slow transfer but grows logarithmic. Still, the BitTorrent based algorithm was the faster one. In the CERN approach, each hypervisor locally stores all supported images, which reduces boot time but also might be unnecessary when some images are never booted from a hypervisor. Additionally, it puts extra load on the network interconnects.

Nicolae et al. [23] present a virtual file system for VM storage. Their goal is to boot many images from a persistent storage, to periodically store their state and to migrate them offline. For their storage they use a part of each local disk of the cloud's PCs, reducing the available space for hypervisor and virtual machines. This is similar to our approach, albeit we take it a step further and use the main memory of particular nodes as storage. They rely on BlobSeer [22], a distributed storage system, as their storage management system. BlobSeer uses segment trees to manage its data and handles its metadata in a distributed fashion to avoid a potential bottleneck. Also, the images are split into chunks to reduce contention. For snapshotting, they try to only store incremental changes. To avoid interdependencies between different images, a cloning mechanism is used. On booting, the image is presented as a local file, but reads into missing pages are trapped and issue a fetch instruction. Writes are always performed local. This approach on the one hand reduces network traffic, but on the other hand it increases boot time and decelerates the virtual machine until all needed data is at least touched once.

In another article, Nicolae et al. extend their previous work with self-adaptive prefetching [24]. Again, they use a part of each local disk of the cloud's servers as their storage and BlobSeer as their management system. But this time, a prefetching module is added, that adds access information to the tree nodes BlobSeer uses to manage the chunks. With these information they try to reduce the boot time by sending prefetching hints to the hypervisors. The hints are piggybacked on normal read requests and the actual data fetch is done during periods of I/O inactivity.

2.2 Migration

The authors in [16] pointed out that a successful VM deployment needs to use a distributed file system. They focus on an ubiquitous working environment. Therefore they use offline migration to provide the user with the same operating system and user interface, independently of the location of the user (for example at work and at home). Although they use NFS as backing storage, which could become a bottleneck, they show that migration is possible within a couple of minutes using commodity hardware which is an adequate time for their scenario, because users

normally need a longer time to change locations.

Live Migration of virtual machines [8] focuses on virtual machines with interactive workload. They show that they can migrate a running online game server without the connected clients taking notice of this. This is achieved by dividing the migration process into three phases namely push phase, stop-and-copy phase and pull phase. The push phase is executed several times and is used to copy the memory of the VM while it continues execution. After the first round, where the complete memory is copied, only the pages that changed during the last phase are copied again. This repeats several times until there are only pages left that change too frequently to copy them during execution of the VM. They are copied in the stop-and-copy phase that comes next. In this phase, the source VM is paused, the remaining pages are copied and, once this ended with a success, the new VM is started. When the new VM is in the same network, it sends an ARP message, announcing that the IP address has been moved. At the end, there is the pull phase, where yet still missing pages are fetched from the source VM if they are needed. Although this approach reduces downtime, the total migration time is increased in comparison to a simple stop-and-copy approach. Secondary it forces one to monitor the pages that change during the several copy rounds in the push phase. Third, the migration is limited to a single switched network since a migrated OS shall maintain all open network resources without relying on forwarding, or on redirection mechanisms.

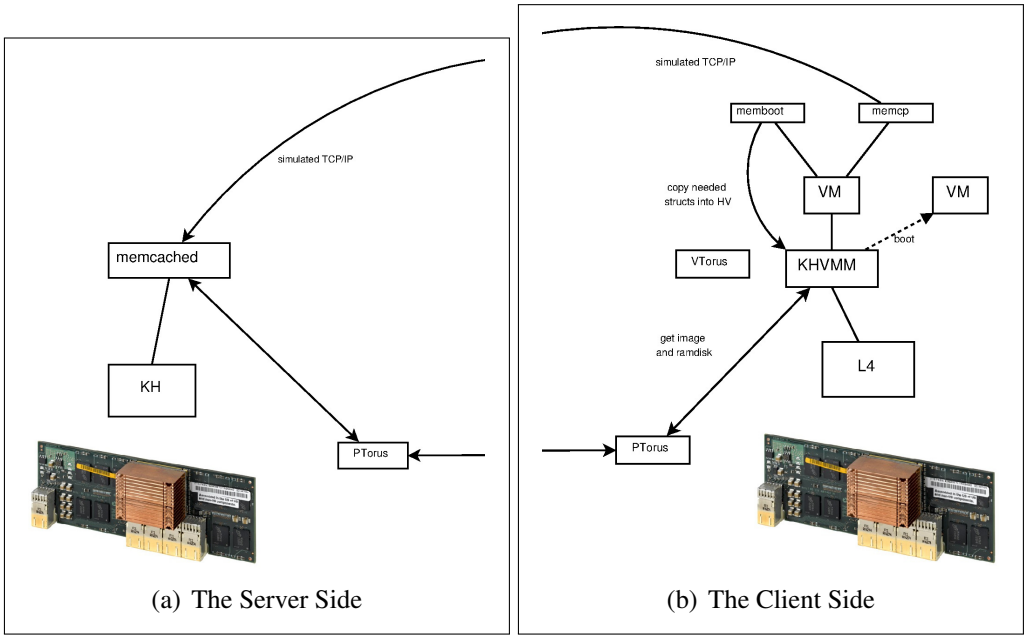
Huang et al. [11] exploit the RDMA feature of modern interconnects to decrease total migration time on live migration. They use an InfiniBand architecture's RDMA feature to transmit pages from the source to the destination node and decrease the time of a copying round by doing so. To further benefit from RDMA they try to cluster pages to send as many data as possible in one RDMA operation. As their tests show live migration benefits from RDMA with a reduced migration time of up to 80 %. We hope to reach similar results in our approach. However Huang et al. limit their research on migration whereas we believe that deployment and snapshotting of virtual machines can also benefit from modern interconnects.

Chapter 3

Design

To enable fast VM deployment and migration, we have to achieve a scalable system and to provide high network speed. We try to accomplish scalability by splitting the data into chunks and distributing them between a number of storage nodes (see section 3.2). High network speed can be attained by using the fast torus interconnect provided by Blue Gene/P (sec. 3.4).

3.1 System Overview



(a) The Server Side

(b) The Client Side

Figure 3.1: client respectively server side of our approach

Our system can be separated into a client and a server side that can be seen on the right respectively the left side of Figure 3.1. While the server side handles our incoming data, the client side represents everything that is not responsible for storing VM images.

The Server Side

The Server Side (Figure 3.1(a)) is mainly responsible for saving and distributing virtual machine images. Several storage systems are possible but we choose a key value storage as it provides the needed flexibility to store and distribute images in a scalable way and also avoids the burden of an entire file system or the complexity of a database system.

The nodes run a server application atop of a native Linux operating system to handle incoming requests and to distribute data. For the actual data transfer, our system exploits the interconnects of the supercomputer, especially its ability to execute remote DMA.

The Client Side

The Client Side (Figure 3.1(b)) represents the management VM, the virtual machines we booted, but also our hypervisor. The management VM is similar to XEN's dom0 [6] and handles communication with the user. Also saving data and booting can be done in this virtual machine.

The hypervisor, on the other hand, handles the actual data transfer from and to the storage back-end. It also initiates the boot process after the needed data is received. Currently, the boot process is not initiated until all data is received. While this approach simplifies implementation, another approach that relies on *lazy copying* is also thinkable. Here, only an initial state is copied and the remaining data is only copied when needed. SnowFlock [17] uses a similar approach albeit they use it to clone an existing VM.

The hypervisor is also responsible for snapshotting and resuming. After the VM issues a snapshotting command, it is paused. The hypervisor then copies its main memory and stores its CPU registers and opened devices into the servers. Afterwards, control is returned to the virtual machine that continues its execution. To resume a VM on a different compute node the hypervisor first restores the main memory. Once this is done, the hypervisor has to either set a new IP address in the VM or it has to notify the virtual machine so that it can set the new address itself. Setting a new IP address is necessary, because the VM is resumed on a different node and it is not easily possible to also migrate MAC and IP address although there are ways to solve that issue [8, 26]. After that, the hypervisor restores the VM's CPU registers and then returns control to the now resumed VM.

3.2 Loading and Storing Images

To enable fast image distribution we use a main memory based distributed storage [9]. Some nodes throughout the installation offer large parts of their main memory as storage. These nodes are called *storage nodes*. As they use main memory as storage media, we can handle data transfers with remote DMA which significantly increases our performance (see section 3.4).

To store data, several steps are taken. First, the data is split into equal sized chunks and a hash value based on the name of the data plus its chunk number is calculated. This value is then split into two parts. The first one is used to specify the server on which this particular chunk shall be saved since every storage node is responsible for a range of possible hash values. The second part is used as an index into a *hashtable* that contains the memory location for each chunk. Every node maintains such a hashtable avoiding the need for a central entity that might become a bottleneck.

We use a pseudo-random hash function for several reasons. First, only the name of the data needs to be known to receive it from the storage nodes. Second, since the hash function is pseudo-random, it ensures an equal distribution among the available storage nodes which leads to the third advantage: when several clients start to load the same data the load is uniformly distributed among the servers. The result is an equal utilization after the initial phase, where all clients try to load the first chunk, is completed.

Our approach has several downsides as well. Adding a new server implies copying most chunks to keep the uniform distribution. Although our approach is not bound to a specific hash function, so that we can exchange it to solve this problem, we might also affect the advantages our hash function offers. Another drawback is that an image, once it is uploaded, is an immutable entity. In our current design, if we want to change an image, for example, if we want to update a snapshot, we need to upload the whole image again. If the snapshots become large, this definitely impacts our performance. At the moment it is not possible to share identical chunks between different images, either.

3.3 Booting and Snapshotting

The typical procedure to store, snapshot, boot and resume an image respectively a virtual machine is depicted in Figure 3.2. At the beginning the user of the VM who has the appropriate rights to call the corresponding commands, stores one or more images in the storage servers using the management VM. If he then wants to boot a specific image, he sets up a request. The hypervisor then collects the chunks and, after all chunks are fetched, it issues the booting process. This approach

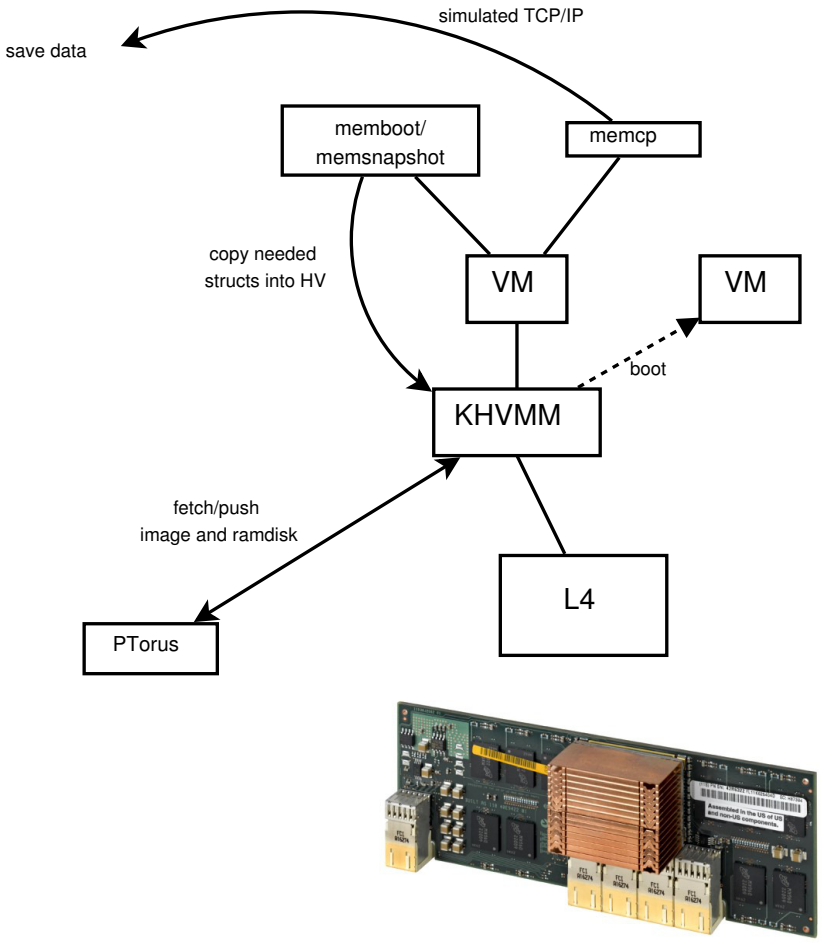


Figure 3.2: Illustration how data is saved and loaded to deploy VMs. The VM copies needed data into the hypervisor which then issues the data transfer and the booting procedure

reduces computation time of the issuing VM because the main data transfer and the following booting procedure is handled in the hypervisor and does not involve any action from the VM but calling the appropriate command.

If the VM user wants to do a snapshot of his VM, he issues the snapshotting command and a name of the snapshot as an argument. The underlying hypervisor then handles the necessary data transfer. Meanwhile the VM pauses so that the hypervisor copies a consistent state. While this approach simplifies implementation, it forces a downtime on the virtual machine while it is saved. Depending on the time the VM is paused, other approaches are also feasible to reduce the downtime during the snapshotting procedure. One approach is used in [8] for migration but is adaptable to our scenario with little effort. Here, the migration

process is divided into a *push phase* where certain pages that are not frequently accessed are copied in several rounds, a *stop-and-copy phase*, where the source VM is paused and the new one is started and finally a *pull phase*, where not yet copied pages are transferred when they are requested in the new VM. The drawback of this approach is that it forces us to monitor all pages to detect the ones that have been dirtied between our copying rounds. We also have to decide whether a page changes too frequently and should be copied in the stop-and-copy phase. Nevertheless it is an alternative to the approach we used.

To resume a VM the user sets up a request specifying the name of the image he wants to resume. The procedure then works similar to booting. The data is collected by the hypervisor and, after that, the virtual machine is resumed.

3.4 Exploiting HPC interconnect capabilities

Our approach utilizes the remote direct memory access (RDMA) feature provided by Blue Gene/P to handle the whole data transfer. Hence, it is the critical feature we must have in order to attain a scalable system. In an interconnect fabric that features RDMA one computer can access another computer's memory without involving either one's CPU or operating system. To achieve this, RDMA supports *zero-copy* to avoid copy operations of data between application and operating system. This allows a higher performance and a lower networking overhead which results in a higher network throughput because the application is able to bypass the kernel to transfer data. This is especially true for large data transfers, which is exactly what we do. An example for a network architecture that supports RDMA is InfiniBand [27].

InfiniBand's RDMA feature offers two main message types: RDMA write and RDMA read. An RDMA read message that is received from the server causes it to generate a reply and send the data back to the address specified by the client. An RDMA write transfers data from the client to a previously allocated buffer at the server side.

The way we use RDMA to store our incoming data is depicted in Figure 3.3. There are three different queues: a send queue (SQ) where read and write messages are posted and a receive queue (RQ) that contains buffer information for incoming messages. The third queue is the completion queue (CQ). It allows the user to retrieve completion status of his read and write requests. Every queue is handled in a first in first out manner.

At the beginning the client notifies the server that data shall be stored on the server side. The server then checks whether the request is valid and, if it is, allocates a buffer of sufficient size if necessary. After that the server first places a message item into its receive queue to announce the location of the previously

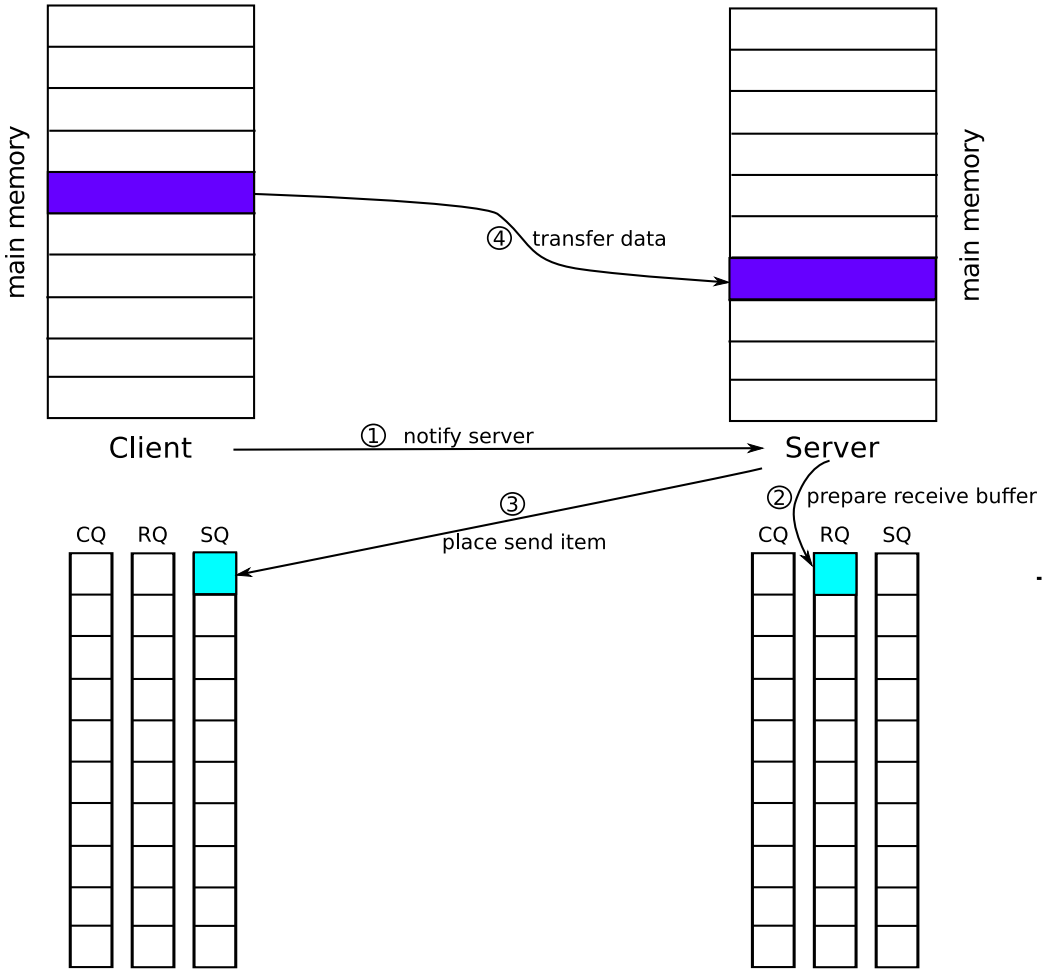


Figure 3.3: RDMA data transfer from the client to the server.

created receive buffer. Another message item is placed into the send queue of the client. The data transfer starts once every previously inserted item is processed.

The transfer can be completely handled by the network adapter, so that the load on the CPU is decreased and prevents it from becoming a bottleneck. The hardware also automatically queues incoming requests into the proper queues and processes them as resources become available.

Chapter 4

Implementation

In this chapter we will give an overview of our implementation and describe some of the issues we were confronted with. We implemented our solution on top of Blue Gene/P and several other projects that provide a cloud environment on top of this supercomputer. At first we name these projects and summarize them. After that we will present our solution that is build extending these projects.

4.1 Implementation Platform

4.1.1 Blue Gene/P

Our implementation is based on Blue Gene/P, an architecture for high performance computing mainly developed at IBM. A Blue Gene/P rack consists of at most 1024 compute nodes, each containing four PowerPC 450 cores operating at 850 MHz and two or four GB of main memory, giving us a total of 4096 cores and two or four TB of memory per rack. The node-to-node communication is handled by three networks, a 3D torus network, a collective network and a global barrier network. The nodes can be viewed as general purpose computers with an enhanced FPU to handle super-computer workloads.

A DMA engine handles packages injection and reception in the torus network. A message is represented by a descriptor item that contains information such as the message type, destination, length, and start address of the message. The descriptors are placed in FIFOs of arbitrary length that are managed as a producer-consumer queue in a circular buffer.

For additional information on the Blue Gene/P system, we refer to [13] and [2].

4.1.2 KHVMM

For virtualization we use a hypervisor running on top of an L4 microkernel [18]. It consists of a small μ-kernel with virtualization capabilities and a user-lever VMM that manages virtual BG/P cores, memory and interconnects, but also allows native applications to run directly on top of the μ-kernel [29]. The VMM functionality is hereby implemented as a user-level application, decoding and emulating the sensitive instructions the guest kernel traps into L4. The L4 kernel itself merely acts as a message passing interface between VM and hypervisor.

Using L4 as a hypervisor allows the enforcement of admission control to the hardware, which can't be guaranteed by Blue Gene/P because its current security model is that of a single application and all nodes are trusted among each other.

4.1.3 Kittyhawk Linux

Kittyhawk [5] is a project that aims to enable generic, heterogeneous workloads on the Blue Gene/P. Therefore the authors use a regular Linux kernel, but add device drivers for the collective network, the torus network, the interrupt controller and a console driver, which sends console messages over the collective network. They also add a virtual Ethernet driver that emulates Ethernet on top of BG/P's torus and collective networks and provides a full TCP/IP based network stack. Additionally, the authors add support for jumbo packets for an Ethernet adapter upon the internal networks. The kernel can either be booted natively on a compute node, or virtualized on top of KHVMM. Additionally, since it is a regular Linux kernel, it can host any Linux application that can be compiled for the PPC architecture.

Kittyhawk is our operating system of choice for both the servers running our storage application and the management VM.

4.1.4 Distributed Storage System

Our solution is based on the distributed cache described in [15]. The cache uses the main memory of a set of nodes as storage (these nodes are called *storage nodes*). Each node that is not a storage node can access a storage node's memory through the hypervisor which handles the data transfer. Sending and collecting data is implemented with memcached `memget` and `memset` methods. The hypervisor also initiates the boot process.

The booted images are split into chunks and are evenly distributed among the storage nodes using their names as hash-values. To load images, the remote DMA capabilities of BG/P torus interconnect are used to improve scalability.

4.2 Overview

Implementing our solution falls into two sub-tasks. First, we had to extend the memcached implementation on the server side to enable communication with the hypervisor. Second we had to provide a command to issue a snapshot at the VM on the client side and to add code to the hypervisor so that it can send data to the server. In the following, we will first describe the communication between client and server and continue with a more detailed description of the changes we made at server and client.

4.2.1 Communication between Client and Server

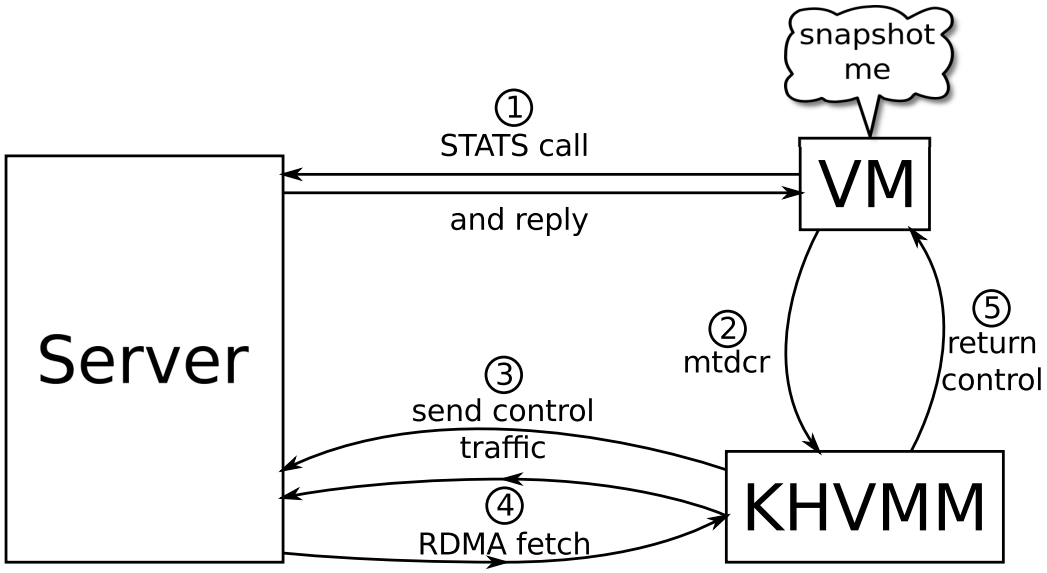


Figure 4.1: Interaction between client and server to perform a snapshot

The whole communication between server and client is depicted in Figure 4.1. First the virtual machine uses a STATS call to fetch the information that is necessary for remote DMA from the server using a TCP connection (①). This connection is processed by the virtual Blue Gene Ethernet driver implemented in the Linux kernel. Afterwards, the VM uses virtual Device Control Registers (see Section 4.2.3) to push the information the VM fetched with the STATS call into the hypervisor (②). The VM automatically pauses after the last transfer so that the VM's state can be copied. The hypervisor therefore first sends a control message containing physical address and size of the data chunk (③). The server then allocates a buffer, creates a descriptor item, and places it into the injection FIFO of

the node the hypervisor is running on. The clients RDMA handler then processes the actual data transfer without using the CPU (④). After the last chunk is copied, the hypervisor returns control to the VM, which then continues execution (⑤).

The different communication methods and the clients and servers are described in the following sections in more detail.

4.2.2 The Server Side

On the server side we use memcached as our storage management system [9]. On start-up, memcached allocates memory that it uses as storage, initiates its working threads and opens a socket for communication. It then enters a loop and waits for incoming requests.

We enhanced memcached by adding a packet socket [19]. This socket behaves like the ordinary sockets used from the traditional memcached. So, the packet socket can be used to transmit control traffic between client and server, too. The difference is, that packet sockets work at the device driver level (OSI Layer 2) whereas STREAM Sockets work at the transport level. Thus, a packet socket allows the injection and reception of any network data that is embedded in a valid Ethernet frame. This socket type is necessary since the microkernel we use does not yet have a TCP/IP stack (see Section 4.2.3 for further details).

Our packet socket imitates the behavior of the ordinary streaming sockets memcached uses. It first waits for an incoming connection. We had to create an extra working thread because libevent does not support packet sockets. Once a packet is received, the event status is updated and the received message is decoded. If data shall be stored, an RDMA fetch is issued. The socket then waits for other incoming packets.

A disadvantage of a packet socket is that it receives every packet that is sent to the network interface the socket is listening on. Although packet sockets are normally used just because of this behavior (for example tcpdump [30] uses packet sockets to dump network packets) it is an unwanted feature in our case since we have to filter the incoming packets. For this purpose we created a particular header to detect packets that are addressed to memcached. These packets then are processed from the worker thread while other packages are simply dropped.

4.2.3 The Client Side

The client side can be divided into the virtual machine and the hypervisor running underneath. We provided the virtual machine with a mechanism to save its current state into our servers and put the hypervisor in charge of the necessary data transfer.

virtual machine

We implemented a snapshotting routine that we offer the virtual machine. If the VM wants to save its current state, it just executes the appropriate command, entering the name of the snapshot and the IP addresses of the available servers as the only arguments. Our program then first gathers the necessary information about the servers like their torus coordinates, the sizes and the offsets of their hashtables (number ① in Figure 4.1). Since this information rarely changes, this has to be done only once.

After all needed information is collected, the application sends it to the hypervisor that handles the actual data transfer.

In order to communicate with the hypervisor, we implemented a virtual device that emulates a set of device control registers (DCRs). Because the virtual machine is an ordinary user-lever program from the hardware's view, each access to this device registers causes a privilege exception that is caught by the hypervisor since it is registered as the faulting handler for this device. The hypervisor then handles the device access and returns control to the faulting VM.

Normally a pointer to a data structure will be passed in the DCRs because they are only 32 bits in size. A problem occurred using this method because the VM normally only sees guest virtual addresses. To solve that we use the bigphysarea driver from [20] that we added to the Linux kernel. This driver allows the allocation of a guest physical memory chunk and to obtain its guest physical address. The hypervisor can then compute the corresponding address in its address space and copy the data if necessary.

Because calls to the virtual device registers are blocking, the virtual machine is automatically paused at its last call so there is no special command the hypervisor has to issue to accomplish this.

Hypervisor

When the hypervisor receives the name of the snapshot it immediately starts the snapshotting procedure. Therefore it first fetches the hashtables from the available storage servers using RDMA. After that, the hypervisor splits the data into chunks and sends a control message to the servers containing the physical address and the size of the chunk. The server containing this chunk can be found using the first portion of the calculated hash value.

Normally a TCP message is used for that communication. However our underlying microkernel currently does not have any TCP networking, support nor is there a server that provides support. To solve this problem, we imitated the Blue Gene virtual Ethernet driver used by Kittyhawk. This driver sends Ethernet packets using BG/P's torus and collective network. We build Ethernet packets that we

place into BG/P's torus network simulating an ordinary network transfer. These packets are then received by the packet sockets on the server side who then issue RDMA fetch operations to put the data into the servers.

Using this approach we actually copy the whole communication that is normally based on TCP onto Ethernet packets and packet sockets.

When the server receives the control message, it issues an RDMA fetch operation to get the chunk whose address was specified in this message. After the last chunk is copied, the hypervisor returns control to the virtual machine which then continues execution.

4.2.4 RDMA

The server as well as the client side benefit from the supercomputer's remote DMA capability. Using BG/P's torus interconnect to transmit data significantly reduces CPU load on both the clients and the servers.

There are only two steps that need to involve the CPU. The first one is the calculation of the hash value to gain the information on which server a given chunk shall be stored. This calculation is done by the clients. The servers CPU is used when a control message is received and the servers have to issue an RDMA fetch request and update their hashtables so that they contain the fetched data. As an alternative, the clients can also update the hashtables because they already fetch the hashtables of all available servers.

This approach will further reduce load on the servers CPUs because it will offload CPU usage from the servers and put it into the clients. On the other hand it will yield to several drawbacks that we recognize as being more severe than the additional gain of CPU reduction. First, it will lead to an additional copy operation from the clients to the servers. While the transfer can be handled by RDMA and does not require the servers CPU, the clients will have to update all available hashtables and send them to the correct servers, which will further increase the downtime of the virtual machines. Whereas, when the updates are handled by the servers, they only have to update their own hashtable, which will lead to a distribution of the total CPU time that is needed to update the hashtables among the servers. The second drawback is that we will have to provide a locking mechanism for the hashtables. Otherwise we can not guarantee that the hashtable is currently only updated by one client. But to provide a locking mechanism will induce an additional drawback, because it will slow down the communication in total. These thoughts lead us to the conclusion to implement the update mechanism for the hashtables on the server side.

RDMA on Blue Gene/P works slightly different than described in section 3.4. BG/P uses a so-called *injection FIFO* to queue RDMA read and RDMA write message items. These message items can be directly placed into the injection

FIFO of another node. Since they also contain the address of the receive buffer, there is no need to first put a message into the nodes own receive queue.

Chapter 5

Evaluation

Our implementation and evaluation platform is a single rack Blue Gene/P system located at the Argonne National Laboratory [25]. The BG/P system is equipped with 1024 nodes (4096 cores), 2 GB RAM per node and uses one I/O node per 64 compute nodes.

We made the following changes to allow communication between memcached server and hypervisor: we added a packet socket to memcached that waits for incoming requests and provides the same functionality as the other sockets used. Therefore we use the methods provided by memcached, but change the event handler of the packet socket afterwards. This step is necessary because memcached's event handler uses socket calls that are not supported by packet sockets. Memcached then uses a dedicated thread to wait for incoming messages because we did not receive any data using libevent.

We also extended the hypervisor to communicate with the packet sockets at the server side. If the hypervisor wants to send a message to a server, we create an Ethernet header and insert the MAC addresses of hypervisor and server. Then we put a torus specific header around and create a descriptor item. To start the transfer, we insert the descriptor item into the FIFO. BG/P's DMA engine then handles the transfer.

By exploiting the tight interconnects BG/P offers, we believe that we can reduce the time it takes to snapshot a virtual machine, to reduce CPU utilization and thus reduce the potential thread the CPU on a network based storage poses and to provide a well-scaling alternative to the existing virtual machine management approaches. To back up our assumption, we refer to a research that uses RDMA for an NFS storage [7]. The results of this project show that an NFS storage benefits from RDMA with a higher throughput and a reduced CPU utilization in comparison to non-RDMA interconnects. This is possible because RDMA bypasses the CPU to transfer data and allows *zero-copy*. So the NFS server's CPU is only needed to put the necessary data into main memory, to prepare receive buffers and

to put message items into according queues, if RDMA fetches are issued. Because our whole storage resides in main memory, we only need the CPU to update the servers hashtable and to fetch data. So, in a direct comparison to NFS over RDMA we further reduce the CPU overhead which leads us to the conclusion that we can achieve similar results than the NFS over RDMA approach or even outperform it.

To further support our assumption we want to point to the work of Jose et al. [14]. In their work they show, that RDMA provides a big advantage for memcached. They show that RDMA based set and get operations outperform the traditional Socket transfer by at least a factor of four. The difference gets even bigger the larger the data to transfer is. As can be seen in their results, memcached behaves pretty well on an increased message size, but further experiments need to show that it also scales well when multiple clients access the servers. For that reason we delegate to Jens Kehne's thesis [15]. He shows that loading time increases at most linearly with the number of clients. However, his research is limited to booting multiple VMs, so we still need to evaluate the behavior of memcached on a more distributed workload that not solely consists of get, but also of put requests.

Chapter 6

Conclusion

In this thesis we have presented an approach to enable fast deployment of virtual machines in an HPC environment. It provides the hypervisor with a mechanism to communicate with the storage servers. This is necessary because our hypervisor runs atop of a microkernel that does not provide TCP networking support yet.

Using our communication method the hypervisor is now able to push data into the memcached servers which is the crucial feature to enable snapshotting and migration. Our approach does not use a centralized storage and thus avoids a potential bottleneck. Exploiting the tight interconnects the supercomputer provides we are able to enhance the data transfer and reduce CPU utilization. Additionally our approach is an alternative to the traditional procedure to push data into the memcached servers.

6.1 Future Work

There are some opportunities to further optimize our implementation. First, our implementation would benefit from incremental snapshots. But, since a chunk is an immutable entity once uploaded to the memcached servers, incremental snapshotting would require the possibility to decide in which chunk the changed data is located and to only upload this specific chunk. Additionally the hypervisor would need some monitoring feature to find the pages that change during snapshots.

A second feature would be the online and offline migration of a virtual machine. Albeit our approach can perform an offline migration by first doing a snapshot and then resume the VM on another node, it can definitely be improved when the unnecessary storing step is avoided. Therefore the hypervisor only needs to know the destination node the VM shall migrate to and could then copy the whole VM state using the HPC's interconnect.

To perform an online migration the hypervisor again needs to know which

pages change during the copying rounds, if we follow the same procedure used by Clark et al. [8], and additionally the starting point of the VM's main memory at the destination node. The hypervisor could then successively copy the pages until a consistent stage between source and destination is reached. The VM could then be started at the destination node. As research evidence indicates, online migration would benefit in great extend from the RDMA capability of a supercomputer [11].

Bibliography

[1] scp-wave. http://code.google.com/p/scp-wave/.

[2] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early evaluation of ibm bluegene/p. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 23:1–23:12, Piscataway, NJ, USA, 2008. IEEE Press.

[3] Margaret Allen. Mathematics + supercomputers = big bang explained. http://www.physorg.com/news197816941.html, 07 2010.

[4] Amazon. Amazon elastic compute cloud amazon ec2, 2010.

[5] Jonathan Appavoo, Volkmar Uhlig, and Amos Waterland. Project kittyhawk: building a global-scale computer: Blue gene/p as a generic computing platform. *SIGOPS Oper. Syst. Rev.*, 42:77–84, January 2008.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, and Neugebauer Neugebauer. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating systems principles (SOSP)*, pages 164–177. ACM Press, October 2003.

[7] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. Nfs over rdma. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, NICELI '03, pages 196–208, New York, NY, USA, 2003. ACM.

[8] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[9] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004:5–, August 2004.

[10] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37:193–206, October 2003.

[11] Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K. Panda. High performance virtual machine migration with rdma over modern interconnects. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 11–20, Washington, DC, USA, 2007. IEEE Computer Society.

[12] IBM. *Control Register Bus 3.5 Architecture Specifications*, 01 2006.

[13] IBM. Overview of the ibm blue gene/p project. *IBM Journal of Research and Development*, 52(1.2):199 –220, jan. 2008.

[14] J. Jose, H. Subramoni, Miao Luo, Minjia Zhang, Jian Huang, M. Wasiur Rahman, N.S. Islam, Xiangyong Ouyang, Hao Wang, S. Sur, and D.K. Panda. Memcached design on high performance rdma capable interconnects. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 743 –752, sept. 2011.

[15] Jens Kehne. A distributed cache for fast vm booting in tightly interconnected cloud environments, September15 2011.

[16] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pages 40 – 46, 2002.

[17] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 1–12, New York, NY, USA, 2009. ACM.

[18] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles (SOSP)*, pages 237–250. ACM Press, December 1995.

[19] Linux Programmers manual. *Packet Sockets*, 08 2008.

[20] Pauline Middelink. Bigphysarea. `http://www.polyware.nl/~middelink/En/hob-v4l.html#bigphysarea`.

[21] Justin Moore, Ratnesh Sharma, Rocky Shih, Jeff Chase, Rakant Patel, Parthasarathy Ranganathan, and Hewlett Packard Labs. Going beyond cpus: The potential of temperature-aware solutions for the data center. In *In Proceedings of the Workshop of Temperature-Aware Computer Systems (TACS-1) held at ISCA*, 2004.

[22] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. BlobSeer: Next Generation Data Management for Large Scale Infrastructures. *Journal of Parallel and Distributed Computing*, 71(2):168–184, February 2011.

[23] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds. In *The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2011)*, San José, CA, États-Unis, June 2011.

[24] Bogdan Nicolae, Franck Cappello, and Gabriel Antoniu. Optimizing multi-deployment on clouds by means of self-adaptive prefetching. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 503–513. Springer Berlin / Heidelberg, 2011.

[25] U.S. Department of Energy. Argonne national laboratory.

[26] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36:361–376, December 2002.

[27] Gregory F. Pfister. An introduction to the infiniband architecture. IBM Enterprise Server Group.

[28] Gilad Shainer, Tong Liu, John Michalakes, Jacob Liberman, Jeff Layton, Onur Celebioglu, Scot A. Schultz, Joshua Mora, and David Cownie. Weather research and forecast (wrf) model performance and profiling analysis on advanced multi-core hpc clusters. *Linux Cluster Institute*, 2009.

[29] Jan Stoess, Jonathan Appavoo, Udo Steinberg, Amos Waterland, Volkmar Uhlig, and Jens Kehne. A light-weight virtual machine monitor for blue gene/p. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 3–10, New York, NY, USA, 2011. ACM.

[30] Craig Leres Van Jacobson and Steven McCanne. tcpdump. `www.tcpdump.org`.

[31] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev.*, 39:148–162, October 2005.

[32] Romain Wartel, Tony Cass, Belmiro Moreira, Ewan Roche, Manuel Guijarro, Sebastien Goasguen, and Ulrich Schwickerath. Image distribution mechanisms in large scale cloud providers. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 112–117, Washington, DC, USA, 2010. IEEE Computer Society.