Karlsruher Institut für Technologie

A distributed cache for fast VM booting in tightly interconnected cloud environments

Studienarbeit
von

Jens Kehne

an der Fakultät für Informatik

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

Bearbeitungszeit: 15. Juni 2011 – 15. September 2011

Abstract

In modern cloud computing environments, elasticity is regarded as one of the most important features. Part of this elasticity is the ability to boot virtual machines quickly to accommodate suddenly emerging workloads. However, booting large numbers of VMs simultaneously in a scalable way is a difficult problem. If the file system images used for booting are stored in a central location, it will most likely become a bottleneck. Therefore, a distributed solution is necessary. In this thesis, we present a new solution for storing kernel and ramdisk images, as well as loading these images into newly created VMs efficiently. Our approach combines a fast and scalable distributed storage system with the high bandwidth and low interconnect latency of a BlueGene/P supercomputer. The client program runs directly inside the hypervisor, thus avoiding virtualization overhead and allowing it to access the memory of the newly created VM directly. We make use of the remote DMA facilities offered by BlueGene's interconnect to minimize the load of the storage nodes, mitigating the potential bottleneck they pose. The resulting system is both fast and scalable. It can boot large numbers of VMs simultaneously with low delay.

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 world of computing, cloud environments are becoming ever more important. The flexibility to request – and pay for – only the resources needed can be a significant economic advantage for customers. To achieve such flexibility in a cloud computing platform, one key feature it needs to offer is fast provisioning of new virtual machines. So far, this capability is limited on most platforms – requests for new virtual machines usually take minutes to be served. One reason for this is a lack of scalability in the backing storage system. Booting hundreds or even thousands of virtual machines causes a load spike that turns any centralized storage system and any slow interconnect into a bottleneck.

Recently, there have been attempts to host a cloud platform on a supercomputer, namely a BlueGene/P system, rather than on commodity hardware [3, 23]. Using a supercomputer has several advantages: First, the cost per processor is significantly less than using commodity hardware [8]. Second, supercomputers (and especially BlueGene systems) are more energy efficient (in terms of watts per cycle). Third, and most importantly for our purposes, supercomputers offer specialized (and often proprietary) interconnects which offer both high throughput and low latency. Especially due to the properties of these interconnects, supercomputers are ideally suited for hosting extremely flexible compute clouds.

In this thesis, we present a distributed storage system meant to complement an existing, disk-based storage. Our system is capable of serving large numbers of boot requests simultaneously. The data is stored fully distributed in the RAM of a set of storage nodes. Clients can read from the storage nodes in parallel, and the system does not use any central components such as master file tables, thus avoiding a potential bottleneck. Our solution also makes efficient use of the remote DMA capability of the underlying interconnect to read data from the storage nodes completely without intervention from the server's CPU, pushing as much computation as possible into the clients. Also, the number of storage nodes can be increased arbitrarily to increase both bandwidth and storage space.

All these properties allow our system to provide the data necessary for efficiently booting arbitrary numbers of virtual machines in parallel.

There are few projects addressing the problem of fast and scalable booting to this date. **SnowFlock** and the **Potemkin** virtual honeyfarm both allow swift creation of new virtual machines, but they are both limited to cloning running VMs. There are however storage systems available which allow scalable distribution of data. **Distributed databases** generally distribute load efficiently, but their ability to perform complex operations on the retrieved data is unnecessary for our application. **Distributed file systems** – which are currently used for booting in the cloud – carry less extra weight than databases, but most of their features are still redundant, and they generally rely on disks as backend storage. **Key-value-stores** are far simpler, providing exactly the functionality needed for fast booting. They scale well, and they usually do not rely on a specific backend storage. They cannot be booted from directly, but as we show in this thesis, an efficient image distribution system can be built on top of them.

Chapter 2

Related Work

To date, not much research has been done on the subject of mass booting virtual machines. **SnowFlock** [17] is somewhat similar to our solution in that in allows creating large numbers of VMs quickly. However, it can only clone already booted VMs to different compute nodes. It uses a lazy copying (called *Memory-On-Demand*) and fetch-avoidance heuristics to reduce the amount of state that has to be copied.

SnowFlock is different from our solution mainly in that it does not allow static storing of VM state. There must always be a running VM that can be cloned, which implies that all data originates from one node. Even if the amount of data is relatively small, the parent node will eventually become a bottleneck if the number of child VMs becomes large enough. SnowFlock's memory-on-demand scheme also results in data being copied even after the child VMs have booted, which makes the overall system less predictable.

The **Potemkin** virtual honeyfarm [27] uses similar techniques to create large numbers of virtual honeypots on demand with low latency. Potemkin does use statically stored images of VM state however, it is also limited to restoring the state of previously booted VMs. It also uses copy-on-write, physically sharing unmodified memory pages between multiple VMs. However, the reference images being cloned are stored in the RAM of the compute nodes. Thus, copies of an image can only be cloned to the same host on which they are stored. This effectively limits functionality to one application. However, it may be possible to combine their delta virtualization technique with our storage system to distribute snapshots of one VM to multiple nodes.

The key to efficiently booting large numbers of virtual machines in parallel is the ability to store and, most importantly, distribute data in a scalable way. While few solutions exist for mass booting itself, the topic of distributed data storage has been extensively studied. To our knowledge, there are three main types of distributed storage systems available: Distributed database systems, distributed

file systems and key-value stores.

Distributed database systems offer efficient storage and retrieval of distributed data. However, they are optimized for complex filtering and processing operations on large amounts of data (e.g select, join). They also have to support complex query languages like SQL to describe these operations. Since our solution only touches one bucket at a time, it cannot use these features, but they still add extra latency to the system (e.g. SQL command parsing).

Mariposa [24, 25] is an example of a distributed database system. Its main advantage is that, unlike traditional distributed database systems, it does not use a central query optimizer. Its focus lies on distributing both data and query processing efficiently. It employs an economic model in which the storage nodes buy and sell both processing time and data items. This results in efficient, distributed processing of complex queries. Unfortunately, our system would not benefit from this since it does not use such complex queries. Moreover, data items move between storage nodes frequently. While this helps with optimizing query processing in a WAN environment, the extra load it places on the network would probably have more adverse effects in the tightly-coupled environment we are targeting.

Distributed file systems, like pNFS [21], GPFS [22] or the Google File system [12] do not need to optimize complex queries. They therefore do not add add the same latency as databases.

pNFS is an extension to the well-known NFS. It preserves the entire NFS interface, but allows for parallel access to multiple storage sites. It still uses a central server, but that server only holds metadata (i.e. the file system catalog), while the data lives on storage devices which the client can access directly. By taking the server out of the datapath, pNFS eliminates one of the tightest bottlenecks in NFS. However, it is still necessary to keep the catalog in sync with the actual data storage. If the number of clients is large enough, the server is still a bottleneck, albeit a wider one. That said, there have been attempts to leverage remote DMA for pNFS [6] to solve this problem. The same might be possible on BlueGene.

GPFS is a distributed file system specially designed for supercomputers. In GPFS, almost everything is distributed. The only central component is a lock manager for synchronization. To prevent the lock manager from becoming a bottleneck, GPFS features fine-grained (byte-level) locking to reduce conflicts, and most conflict resolution is done between the conflicting clients, without involving the lock manager. GPFS is currently in use on many production supercomputers, including the one we developed our system on.

The Google File System (GFS) is another example of a distributed file system. Like pNFS, it uses a central server to hold metadata, though it is designed

to minimize access to that server. It is also designed to run on commodity hardware, putting a focus on fault tolerance. Both data and metadata are replicated throughout all storage nodes, and any crashed node (both storage and metadata) can be restarted seamlessly. On the downside, it uses a heartbeat protocol to keep the metadata- and storage nodes in sync, which requires frequent communication.

To date, file systems like these are the method of choice for sharing files and folders across multiple (virtual) machines. They are also often used as root file systems for disk-less systems. However, if the number of virtual machines becomes large enough, any shared file system will eventually become a bottleneck – especially when multiple virtual machines boot in parallel, as booting creates high traffic on the file system. It is therefore desirable to have a local file system in each virtual machine, mounted in RAM if no disk is available. To achieve this, an image of the entire root file system, stored in a simpler storage system, can be distributed at boot time. Network file systems containing shared data can then be mounted as necessary after booting is complete.

Key-value-stores, in contrast to distributed file- or database systems, lack all the unneeded complexity. The most well-known example is Amazon's Simple Storage Service (S3) [18]. Little is known about its implementation, but performance data indicates that it does not meet the latency requirements for our project (Amazon's EC2, which uses S3 as a backing store, typically boots VM instances in a matter of minutes). Other examples include Amazon's Dynamo [10], Yahoo!'s PNUTS [9] and Google's Bigtable [7].

PNUTS is a data storage service offered by Yahoo. It is far less complex than full-blown file- or database systems. However, it targets a global scale, with storage sites being geographically separated. Consequently, it replicates data across multiple sites to provide uniform access latencies for all users and fault tolerance. Replicas need not necessarily be consistent across all locations, so locating the most up-to-date copy of a record might take a while since communication with multiple sites may be required.

Dynamo puts even more focus on high availability than its competitors. It also prioritizes writes over reads. Writes to Dynamo are never rejected, and any conflict resolution that might be necessary is done during reading. One advantage of dynamo is that it is fully decentralized. There is no shared directory; instead, a variant of consistent hashing is used to map data items to storage nodes. Data items are replicated across multiple nodes using and eventual consistency model.

Google's **Bigtable** targets more tightly interconnected platforms. It uses a master server to assign data to storage nodes, but the design minimizes communication of clients with the master to prevent it from becoming a bottleneck. The location of data is stored in two layers of metadata, which are saved in the stor-

age nodes. A distributed locking scheme called Chubby is used to ensure consistency. Bigtable itself does not provide any replication or fault-tolerance, but instead relies on GFS as underlying storage. As a consequence, it also inherits the disadvantages of a full-blown file system.

Generally, key-value stores seem like a natural choice to use as backend storage for our system. They lack the extra complexity of file- or database systems, leaving only the features that we need. Because of their rather simple architecture, they are fast and scalable. However, the solutions mentioned above were designed to operate on persistent storage (i.e. hard disks), while our goal is to build a cache to complement an existing, persistent storage system. Also, the wide (global scale) distribution and fault-tolerance found in existing systems are unnecessary for our project, since in a supercomputer, everything is tightly connected and failures – especially network link failures – are rare compared to globally distributed systems.

Chapter 3

Design

In order to boot large numbers of VMs quickly, the system needs to achieve high scalability, as well as high network speed and low latency. Our approach achieves scalability by distributing file system images between a number of storage nodes. Both low latency and high speed can be achieved by using BlueGene's torus interconnect efficiently.

3.1 System overview

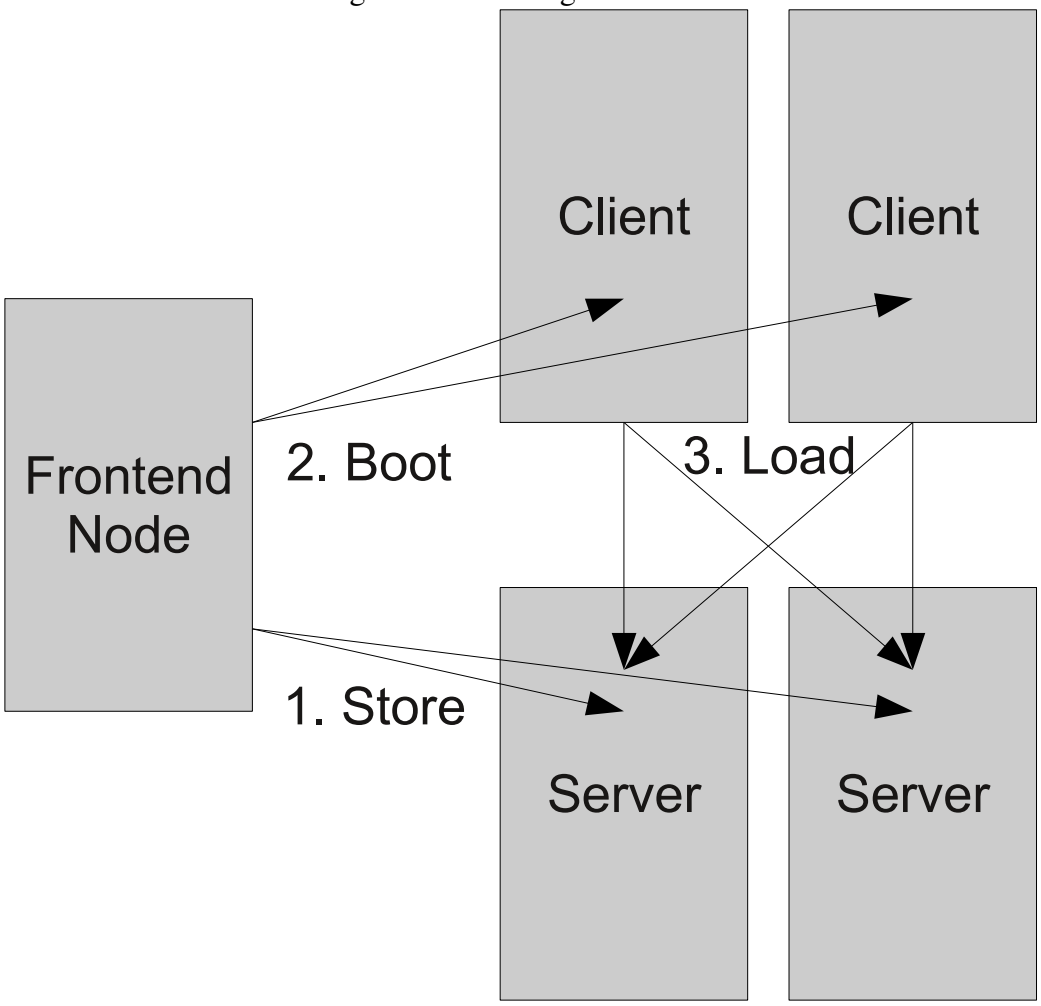
Our system works by designating a subset of the compute nodes as *storage nodes*. A large portion of each storage node's RAM is used exclusively as a storage media. The storage nodes run a server application, which accepts requests from the client nodes, on top of a native (i.e. non-virtualized) Linux system.

A client node can be any node that is not a storage node. The client nodes run an application consisting of two parts: The first part, running on top of a Linux-based *management VM* – similar to XEN's dom0 [4] – handles communication with the user, while the second part, which is integrated into the node's hypervisor, handles the data transfer from the storage nodes to the new VM's memory. Since the second part is integrated into the hypervisor, it can also initiate the boot process of the new VM once the transfer is complete.

For the data transfer, our system leverages the tight interconnects in the supercomputer, and especially its remote DMA capability. The use of remote DMA considerably reduces the CPU load on the storage nodes, as their network devices can send the requested data autonomously, without CPU intervention.

The basic procedure is depicted in Figure 3.1. The customer or the cloud administrator first stores one or more images in the storage nodes. Each image is stored in chunks, which are distributed across all servers (see Section 3.2). At a later time, the customer issues a boot command to a number of compute nodes.

Figure 3.1: Booting a new VM



The compute nodes then load all chunks belonging to the requested image from the servers. It is possible for multiple clients to load different chunks in parallel from different servers.

3.2 Image distribution

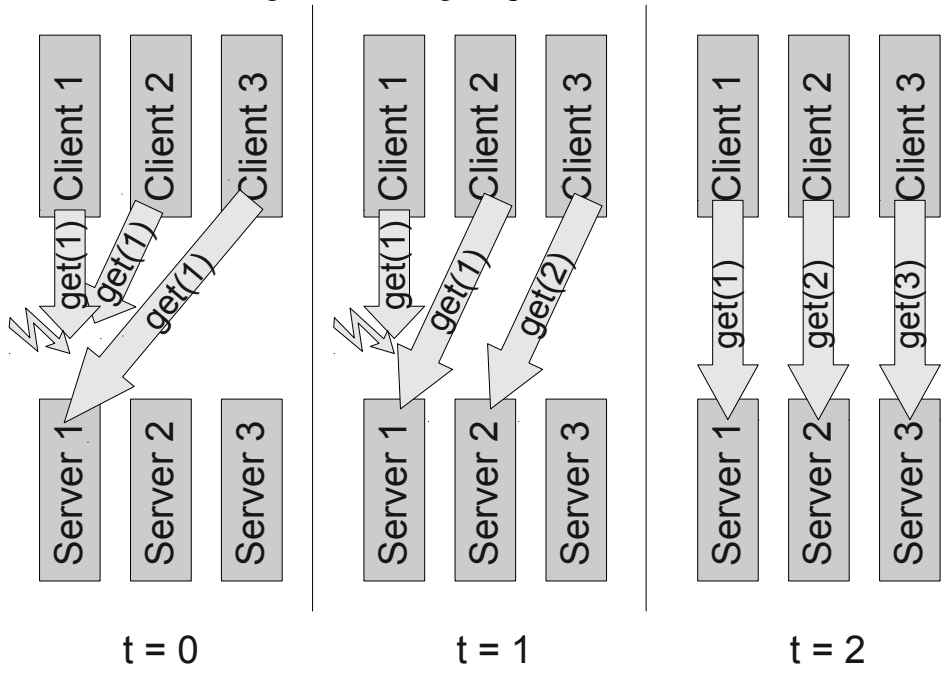
The images to be stored are split into fixed-size chunks, which are then uploaded to the storage nodes. Each chunk is assigned a unique key, which is derived from the file system image's name, and we then compute a hash of this key. The first portion of this hash is used to determine the storage node responsible for each chunk (each node is responsible for a range of possible hash values). Each node maintains a so-called *hashtable*, which contains the memory location for each chunk. The remaining portion of the hash is used as an index into this hashtable.

This approach has two main advantages. First, it is possible for a client to load and store image chunks knowing only the name of the image. The clients can compute all required information by themselves, which means that there is no need for a central directory service. Second, since the hashes are pseudo-random, the chunks are automatically spread evenly across all storage nodes. As a consequence, i) the available storage space is used efficiently, and it can be expanded simply by adding more storage nodes, and ii) the traffic caused by large numbers of clients loading the same image is also spread evenly across all participating storage nodes.

Figure 3.2 shows the behavior of our system when multiple clients start loading the same image simultaneously. Initially (at t=0), there is congestion when all clients try to request the first chunk. The server can only serve one request at a time, so the remaining clients have to wait. Once the request is served, the server processes the next request, while the client that just received the first chunk requests the second one from the second server (t=1). In time (at t=2), all three clients are loading their chunk in parallel. This model is highly idealized, but our system exhibits a similar behavior whenever there is congestion to resolve. Since the chunks are also spread evenly across all servers, the load (i.e. the number of requests per time) is spread evenly across all servers, too.

There are also downsides to this approach. The first one is that storage nodes cannot be easily added or removed, because most chunks would have to be moved in order to restore the uniform distribution. Fortunately, our approach is not tied to a specific hash function, so we can use techniques like consistent hashing [15] to solve this problem. Another issue is that each image is, once uploaded, an immutable entity. In our current design, any modification to an image requires a complete re-upload. It is also not possible to share identical chunks between multiple images. There are ways to solve this problem [14, 16], but they are outside

Figure 3.2: Image requests over time



the scope of this project.

3.3 Mixed hypervisor/guest execution

Booting a new VM cannot be done entirely in the management system, as the management system – which is itself a VM – cannot access the memory of other VMs directly. On the other hand, it is also not possible to handle the entire process inside the hypervisor, as the hypervisor lacks an interface for the customer to start the boot process. We therefore chose to employ a mixed approach: Some parts of the process, like communication with the customer, execute in the management VM, while others, like access to the new VMs memory, are handled in the hypervisor. However, this implies the need for an interface between the management system and the hypervisor.

Once the management system has received a boot command from the customer, the actual data transfer can be handled either in the management VM or in the hypervisor. We therefore explored two methods for booting. The first method is simple: The management VM loads a chunk of an image, then sends a pointer to the chunk to the hypervisor. The hypervisor then copies the chunk from the management VM into the new VM's memory and returns. The management VM repeats the process until all chunks of the image have been transferred. The sec-

ond method is more complex: We send pointers to the requested images' names to the hypervisor. The hypervisor then handles the entire transfer.

While the second method is far more difficult to implement, it also promises to yield considerable performance benefits. First, the data transfer does not need to pass through the virtualization layer. Second, sending data to the hypervisor involves a guest/hypervisor switch, which is an expensive operation. The first method requires one switch for each loaded chunk, while the second method needs only a fixed number of switches once the transfer starts. For these reasons, the second method appears more promising, which is why we chose to implement it.

3.4 Remote DMA

Our system uses the remote DMA capabilities of BlueGene's torus interconnect to further improve scalability. In general, a torus data transfer is initiated by placing a descriptor item in a so-called *injection fifo*. The network hardware then handles the transfer by itself, while the CPU is free to do other work. For a remote DMA transfer, the data sent is another descriptor item, which is placed in an injection fifo at the remote end. In simple terms, a node A can instruct node B to send a piece of data back to node A. If the request is valid from B's point of view (for example, remote DMA must be explicitly allowed), the network hardware processes the request and sends the requested data without CPU intervention.

Using remote DMA for the data transfer allows the storage nodes to handle large amounts of load efficiently. Most importantly, it prevents the storage nodes' CPUs from becoming a bottleneck by significantly reducing the load on them. Second, the specialized network hardware processes the requests more efficiently then the CPU, resulting in lower latency and a higher network utilization. The hardware also helps with handling overload conditions: If multiple requests arrive at the same time, they are automatically queued in the remote DMA injection fifo, and processed as resources become available. There is even some degree of parallel processing: If two clients place a request with the same server, the server can serve the requests in parallel through separate torus links.

While our implementation is currently highly BlueGene-specific, the underlying concept does not depend on BlueGene. The crucial feature we need is remote DMA, so in principle, our system could use any network architecture that supports remote DMA, like InfiniBand [20].

Chapter 4

Implementation

In this Section, we will give details about how we implemented the design described in Section 3, and the problems we encountered during implementation. We first give some background on the hard- and software we used, before giving an overview of our system that is more specific to this hard- and software. We then present the implementation details of the key concepts of our solution: The mixed guest/hypervisor execution and remote DMA.

4.1 Implementation Platform

4.1.1 BlueGene/P

BlueGene/P is an architecture for supercomputers, mainly developed by IBM. The basic building block of a BlueGene/P system is a *compute node*, which is composed of four PowerPC 450 cores clocked at 850 MHz, either two or four gigabytes of RAM and several network adapters, all integrated to a single system-on-chip. A BlueGene/P *rack* holds up to 1024 of these compute nodes, plus additional I/O nodes. A complete BlueGene/P system can consist of up to 216 such racks. The Compute nodes are connected by a three-dimensional torus network for point-to-point communication, a collective network for broadcast communication, and a global interrupt network. The I/O-nodes feature a 10G ethernet interconnect instead of the torus, which connects them to the frontend nodes. For more details on BlueGene/P, see [1].

4.1.2 khvmm

For virtualization, our system uses a light-weight user-level hypervisor running on top of an L4 Microkernel [23]. The virtualized guest systems run as L4 user-level

processes alongside the hypervisor. Each guest VM runs in its own address space, which isolates each VM's memory from the others. The hypervisor acts as an interrupt handler for the guest systems, allowing it to provide a virtual platform in a trap-and-emulate manner: Whenever a guest VM executes a sensitive instruction, a privilege exception is delivered to the hypervisor via L4's IPC. The IPC message contains guest state – e.g. register contents – related to the fault. The hypervisor can then emulate the sensitive instruction and inject the updated state back into the guest.

Khvmm exposes a virtualized version of each BlueGene/P device to each VM. As a consequence, the guest operating system needs device drivers for BlueGene hardware, especially the network devices. There is no virtual ethernet available from the hypervisor, though the guest systems can emulate ethernet themselves on top of the BlueGene networks [3].

4.1.3 Kittyhawk Linux

We used IBM's Kittyhawk Linux kernel [2] for both the storage nodes and the management VM. The kernel is a regular Linux kernel with some BlueGene-specific extensions. Specifically, it contains additional device drivers for BlueGene's network devices and interrupt controller, a console driver, which sends console messages over the collective network, and a virtual ethernet driver [3], which emulates ethernet on top of BlueGene's torus and collective networks, giving applications access to a full, TCP/IP-based network stack. The kernel can be booted either natively on a compute node, or virtualized on top of khvmm. Furthermore, it can host any Linux application that can be compiled for PowerPC. Both the storage nodes and the management VMs run a minimal Linux installation based on the Kittyhawk Linux kernel, uClibc [26] and BusyBox [5].

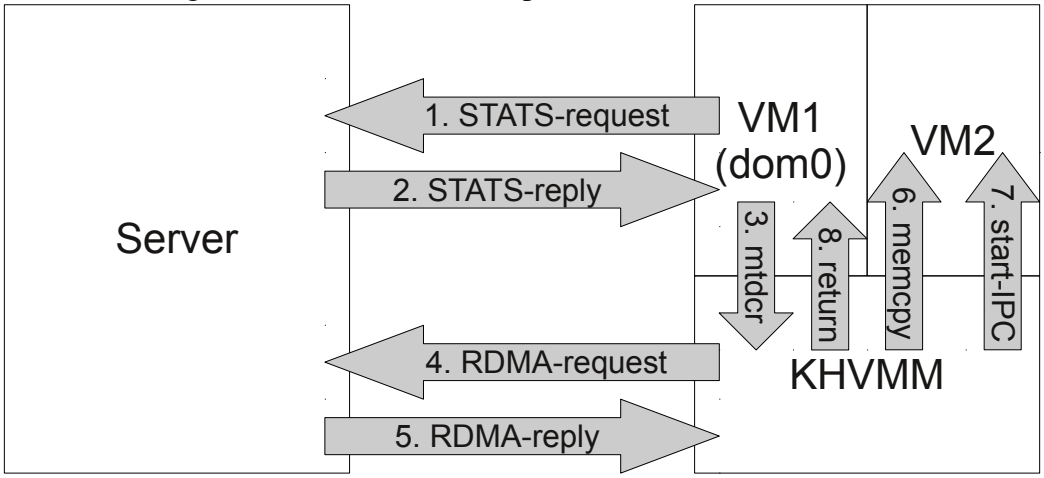
4.1.4 Memcached

Our solution uses a key-value-store called *memcached* [11] as backend storage. Memcached uses hashes of the (user-assigned) keys of the stored objects to track their locations. A part of the hash is used to identify the storage node holding the object, while the other half identifies the exact memory location inside the node. Each node maintains a table of all objects it holds, as well as their corresponding hashes. This table is called the *hashtable*.

4.2 Overview

The basic boot procedure is depicted in Figure 4.1.

Figure 4.1: Communication path used to boot a new VM



Our client application is a Linux program running inside the management VM. It therefore has access to the VM's virtual ethernet driver and TCP/IP-based network stack. At startup, it is given the IP-addresses of all memcached servers. Its first step is then to determine the information necessary for remote DMA, like the torus coordinates of the servers and the offsets and sizes of their hashtables. To retrieve this information, it sends a STATS-command to each server (steps 1+2). The semantics of memcached's STATS-command have been extended to include the required information in the reply. Fortunately, the information rarely changes, so the client application only needs to issue this call once. When the information has been received, the application sends it to the hypervisor in preparation for remote DMA. Details about how this happens are given in Section 4.3.

Once the application has all the required information, it waits for a boot command from the frontend nodes. This command – which can, if needed, be in the form of a web service request – is also sent over a TCP connection. Once this request has been received, the application proceeds to step 3: It issues a call into the hypervisor, instructing it to fetch the requested images and start a new VM. The hypervisor then sends remote DMA requests to the servers to fetch the images (see Section 4.4). Whenever a part of the requested image has been received, the hypervisor copies it to its final location in the new VM's memory. These steps are repeated until all requested images have been transferred.

Once all image data has been received, the hypervisor sets up the VMs remaining state, initializes the virtual devices connected to the VM, and loads some data that cannot be modified by the user, like the u-boot script that determines the VM's IP addresses. It also invokes an ELF loader to load the bootloader into the VM's memory. When this is complete, it sends a startup IPC to the new VM,

which invokes the bootloader.

4.3 Mixed hypervisor/guest execution

In order to offload work from the management system into the hypervisor, we first need a means to send data to the hypervisor. Since both the management system and the hypervisor run as separate L4 threads, the straightforward way would be to use L4's IPC mechanism. L4's IPC implementation stores IPC-related information in CPU registers for speed. Information that does not fit into physical registers is stored in virtual message registers contained in a data structure called the *user-level thread control block*. Unfortunately, threads representing virtual machines do not have their user-level thread control block mapped into their address space. Doing so would be dangerous because the guest kernel is not aware of the underlying hypervisor – it expects the whole (guest-)physical memory (i.e. the L4 thread's address space) to be empty and thus might try to overwrite the UTCB. As a consequence, guest virtual machines are unable to use IPC, so we came up with a different mechanism.

We chose to implement the interface as a virtual device. From the physical hardware's point of view, the VMs are ordinary user-level programs, so any access from a VM to a device control register or a memory-mapped I/O address will cause a privilege exception, which is caught by the hypervisor. The hypervisor then emulates the requested device access and returns control to the faulting VM. Normally, this mechanism is used to virtualize a device which is physically present in the node. However, it can also be used to emulate a device that does not physically exist. We created such a purely virtual device, which represents the hypervisor.

Our device emulates a set of device control registers (DCRs), special I/O registers similar to I/O-ports on x86 architectures. On PowerPC-based architectures, device control registers are connected to a specialized bus with its own address space [13]. They are read and written using special, privileged instructions. The DCRs offered by our virtual device do not have physical counterparts, as they are used only for communication with the hypervisor. Since the DCRs are only 32 bits in size, the usual mode of operation is to put a pointer to a data structure in the guest's memory into the DCR. The hypervisor can then read the contents of that data structure directly.

One problem with passing pointers to the hypervisor is that the client application only knows guest virtual addresses. If those were passed to the hypervisor, it would have to parse Linux' page tables in order to obtain the corresponding guest physical address. Instead, we used Pauline Middelink's bigphysarea driver [19]. This driver allows the client application to request a chunk of memory that is

(guest-)physically contiguous, and to obtain the (guest-)physical address of that chunk. From that address, the hypervisor can easily compute the corresponding address in its own address space, and then – since the chunk is contiguous – copy the chunk to a safe place in one operation if necessary.

4.4 Remote DMA

The torus interconnect's remote DMA functionality allows us to significantly reduce the CPU load on the storage nodes. Prior to the actual data transfer, there are two main steps requiring the CPU: i) calculating which node a given chunk is on, which is done by computing a hash of the chunk's key, and ii) parsing the server's hashtable to determine the location of the chunk in the server's memory. In the original memcached, the second step was done on the server. However, in our implementation, the server's hashtable is available to the client. The reply to the STATS-request issued during startup (see Section 4.2) contains the location and size of the hashtable. With this information, the client can use remote DMA to obtain a copy. Given the hashtable has not changed since it was retrieved, the client can look up the location of every chunk it wants to get and place an appropriate remote DMA request. Thus, the load incurred by the parsing of the hashtable is taken off the server's CPU and pushed into the client.

The remote DMA mechanism itself works by writing special request items into a so-called *injection fifo*. This request item instructs the nodes DMA engine to write a piece of data directly into another node's memory. This piece of data can be another request item, which is automatically written to an injection fifo by the DMA engine of the receiving node. In other words, a node can instruct another node to send it a chunk of data if it knows the physical address of that chunk. In our case, the chunks are pieces of an image, and their physical location is contained in the hashtable. Thus, all data transfer can be done without intervention from the server's CPU.

On the client end, this mechanism implies that the client, after placing its request in the injection fifo, has to wait until a reply from the server arrives. At the moment, waiting is implemented by polling a so-called *receive counter*, a hardware register counting the amount of data received, until it reaches a target value. The target value is easily computed from the size of the chunk, which is contained in the hashtable. This mechanism works well, but further optimization may be possible.

4.4.1 Open problems

We did encounter some problems when we implemented the remote DMA mechanism. However, most of these were caused simply by a lack of hardware documentation. For example, the torus often signals errors as bits in error registers, which are hard to interpret without proper documentation. Therefore, even simple programming errors frequently required extensive debugging.

While our current implementation works well, there are some problems left when the server encounters overload. When it does, certain types of torus errors can occur – the most frequent one being an "injection counter overflow" – which are signaled to the server's Linux kernel as interrupts. Unfortunately, Linux' torus driver cannot properly handle these interrupts, and reverts to calling panic(). Interestingly, reading images from the server is still possible even after the kernel panics, regardless of which error occurred. This suggests that really none of the errors are serious problems. Given proper documentation, we should be able to add working interrupt handlers to the Linux kernel, which would allow our system to handle arbitrary amounts of load.

Chapter 5

Evaluation

5.1 Testbed

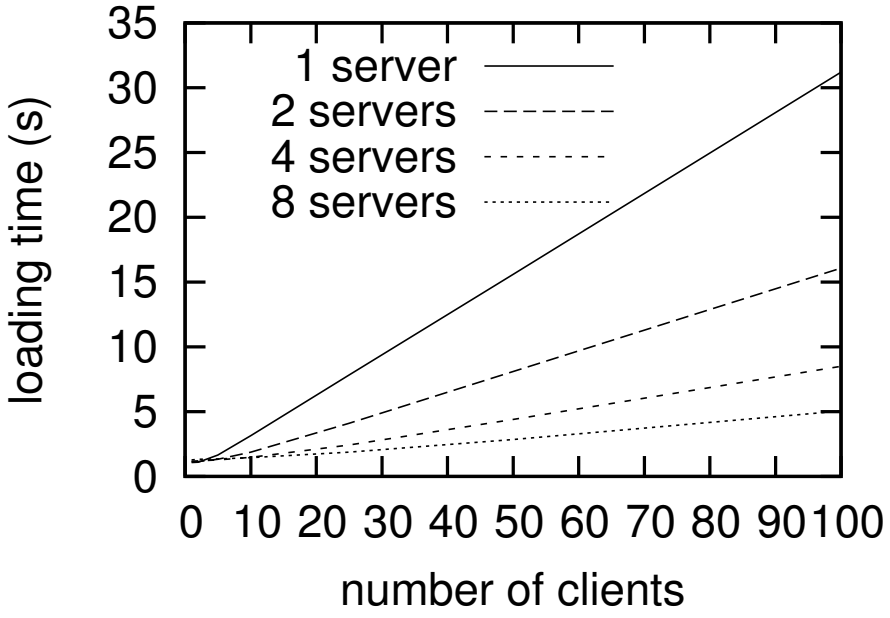
We conducted our experiments on surveyor, a 1-rack BlueGene/P system run by Argonne National Labs. Surveyor features 1024 compute nodes (4096 cores) and 2GB RAM per node. There is one I/O node per 64 compute nodes.

To measure boot times, we disabled the last step of the boot process, that is, setting up VM state and sending a startup IPC. The resulting times can be read as the time until the VM is ready to boot. All remaining work after this point is done locally on the compute node, and is therefore not influenced by any other nodes. Since we are primarily interested in the data transfer, this part is not relevant for the performance of our system; however, it would likely add additional noise to our benchmarks, which is why we chose not to include it in our experiments.

Since the VMs are not actually booting, the images loaded onto the server do not have to be bootable. We therefore simulated images by using large files filled with random numbers.

In order to start the data transfer simultaneously on all clients, we made our client application stop and wait for an incoming TCP connection after the initial setup (i.e. the STATS-call). We then sent the image names to be booted over that TCP connection. To minimize connection overhead, we established and closed a connection to each node without sending an image name before each test run, causing the connection to remain in the TIME_WAIT-state, from which it can be re-established quickly. Using this method, we could get 100 clients to start loading their images within 0.3 seconds. For smaller numbers of clients, the difference was much less.

Figure 5.1: Number of clients versus boot time, 100 MB images



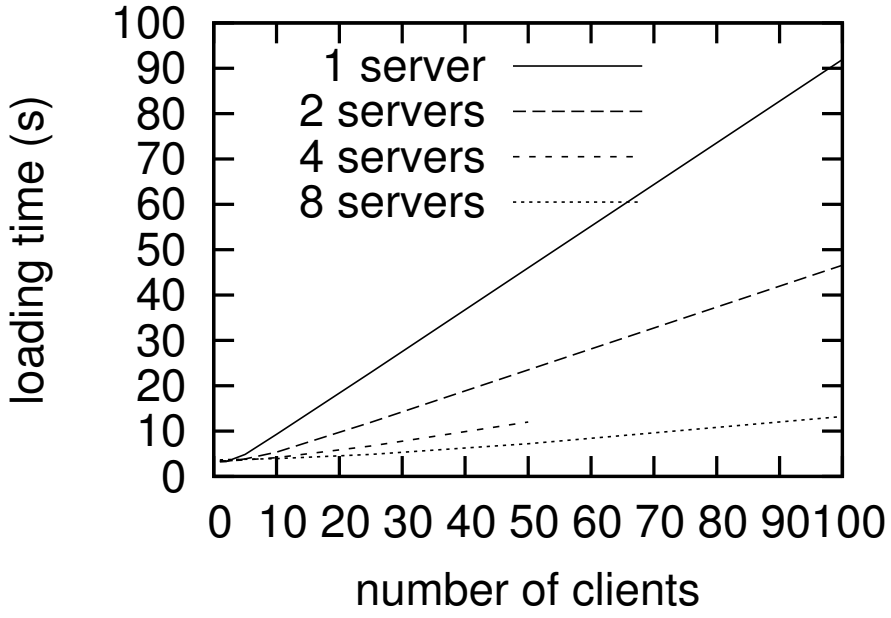
5.2 Test goals

Our main test goal was to prove that our system scales well. We did this by varying the numbers of clients and servers. Ideally, what we wanted to see was:

1. Loading time increases at most linearly with the number of clients
2. Loading time increases linearly with image size
3. Loading time decreases at least linearly with the number of servers
4. Our system outperforms the original memcached

We also investigated the effect of the chunk size on the data transfer. Larger chunk sizes result in fewer chunks per image, and in turn in the chunks' distribution being less uniform. On the other hand, there is some overhead each time a new chunk is requested (e.g. for generating the corresponding remote DMA request), so the total overhead will grow with the number of chunks. We expected to see an "optimal" chunk size, at which these two factors are in balance.

Figure 5.2: Number of clients versus boot time, 300 MB images



5.3 Scalability

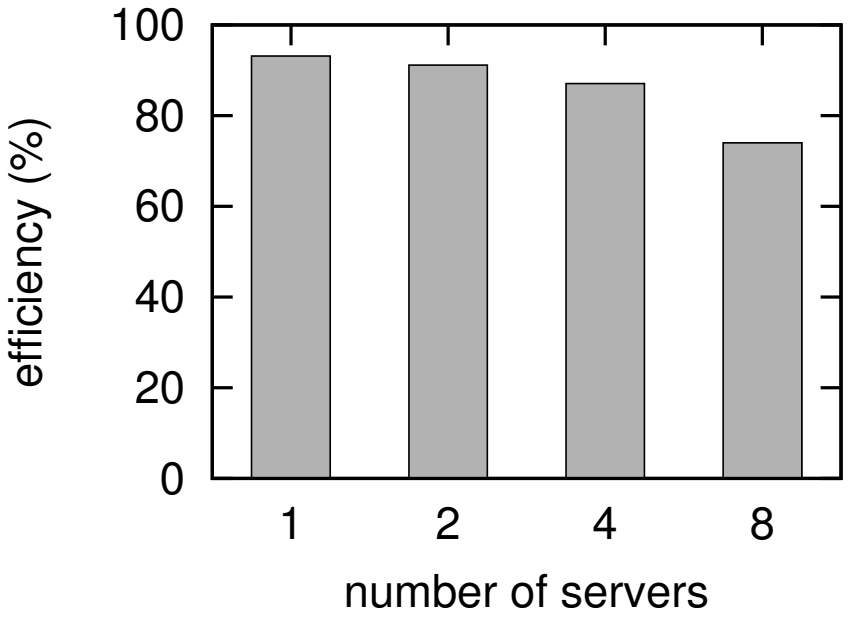
Figures 5.1 and 5.2 show the times needed to boot a number of clients from a number of servers, for varying numbers of clients and servers. Figure 5.1, was measured using a 100 MB ramdisk image file, while in Figure 5.2, we used a 300 MB image file instead. On both occasions, we also loaded a kernel image file of about 1.5 MB, which is close to the size of the real kernel running on the compute nodes. The chunk size used in these experiments was 4 kb. The times shown here are the average loading times of all clients starting simultaneously. The standard deviation of this average was generally below 0.1 second when we repeated the same test multiple times.

These plots show that our system meets the goals specified in Section 5.2.

1. The loading time increases linearly with the number of clients, if there are more clients than servers. If the number of clients is smaller than the number of servers, in increase in loading time is almost negligible (i.e. all clients are served in parallel).

2. The loading times for the 300 MB image is about three times the loading times for the 100 MB image, for the same numbers of clients and servers. This suggests that there is no additional overhead as the image files grow.

Figure 5.3: Torus efficiency

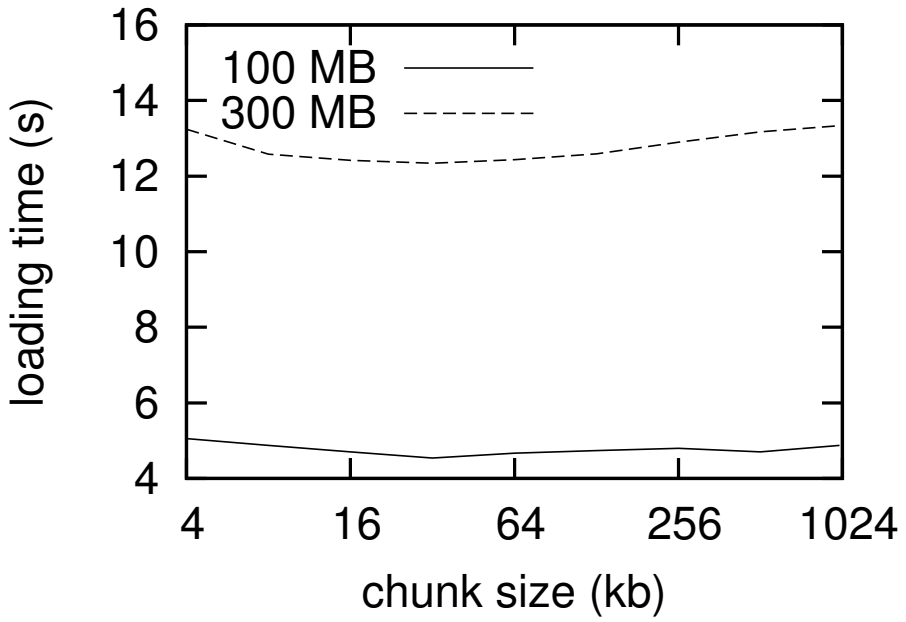


3. For up to 4 servers, the loading time decreases almost linearly with the number of servers. When increasing from 4 to 8 servers, the speedup is about 1.7. This suggests that torus congestion is starting to dominate.

To show the effect of congestion more clearly, we calculated which fraction of total torus bandwidth our system achieves. We assumed that only one torus link per server is active at a time, which makes the peak bandwidth n times the bandwidth of a single link, where n is the number of servers. To calculate the bandwidth during load, we divided the total amount of data transferred (i.e. number of clients times image size) by the loading time for the same number of clients and servers.

Figure 5.3 shows the results for 50 clients and a 300 MB image. It shows that the efficiency is decreasing with the number of servers. A possible reason is that servers are currently placed adjacent to each other, which probably results in them competing for the same torus link frequently. To further investigate the effect, we would have to repeat the experiments with the servers placed at different (i.e. non-adjacent) locations. Unfortunately, choosing the torus locations of the servers is not easily possible as of yet.

Figure 5.4: Chunk size versus boot time



5.4 Chunk size

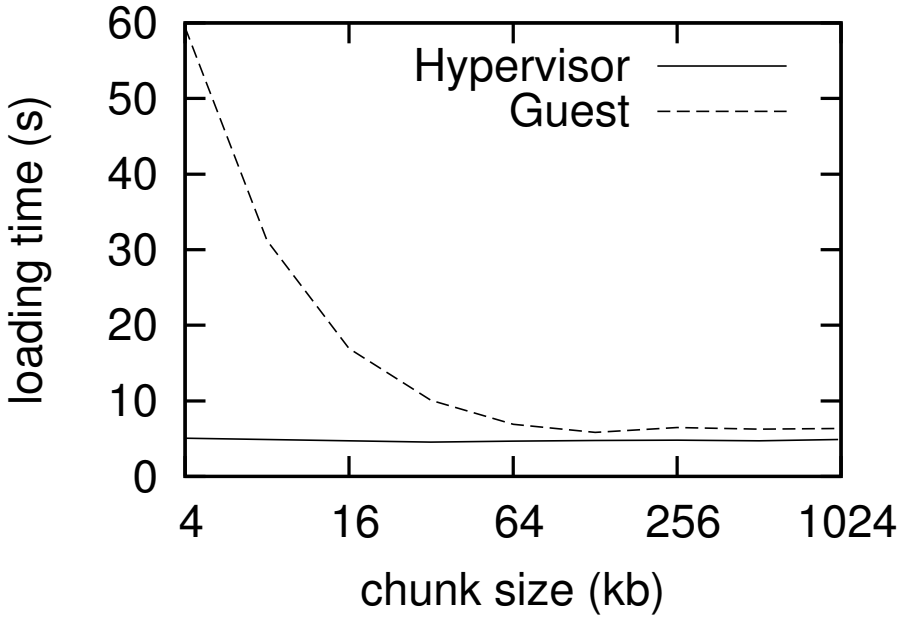
Figure 5.4 shows the effect of the chunk size on the total loading time. To measure it, we used 100 clients and 8 servers. Figure 5.4 shows the results for both 100 MB and 300 MB ramdisk images. As we expected, there is a single chunk size that appears to be optimal for both image sizes, namely 32 kb. Note that the speedup achieved by changing the chunk size is 7.8% when increasing the chunk size from 4 to 32 kb for a 300 MB image, which is quite significant.

As mentioned before, increasing the chunk size (and thus decreasing the number of chunks) can result in a less uniform distribution of the chunks, which would negatively affect load balancing. However, this effect is not significant as long as the number of chunks is significantly greater than the number of servers. There might however be a negative effect if chunk size would approach the image size. Fortunately, for 32 kb chunks, this is unlikely to happen.

5.5 Comparison

To find out whether implementing the data transfer in the hypervisor yields any benefit, we compared it to an alternative version which implements the data transfer inside the management VM. In this version, the client application uses the

Figure 5.5: Comparison between hypervisor- and guest-based transfer



virtual torus of the management VM to fetch chunks, and then passes pointers to these chunks to the hypervisor. The hypervisor then copies the chunks from the management VM to the new VM. This is the first method we described in Section 3.3.

Figure 5.5 shows the results for 100 clients, 8 servers, a 100 MB image and various chunk sizes. As you can see, this guest implementation is much slower for smaller chunk sizes, but gets close to our implementation for larger chunks. The reason for this are the frequent writes to the virtual DCR needed to send the chunks to the new VM. Each such write requires a switch into the hypervisor, which is an expensive operation. Larger chunks result in a lower frequency of DCR writes, which consequently makes this implementation much faster. However, our implementation is still about 19% faster than the guest implementation.

Interestingly, the DCR writes dominate so strongly that the loading times of this implementation become independent of the number of clients for small chunk sizes. For 4 kb chunks, the loading time for a 100 MB image was 59.5 seconds on average regardless of the number of clients.

Chapter 6

Conclusion

In this thesis, we presented a distributed storage system for supercomputer-hosted cloud platforms, capable of serving large numbers of concurrent boot requests efficiently. The system is designed to keep load off the servers as much as possible, and it does not use any central components that could become bottlenecks. Load is distributed evenly across all servers, allowing the system to handle increasing load by increasing the number of servers. It also leverages the special properties of the tight interconnection networks found in supercomputers to optimize the data transfer. Finally, it bypasses the virtualization layer by offloading the data transfer into the hypervisor, thus reducing overhead.

We have shown that our system distributes load evenly across all servers. It can handle large numbers of concurrent requests, with loading times never growing more than linear with the number of requests. In conclusion, even though there are some open issues, our system has to potential to greatly speed up booting of new cloud nodes and thus raise the level of flexibility in cloud platforms.

6.1 Future work

We have some ideas about how to further optimize the remote DMA mechanism. First, we could set the target location in the remote DMA request directly to the chunk's final location. So far, the chunks are sent into a buffer in the hypervisor's memory, and then copied into the new VM's memory. Using the final location of the chunk would make this copy operation unnecessary. However, we do not expect a large performance gain from this. If the number of clients is larger than the number of servers (which we expect to be the common case), each node has to wait after each transfer because all servers will be busy serving requests from other clients . The copy operation takes place during this waiting period and thus does not add any extra latency.

However, using the final location directly is a prerequisite for our second idea. So far, the client places one request for a chunk, and then polls the corresponding receive counter until the transfer is complete. There is however another possibility: The receive counter can be pre-set to a target value. The hardware will then send an interrupt once the counter (which counts backwards) reaches zero. Using this method would allow the client to place multiple requests at the same time in the injection fifo, either using one receive counter per chunk, or grouping multiple chunks together and using one receive counter per group. This would have two advantages: 1) parallel transfers would be possible if the client has more than one free torus link, which could greatly speed up transfers for smaller numbers of clients, and 2) chunks could be transferred out of order, which would further increase server utilization. Unfortunately, implementing this would require major changes to the memcached client library's code, which was not possible in the time allotted for this project.

We also plan to investigate the impact of the storage nodes' location inside the partition on the efficiency of the data transfer. Placing the storage nodes closer to the center of the torus and further apart from each other would allow them to transfer multiple chunks in parallel, which could improve torus efficiency. Proper storage node placement in conjunction with the interrupt-driven data transfer mentioned above could prove effective in resolving link congestion in the torus.

There are some other, functional issues that have not been mentioned before. First, our remote DMA implementation only supports reading at the moment. For storing images in the storage nodes, we have to revert to the original, TCP based memcached protocol. While this may be acceptable as long as the number of reads exceeds the number of writes (as it is the case when only mirroring stock file system images), it could become an impeding factor as writes become more frequent. This would be the case if we were to implement, for instance, snapshotting and cloning of active VMs.

Another desirable feature would be support for adding and removing storage nodes at runtime. This would allow a cloud administrator to quickly accommodate both increasing bandwidth- and storage space requirements. Unfortunately, the hash algorithm used for mapping chunks to storage nodes would cause a shuffle of almost all chunks if a node is added or removed, which means almost the entire contents of all storage nodes would have to be moved to a different node. Consistent hashing [15] could solve this problem, but it would still require some data to be moved to a newly added or from a removed storage node in order to keep the distribution of the chunks uniform.

Bibliography

[1] 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.

[2] 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.

[3] 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, New York, NY, USA, 2010. ACM.

[4] 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 *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[5] BusyBox. The swiss army knife of embedded linux. `http://busybox.net/`.

[6] Lei Chai, Xiangyong Ouyang, Ranjit Noronha, and Dhabaleswar K. Panda. pnfs/pvfs2 over infiniband: early experiences. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, PDSW '07, pages 5–11, New York, NY, USA, 2007. ACM.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gru-

ber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[8] Srini Chari. A total cost of ownership study (tco) comparing the ibm blue gene/p with other cluster systems for high performance computing, November 2008. Available online at `http://www-03.ibm.com/systems/resources/tcopaper_finalfinal_2008.pdf`.

[9] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1:1277–1288, August 2008.

[10] 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 *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

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

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[13] IBM. Device control register bus 3.5 architecture specifications, January 2006. Available online at `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/2F9323ECBC8CFEE0872570F4005C5739/$file/DcrBus.pdf`.

[14] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 7:1–7:12, New York, NY, USA, 2009. ACM.

[15] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[16] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6:13:1–13:26, September 2010.

[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] Amazon Web Services LLC. Amazon simple storage service (amazon s3), 2011. [Online; accessed 22-May-2011].

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

[20] G.F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, pages 617–632, 2001.

[21] D. Noveck S. Shepler, M. Eisler. RFC5661 - Network File System (NFS) Version 4 Minor Version 1 Protocol. URL: `http://tools.ietf.org/pdf/rfc5661.pdf`, jan 2010.

[22] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, 2002.

[23] 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.

[24] M. Stonebraker, P.M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: a new architecture for distributed data. In *Data Engineering, 1994. Proceedings.10th International Conference*, pages 54 –65, feb 1994.

[25] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal*, 5:048–063, January 1996.

[26] uClibc. C library optimized for embedded systems. `http://uclibc.org/`.

[27] 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. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 148–162, New York, NY, USA, 2005. ACM.