1-20-2006

Cheetah: An Economical Distributed RAM Drive

Daniel Tingstrom
University of New Orleans

CHEETAH: AN ECONOMICAL DISTRIBUTED RAM DRIVE

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Daniel James Tingstrom

B.S, University of New Orleans, 2004

December 2005

Dedication

To my grandfather Hugh.

I miss you.

Acknowledgements

I want to express gratitude to my advisor Dr. Vassil Roussev for his unlimited support and guidance. Whenever I was out of ideas, his ideas always came through. During the aftermath of Hurricane Katrina, Dr. Roussev kindly offered his assistance with my thesis even though his home was one of the many unfortunate flooded ones.

Thanks to Dr. Golden Richard III for his support and knowledge. I have never met anyone so smart and so friendly. His interesting ways remind me that creativity and originality still exist. He proved to be not only an entertaining teacher but also an excellent one.

In addition, I would like to thank Dr. Fu for being on my thesis defense committee and for always being there to count on for help. His patience and kindness is honorable. Also, his talent for solving algorithms is unmatched.

I want to thank my sister Jessica for the encouragement and patience she has given me all my life. With so many astonishing achievements, her future is limitless.

I would like to especially thank Nazrin, because without her I wouldn't have started graduate school and accomplished so much. She means the world to me and I owe her everything. "And the wonder of it all, is that you just don't realize how much I love you."

The most special acknowledgment goes to my parents, Randy and Denise, who have pointed me in the best direction for every fork in my life. I am very grateful for the excellent life they have given to me.

Table of Contents

List of Figures

List of Tables

Abstract

Current hard drive technology shows a widening gap between the ability to store vast amounts of data and the ability to process. To overcome the problems of this secular trend, we explore the use of available distributed RAM resources to effectively replace a mechanical hard drive.

The essential approach is a distributed Linux block device that spreads its blocks throughout spare RAM on a cluster and transfers blocks using network capacity. The presented solution is LAN-scalable, easy to deploy, and faster than a commodity hard drive. The specific driving problem is I/O intensive applications, particularly digital forensics.

The prototype implementation is a Linux 2.4 kernel module, and connects to Unix based clients. It features an adaptive pre-fetching scheme that seizes future data blocks for each read request. We present experimental results based on generic benchmarks as well as digital forensic applications that demonstrate significant performance gains over commodity hard drives.

Chapter 1: Introduction

One of the most important and well-known problems in digital forensics is how to handle large amounts of data quickly. One example is file carving, which means extracting files of specific types from a captured disk image. File carving applications cannot avoid sequentially processing the entire disk image to provide the correct results. While there is some CPU processing involved in such a task, there is a greater amount of I/O processing. Every application in digital forensics is highly I/O bound because files must be read from the disk for examination. Since most digital forensic tools belong in the application level inside of the system architecture [Figure 1.1], they cannot dramatically change the I/O performance. In order to achieve better I/O performance, a change in a lower level, such as the block device level, is needed.

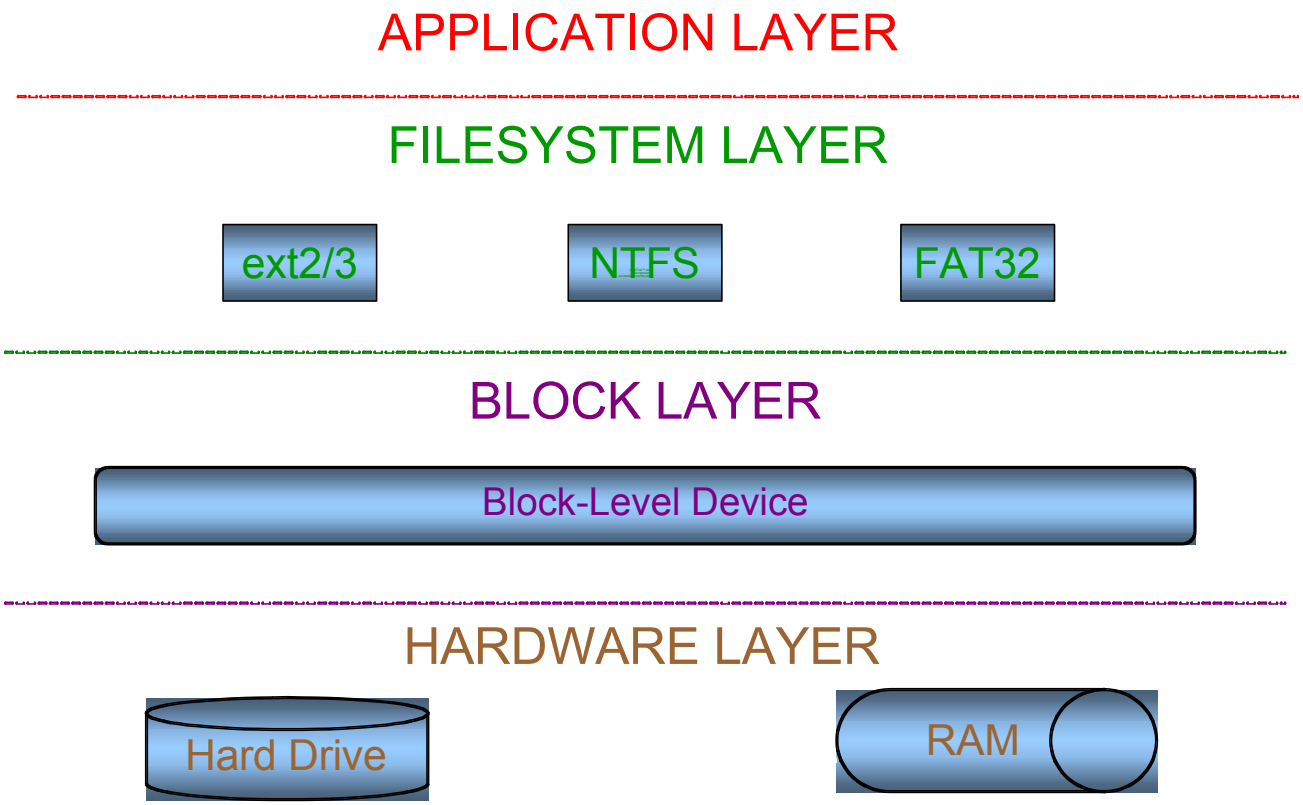

Figure 1.1 The typical architectural layering of an operating system

1.1 Motivation

Caching is a popular way to improve performance by conveniently storing data so that future accesses will be quicker. Internet browsers store web sites on the hard drive so the user won't have to download them until it's updated. Operating systems cache process information and pieces of code in RAM so they can perform frequent instructions faster and give a better experience to the user. CPUs work similarly by caching the most frequently accessed data, with speeds much greater than the Hard Drive and RAM. Unfortunately, CPU cache has a much smaller capacity that RAM, and RAM has a much smaller capacity than hard drives. Figure 1.2

displays the tradeoff of performance cost vs. storage capacity in the 3 main caching devices on a computer system: CPU Cache, RAM, and the Hard Drive.

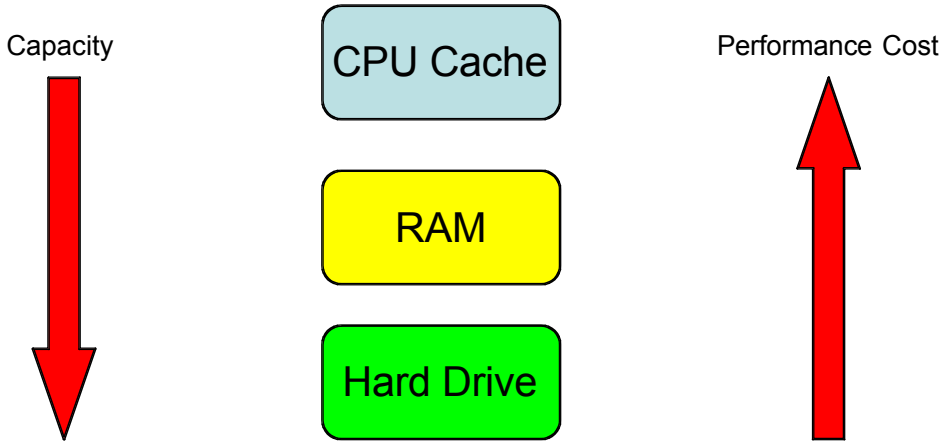

Figure 1.2 A diagram showing the performance cost vs. capacity tradeoff between the 3 main system storage components.

The performance vs. capacity tradeoff is still growing and increasingly becoming a major issue as Patterson [2] points out. Table 1.1 [2] shows the hard drive improvements through the years of 1983-2003. We can clearly see that in 1983, the bandwidth was ($1/50^{th}$ of capacity) per second, and in 2003 the bandwidth equaled ($1/854^{th}$ of capacity) per second. It is clear to see that the capacity/performance gap is increasing with time.

Hard Disk	3600 RPM	5400 RPM	7200 RPM	10000 RPM	15000 RPM
Product	CDC Wrenl 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453
Year	1983	1990	1994	1998	2003
Capacity	0.03 GB	1.4 GB	4.3 GB	9.1 GB	73.4 GB
Interface	ST-412	SCSI	SCSI	SCSI	SCSI
Bandwidth	.6 MB/s	4 MB/s	9 MB/s	24 MB/s	86 MB/s
Latency	48.3 msec	17.1 msec	12.7 msec	8.8 msec	5.7 msec

Table 1.1 A table showing the history of hard disk trends in capacity, bandwidth, and latency

Because of the performance vs. capacity tradeoff, choosing the right system device to implement was an important factor in building this system. Since our main goal was to perform faster than most hard drives, our options were down to RAM and CPU cache for storage. CPU cache is much too small in storage, even if distributed, to be useful for most digital forensic applications. RAM, which performs faster than hard drives and has larger storage than CPU cache is the option we chose to take.

One approach to increasing RAM capacity without the soaring cost of upgrading a single machine is to distribute the RAM on an available cluster. There have been many approaches at pooling the RAM resources, depending on the researchers' goals and network characteristics. Fortunately, commodity network speeds have been able to rapidly rise as shown in Table 1.2. Comparing Table 1.1 to Table 1.2, it is clear to see that with gigabit ethernet network or greater, the bandwidth exceeds that of a modern hard drive.

Local Area Network	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet
IEEE Standard	802.3	802.3u	802.3ab	802.3ae
Year	1978	1995	1999	2003
Bandwidth	10 Mb/s	100 Mb/s	1000 Mb/s	10000 Mb/s
Latency	3000 msec	500 msec	340 msec	190 msec

Table 1.2 A table showing the history of ethernet trends in bandwidth and latency

By choosing our target application area to be digital forensics, we were left with the challenge of processing hard drives that grow in size much faster than the machine can handle them. In digital forensics, this performance gap is a very common and serious problem. Frequently, as a digital forensic tool starts processing, the system will quickly run out of memory, which causes thrashing, leading to multi-day processing. One possible approach to fix this problem is to use parallel applications as demonstrated in [1], but it is the vendor's judgment to apply such an approach. Also, most digital forensic processing is inherently I/O bound, and all data on the drive must be read at least once, and often many more times to successfully gather relevant answers. Therefore, this provides evidence that having more RAM for caching will display notable performance gains, and having more CPU cycles may not yield an evident difference in performance.

After searching for a readily available solution that could be easily deployed for our purposes, we couldn't find one and thus we decided to make our own. By combining the clustered RAM approach with using the block device layer shown in Figure 1.1, we decide to build a distributed block level device that sends and retrieves its storage from RAM on a cluster.

We present this distributed block device as a practical solution that could be easily deployed to utilize RAM resources. In our lab, and most likely others, there have been numerous times when RAM is not being fully profited and part or most of the RAM just sits idle. Generally, the existence of idle RAM is a well-documented fact [4]. Our device will exploit this fact and gain storage benefit from the unused RAM resources, while using the performance of RAM. However, the latency of the network is still an issue, but with bandwidth speeds such as gigabit, an efficient gain in performance over most hard drives can still be attained.

1.2 Requirements

We decided to create some other requirements besides performing faster than the hard disk. Since Cheetah was targeted for digital forensics, we wanted investigators to be able to use this system on the suspect's cluster and also the investigator's cluster. Since we wanted to make it possible to run on a suspect's network, we had to make sure that none of the persistent data, which might contain evidence, changed. With this kept in mind, we decided to add some additional requirements as well.

1. *LAN Scalability*

   Commodity RAM on a single machine cannot store nearly as much as the hard drive on the machine. However, since digital forensics is our target, having enough space to fit disk images and other large files is required. So distributing available RAM resources on a cluster could allow for a large storage container necessary for digital forensic to copy their files to.

2. Commodity Solution

We also wanted Cheetah to be a commodity solution so that investigators will be able to benefit from this project at the location of the crime scene as well as the investigator's lab. Our system is a readily available solution, and expensive or rare components are not required.

3. Lightweight

Another important goal was to design Cheetah lightweight so it is not difficult to manage for an average computer user. Instructions to run Cheetah should be short and simple, so more time will be spent on processing from the digital forensic applications being used.

4. Digital Forensics Support

Lastly, we designed Cheetah for digital forensics, so we wanted support of multiple file systems, the ability to run from a live CD, and the opportunity to run arbitrary forensic tools. Developing a custom forensic file system in the file system layer could gain I/O performance for specific problems, but this would eliminate the ability to use other popular file systems such as FAT32, NTFS, and ext2/3. We wanted the option to lay a copy of a complete file system over Cheetah's block device, so no requirements on the type of file system should be set.

Since there are many different Linux live CDs available on the internet, Cheetah should be able to run on Linux, giving the option to inject a custom live CD with Cheetah's

software so the investigator can efficiently carry it on the same disc as the other digital forensic tools. Cheetah should also be able to run arbitrary forensic tools and should not be limited to a certain subset of digital forensic applications.

1.3 Thesis Statement

Our project, Cheetah, takes advantage of available RAM on a cluster and the bandwidth of a gigabit network to outperform hard drives while not losing the storage capacity penalty since it is LAN scalable. It tightens the capacity/performance gap, allowing intensive I/O digital forensic applications to perform better. Cheetah contains a distributed block device that performs its operations on the other servers' RAM in a cluster. The sharing of the RAM is transparent to the application being performed, allowing for a wide range of digital forensic tools.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 reviews the current solutions to increase drive speed performance and drive storage scalability. Chapter 3 presents the design of our specific approach. Chapter 4 explains in detail the implementation of this project. Chapter 5 displays the test results to prove that performance gain is achieved. Chapter 6 includes our conclusions and the ideas being developed for future work.

Chapter 2: Related Work

Distributed RAM sharing is a well-established idea, and a number of implementations have been developed over the years. Generally they fall into two broad categories depending on their interaction with the user process. The first approach is to hide the fact that the sharing takes place and by tricking the application into believing that there is a greater amount of RAM available than there actually is. This behavior is similar to the way virtual memory works. The difference is that, instead of coming from the hard drive, the extra memory is physical RAM on another machine on the network. The second approach is to expose the sharing and give the application some means to control the sharing process.

This section will discuss a few related systems such as Distributed File Systems (DFS), the iSCSI drive, Google FS, Network Block Device (NBD), and Remote Direct Memory Access (RDMA) over InfiniBand. Before summarizing these systems, we should mention that a number of simulation studies have been performed to explore the viability of different ways of distributed RAM sharing. For example, Dahlin et al [3] used a trace-driven simulation to study the performance benefits of cooperative file caching using several cooperative caching algorithms. “Cooperative caching seeks to improve network file system performance by coordinating the contents of client caches and allowing requests not satisfied by the cache of another client.” [3] This caching technique is designed to improve cache performance for system reads only, and does not address issues such as write performance and large file performance which happen to be extremely important in the digital forensics field.

In [5], and later[6], Xiao et al. studied the impact of combining network memory and job migration for system scalability and throughput improvement. A Parallel Network RAM solution, based on global management was proposed for scientific applications.

2.1 Distributed File Systems

A Distributed File System (DFS) is a file system that supports sharing of files and resources in the form of persistent storage over a network. Distributed file systems can scale very large, and immense disk sizes may be needed depending on what problem is trying to be solved. This large size capability is an advantage while the weakened performance is a disadvantage. In conventional systems, performance consists of a disk-access time and a small amount of CPU-processing time.

There is a transparency involved with distributed file systems, since the client interface should not make a difference for the user to read or write to local and remote files. This gives not only user friendliness but also allows applications to transparently read and write from the distributed file system even though the files processed might be on a remote server.

In a DFS, our requirement of system layer transparency is fulfilled, but the file system performance is weak because not only does the normal overhead occur, but also an additional overhead from the network's transmission delay. Also, this solution eliminates the possibility of using other file systems for digital forensics work.

2.2 iSCSI

iSCSI enables a machine on an IP network to contact a remote dedicated server and perform block input and output operations just as it would do with a local hard disk . iSCSI operates on top of TCP and uses longer packet headers that include additional information to speed up packet assembling. Scalability is available, but performance is downgraded because of the high latency. The main reason for the latency is because of the iSCSI protocol being layered

on top of TCP, and then the normal SCSI interface is on top of iSCSI. Figure 2.1 illustrated the layers involved to unwrap, causing the latency.

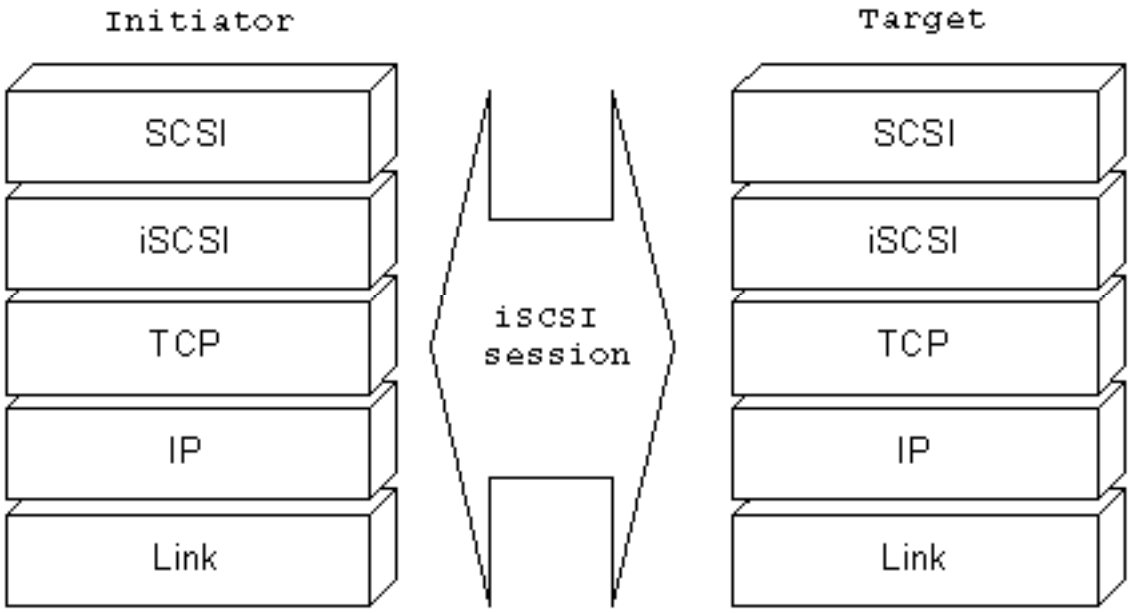

Figure 2.1 This diagram illustrated how many layers that must be unwrapped for each packet received in the iSCSI session, which causes latency issues

2.3 Google FS

According to [7], Google File System (FS) is a scalable distributed file system for large distributed data-intensive applications. It is widely deployed within Google as the storage platform for the generation and processing of data, and it is also used for research and development efforts that require large data sets. As shown below in Figure 2.2, Google FS is composed of a master and chunk servers. The master contains all the file system metadata, including the mappings of the files to chunks, which are stored in the chunk servers.

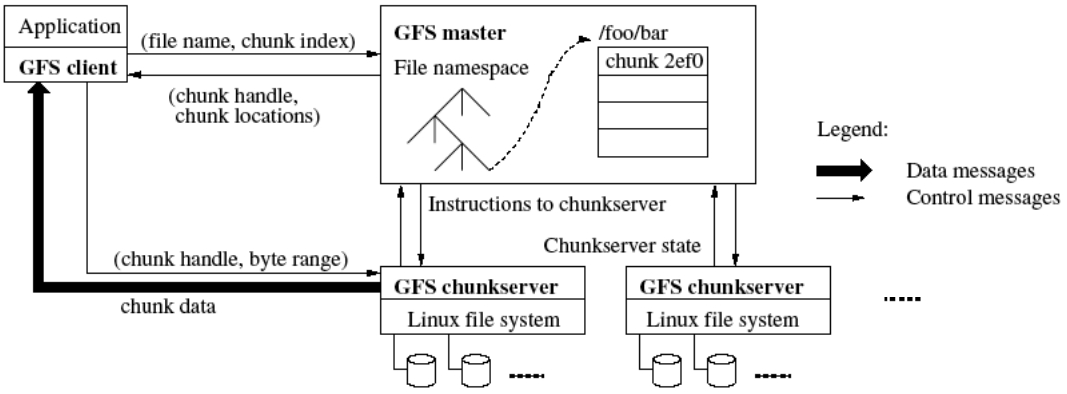

Figure 2.2 The architecture of Google File System

Google FS also provides fault tolerance to a large number of clients by replicating chunks into separate chunk servers. While this technique might benefit the client when one of the chunk servers unexpectedly goes down, for large data sets it does require more chunk servers and could run into a non-commodity to be successfully fault tolerant. Also, Google FS is optimized for sequential reads/writes of files, but unfortunately in digital forensics non-sequential access is very common.

2.4 NBD (Network Block Device)

The network block device application, or commonly referred to as NBD, allows the Linux user to access block data from a remote server. Since it acts as a block device, the user is allowed to lay any file system on top of it. This allows for scalability of the device, similar to the distributed file systems, but now any file system can be laid on top after the nodes are ready.

NBD does have its set of limitations. It is impossible to use it as a root file system, and it only allows the user to run as a read-only block device in user-land. It will also deadlock "within seconds" if the server and client are both on the same machine. Another drawback is the

performance is still hurt by the TCP overhead adding to the overhead of the disk and CPU. NBD's throughput is equal the hard drive's throughput.

2.5 RDMA Over InfiniBand

RDMA is a communications technique that allows data to be transmitted from the memory of one computer to another computer without:

- Passing through either computer's CPU
- Needing extensive buffering
- Calling to an operating system kernel

RDMA helps gain network performance by not having to pass data through the CPUs. InfiniBand is an example of a form of RDMA that sends data in serial form and can carry multiple channels of data at the same time in a multiplexing signal. The channels are created by attaching host channel adapters (HCAs) and target channel adapters (TCAs) through InfiniBand switches. The HCAs are I/O engines located in a server. The TCAs enable remote storage and network connectivity into the InfiniBand interconnect infrastructure, called a fabric. InfiniBand architecture is capable of supporting tens of thousands of nodes in a single subnet and transmission rates begin at 2.5 MB/s. Figure 2.3 shows the layout of the InifiniBand architecture[1].

[1] Figure 2.3 is taken from http://www.oreillynet.com/pub/a/network/2002/02/04/windows.html

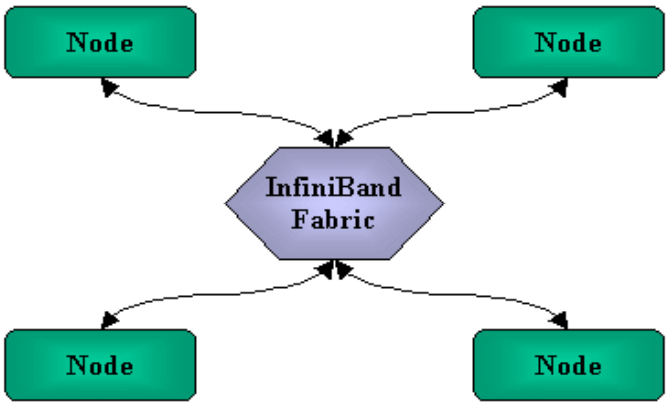

Figure 2.3 Topology of InfiniBand architecture

In [8], an RDMA-optimized implementation of the MPI library is used to provide the transparent use of remote memory. The problem with RDMA is that it's not a commodity, and it has high network latency issues.

2.6 Summary

Each of the described practical solutions was missing at least one of our project's requirements such as higher performance over a hard drive as well as a commodity solution. We found some similar projects only done as simulations such as [9] and [10], but there were no practical implementations to be found. Therefore we had to develop our own design to meet all of the criteria specified in our requirements.

Chapter 3: System Design

In this chapter, we describe the design of the two main system components, Cheetah's block device and the cache servers. In order to properly describe how these parts work, let us also describe the communication between them.

Consistent with our goals for a simple system that can easily be deployed, we have opted for a design that is minimally invasive to the system software and is fairly portable. The basic architecture of Cheetah, shown in Figure 3.1, consists of a set of user-level RAM server processes that provide local RAM access to a central RAM client. The different types of shaded boxes represent certain sections of blocks, and the diagram shows the mappings of their locations in the client host to their locations in the RAM server hosts.

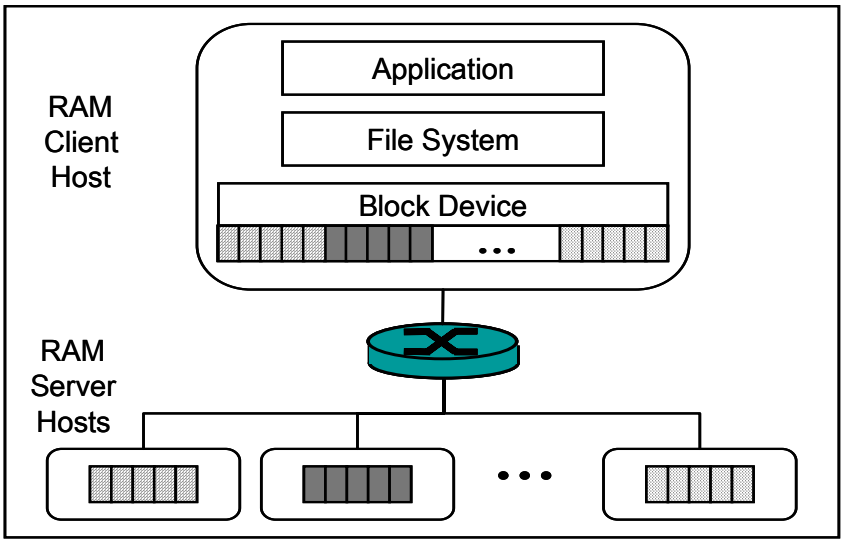

Figure 3.1 The architecture of Cheetah

The central RAM client, consisting of the block device, sends and receives all of the block data from different server hosts, which hold the blocks in each of their local RAM. Since

RAM is volatile, any data transferred through the network or stored in RAM will disappear once powered off, leaving the hard drives and other non-volatile storage on all of the server hosts in tact.

3.1 Block Device Module

There were 2 choices in designing this component of Cheetah:

1) The first choice is to implement file system level caching, which would allow optimizations based on the logical structure of the file system, but would also break one of our requirements by making it file system dependent.

2) The other option is to implement a block level device that does not have the benefit of knowing about files and directories (and likely access patterns) but would work with any file system.

Our decision of using the block level device was based on two factors. The first one is that for our specific application domain, digital forensics, unlike in most other domains, the applications do care about unallocated space and preserving the original block-level layout of the file system is necessary. The other factor is that operating systems already do a very good job of laying out files sequentially so simple read-ahead optimizations may well be enough to achieve good performance.

The block device is a Linux module that runs on any Linux 2.4 kernel. The kernel can be compiled on a bootable live CD, allowing the module to be also installed on the bootable CD, for a portable usage. The device is the first thing to be initialized and manages all the blocks of storage as long as Cheetah is running.

3.2 Cache Servers

The cache servers run on any Unix type platform such as Linux, Mac OS X, Solaris, and FreeBSD. The cache servers can also run on any live bootable Linux CDs. These servers should start up after the Cheetah module is loaded and ready. There is no particular order to start each of the cache servers, but the order in which they are started is the same order of where the blocks will be stored. For instance, if server A was started first sharing 180000 1KB blocks, then A will contain blocks 1-180000.

Here is a diagram showing how the blocks are stored from each cache server:

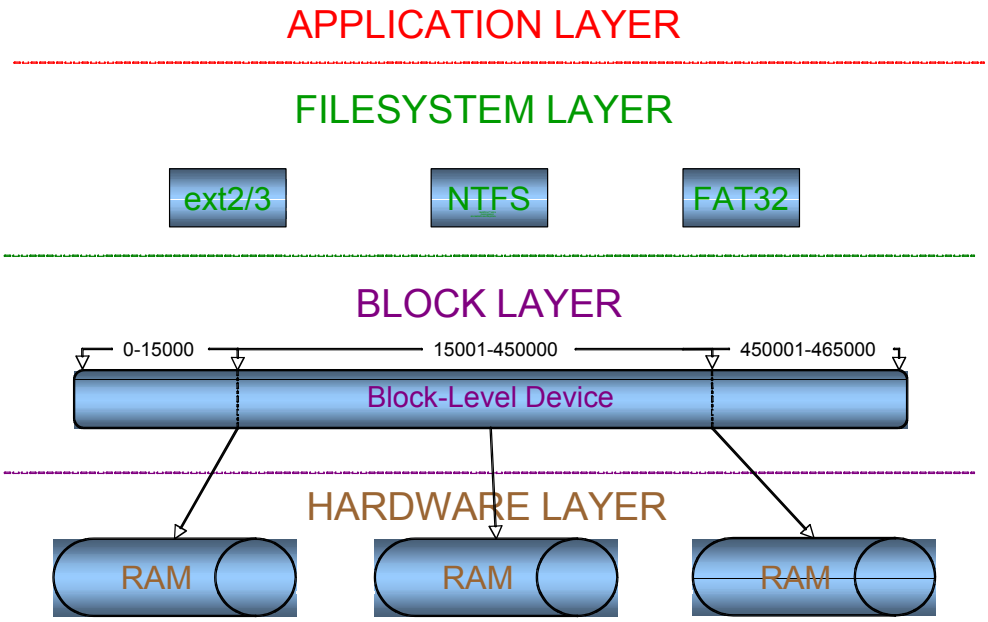

Figure 3.2 A diagram showing how the different cache servers are distributed and connected to the block level device.

To describe how Cheetah works, let us point out the 3 step process involved in the communication between system layers:

1) An application makes operating system calls to operate on the file system layer.
2) The file system determines what to do based upon the given operations. If there is reading or writing involved, it most likely calls the block level device in the block layer, which ends up to be Cheetah's block device.

3) On the block device, the mapping of where blocks are stored on the cache servers is transparent to the file system level and above levels in the system architecture. The device connects to one of the cache servers, specifically RAM in one of the cache servers, which is in the hardware layer.

To get worthwhile speed results, the block device and cache servers must be on the same LAN, preferably with gigabit speeds. If the network speed is not fast enough, the bandwidth will be noticeably downgraded. The network bandwidth does play an important role in the architecture, especially since for every block written and read, it must go through the network first. After data is sent through the network, the overhead of the disk and CPU must then be accounted for also.

One drawback to the cache servers is that allocated memory cannot be locked into RAM so there is the possibility of the memory getting swapped out with other memory via the paging system. Our rationale, supported by our experience, is that for an idle system the user will be able to fill up the memory in RAM since there will be no competition of who gets to be in RAM. The amount of RAM to share on each of the cache servers should be chosen dependent upon the amount of free RAM available on the servers. If a cache server starts another process that happens to be memory-hungry, then performance will notably suffer.

3.3 Communication

We use an application-level protocol to perform service discovery and to exchange block read/write operations over TCP socket connections. TCP/IP processing overhead is a well-known

source of inefficiency, especially for high-speed communication networks. For a commodity solution seeking the lowest common denominator, the only other realistic options are UDP and Ethernet frames. Evidently these would need separate reliable transmissions mechanisms, very similar to the one already provided by TCP. In initial testing, we did not find any appreciable difference between a TCP version and a UDP one (without a reliability mechanism). For our first version, presented here, we decided to go with the basic TCP solution and revisit the issue, if the performance is unsatisfactory.

Another argument supporting TCP/IP is that IT users have been very reluctant to adopt more efficient (but less widely accepted as standards) solutions designed to take advantage of more efficient communication technologies (e.g., InifiniBand). As a result, many vendors are providing TCP/IP emulation that enables users to take advantage of most optimization without parting with the "good old" TCP/IP sockets. Specialized Ethernet "accelerators" are emerging with TCP/IP implementations on a chip. Even SMP machines (e.g. from IBM) come with TCP/IP emulation so that the same code could be run on a cluster and on an SMP machine with shared memory. In other words, we have good reason to believe that TCP/IP is not going away anytime soon even for high-performance computing and that, in many cases, TCP-based solution would be able to directly benefit from hardware improvements.

3.4 Adaptation Scheme

Even though our project was tested on a gigabit network, TCP did tend to slow performance down to about the same as a hard drive. We used 1KB block transfers that comfortably fit into an Ethernet frame, which seemed logical. However, real numbers showed a less than 5% improvement over our hard drive. This kind of performance isn't acceptable since

one of the main goals of Cheetah's device is end-to-end latency improvement. Subsequent experiments confirmed that, for large files, simply pushing the transfer (read-ahead) unit to 100 KB yielded substantial performance improvements over the hard drive. For small files, that is clearly too expensive.

We implemented an adaptive read-ahead scheme to accommodate the conflicting requirements of large and small files. This adaptation technique works as a 3 step process:

1) The initial read-ahead transfer unit is set to the minimum, 4KB
2) If a successful block request is adjacent to the previous block transferred, the size of the transfer unit is doubled subject to a maximum parameter, 128KB
3) If the user stops reading blocks sequentially, then the number of read ahead blocks resets back to the minimum, 4 KB

Since the number of blocks to be read ahead keeps adapting to whatever the user is doing, this technique proved to increase performance dramatically. This scheme is quite similar to the TCP slow-start algorithm – every adjacent block request is treated as a "success" leading to the doubling of the transfer window, while every non-adjacent one is treated as a failure (akin to packet loss) and the window is shrunk. The minimum of 4KB was picked because it is typically used by operating systems as the minimum allocation unit. The maximum of 128KB was picked after testing identified it as the point of the diminishing returns. Further increases beyond 128KB yielded only marginal improvements in performance. The presented adaptive scheme is somewhat similar to the one used by the *Linux* kernel in version 2.6 (which has its own issues [6]). There are at least two notable differences:

1) In the block device level we simply do not know about files, so file-based optimization is not possible.
2) Our read ahead is more aggressive and works along the file system read-ahead.

Chapter 4: Implementation

In this chapter we describe the implementation of our prototype that allows Cheetah's distributed block device to perform faster than a hard drive. This prototype also allows users to keep adding cache servers to gain the desired size of the virtual drive and the ability to lay any file system that Linux can read on top of the device.

4.1 Technology Used

We implement our system around the GNU/Linux operating system for three main reasons. The first is because since the entire kernel is open source, there are a great number of free resources that made the module development process quick. The second reason for developing a Linux module is so the user can lay any file system (FAT, NTFS, ext2/3, ReiserFS…) that the Linux Virtual File System can read on top of the block device. Other operating systems such as Microsoft Windows XP do not allow such a wide variety of file systems to be mounted. The third reason is to give forensics support by putting the module on one of many different Linux bootable CDs. The CDs allow forensic investigators to stealthily use Cheetah on any network of workstations without modifying any of the non-volatile devices on the network.

4.2 The Module

Cheetah's block device is a 2.4.x Linux kernel module written in C using the kernel headers. We added a script for user-friendliness that will re-compile the module to a specified Linux kernel version. For example, if launch the module on a 2.4.31 kernel, but the module is

currently compiled for 2.4.18, then the user can just type the command "./k linux-2.4.31" which will re-compile the kernel. The device can be loaded and unloaded with root permission. The user has an option to load the module manually or with another script "./s". This script will not only load the module but also start the block device at /dev/cheetah. At this point all of the cache servers can be started and the /dev/cheetah block device will incrementally add more block space. There is a thread running in the module that continually looks for cache servers until the device is mounted. In order to achieve this dynamic effect for users to add cache servers at any time after the device starts, Cheetah re-initializes the device while saving some of the data variables that hold information about the past blocks.

Support for developing a normal Linux kernel block device can be found in many places including books and on the web. Even though Cheetah' distributed block device is a "special" block device, these resources shortened the time needed to make the "normal" block device work and gave us more time that was needed to bring the "distributed RAM" block device to life.

4.3 The Cache Servers

The cache servers are also written in C, and they use the normal UNIX networking library for communication calls. Since the source code for the cache server is short and simple, it wouldn't be difficult to port to a Windows machine. The cache servers can be started with two parameters:

1) The IP address of the machine loaded with Cheetah's distributed block device
2) The number of 1KB blocks to share from that local machine

The number of blocks should be carefully picked because it can hog the machine's RAM, periodically freezing the operating system. Also, if another process starts to need more memory, then thrashing could occur. Enough memory should be left free to make the machine at least somewhat usable. We tested machines with 2GB of total RAM and shared 1800000 blocks, or 1.8GB.

4.4 User Interaction

This section describes how to start the block device and cache servers. The block device must be started first with the following 2 commands: insmod and mknod:

```
$ insmod cheetah.o
```

The `insmod` command inserts the compiled block device module object file (named cheetah.o) in the Linux kernel. This will also create a new block device that can be reference in Linux with a major number 254. This number is used in the next step.

```
$ mknod /dev/cheetah b 254 0
```

`mknod`'s first parameter is the path of the new device that you want to create. The second parameter, "b" is for a block device type. 254 is the major number that was created in step 1 and 0 is the minor number, a number that references different devices if more than 1 are created.

Now since the block device is started, all of the cache servers can be initialized with a specific amount of memory blocks to share by using this command:

```
$ ./c 10.0.0.1 1800000
```

c is the binary executable that shares a specified amount of 1KB blocks of RAM (in this case 1800000, or 1.8 GB). The first parameter is the IP address of the machine containing the distributed block device.

As each cache server starts to share empty blocks in their local RAM, the block device will continuously sum up the total blocks from the previous cache servers. After the last cache server is started, the block device will be loaded with as many blocks as the total amount of blocks that the cache servers are sharing. It is important to note that all of the blocks shared by the block device are empty, which allows any file system to be layered on top, or even a single file as well. In order to lay a file system on top it must be mounted properly by following Linux Virtual File System (VFS) specifications.

4.5 Communication Protocol

The communication protocol that Cheetah uses is TCP for reliability. UDP was tested but showed many problems. The following diagram shows the steps of typical transferring in Cheetah:

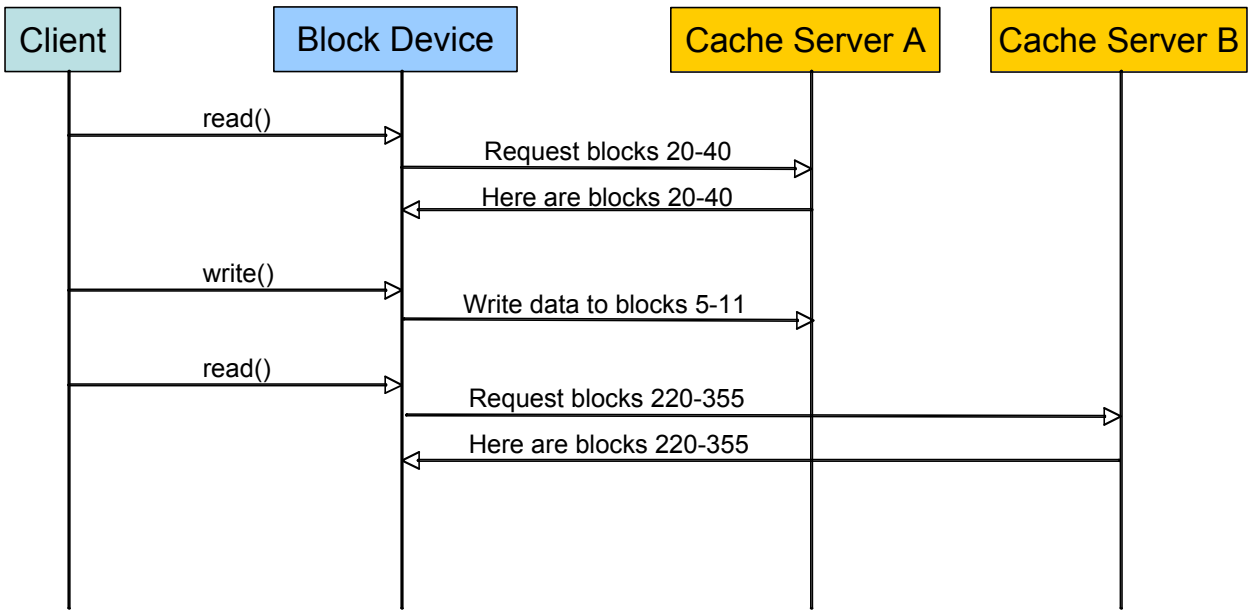

Figure 4.1 A flow chart showing some examples of communication between the cache servers and the block device.

Figure 4.1 shows how the block device will execute commands to more than one cache server, in a synchronous order caused from TCP stream communications. When a client executes a read() command for a file, the block device first figures out which cache server and which local blocks to read from that cache server. Then it requests from the cache server the right amount of local blocks. When a write() command is executed, the block device figures out which cache server and which blocks on that cache server to write to. It then sends the data and the location of which blocks to write the data to.

In UDP, the ordering of packets arriving, or the knowledge if packets ever arrived, is not implemented. So for example we used UDP, and modifying the order in Figure 4.1, let's say we to do a write to blocks 35-50 in between the first request and response of blocks to read. This means that we requested blocks 20-40 to read, wrote new data into 35-50, and then read 20-40, with 5 changed blocks, giving back the incorrect result. The communication of commands

getting executing in Cheetah is just as important and reasonably has the same focus as read/write concurrency in operating system memory management.

The blocks were being requested too fast and some requested blocks were either not coming back to the module or out of order. Since this project is set to be a digital forensics solution, the forensics rule of preserving data is vital for this project. So a reliability implementation was needed, and the basic TCP fits our problem precisely.

Chapter 5: Evaluation

Throughout our design and implementation process, we have targeted the development of a practical solution that can benefit users. Therefore, a principal question for our testing methodology was the selection of test cases that best represent typical access patterns. After considering our goals, we concluded that the main measure of success is the ability to speedup sequential access patterns. The rationale here is twofold:

1. It is the best side of hard drive performance – randomized patterns clearly kill HDD performance and play to our strengths.
2. Today, non-sequential access patterns are not the norm, but the aberration.

For example, Google FS [7] does not even attempt to optimize for non-sequential access. Applications that do need to access large amounts of data with potentially randomized patterns explicitly manage their I/O requests to improve performance (DBMS are an obvious example). A common exception from these cases is file servers: due to concurrent independent requests, the block requests could become really scattered. This, however, should naturally favor solution over a mechanical drive.

We wanted to evaluate our block device performance by using both a disk benchmarking tool and various digital forensic applications. The reason for using the benchmarking software is to test the true I/O performance, and get results such as how fast the reads/writes from the operating system are without having to worry about interference from other resources. Since we focus on digital forensics, it makes sense to see how forensic applications perform using

Cheetah.

5.1 Hardware Setup

All testing was done in the NSSAL lab at UNO. There were 6 Dell Workstations used, each with 2 GB of RAM and 3.0GHz processors. One of the machines ran the module while the other 5 machines acted as cache servers. The machine that ran the module also gave up 1GB of its RAM also acting as a cache server. Since the 1GB was on the same machine as the module, the performance was very fast when those blocks inside the dedicated 1GB were being accessed. It made sense to use the local RAM wisely. This machine with the module was running Red Hat 9 on a compiled 2.4.31 Linux kernel. The other 5 machines were booted with KNOPPIX live CD, which also ran a Linux kernel, but in its original KNOPPIX version. Since the purpose of the live CD is to run the kernel and file system only in RAM, then some memory had to be left free for the operating system to remain stable. Therefore out of 2 GB total RAM, 1.8 GB was taken and used for each of the 5 cache servers. The diagram in Figure 5.1 shows our hardware setup for testing.

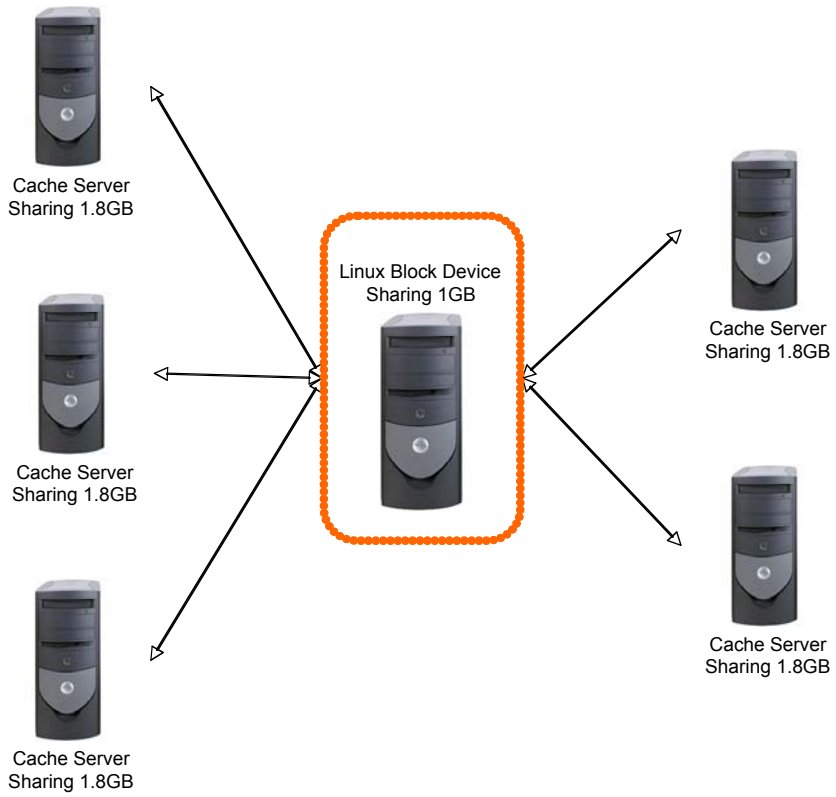

Figure 5.1 A diagram representing our block device connecting to our 5 cache servers that were used for testing

Summing the cache servers' total memory shared, we end up with a total of 10 GB (5x1.8+1) of free memory allocated for Cheetah's block device to use.

The HDD used in the block device's machine for comparison with Cheetah was a randomly picked 60 GB Hitachi IDE drive from our lab and was directly attached to the host executing the applications. For testing, we used the complete content of two randomly chose hard disks from our general purpose lab, 4.3 GB and 6.4 GB, respectively. For the network experiments, the test images were preloaded onto the distributed RAM drive. Before running the tests, we benchmarked both the HDD and the network as follows:

Network: End-to-end sustained bulk IP network transfer observed by processes: 100MB/s. This was higher than our expectations so we performed the same experiment with two other switches – bigger and much more expensive – and they established similar results.

HDD: Sustained file system level bulk transfer (mass sequential copy): 24MB/s.

These baseline results show that the commodity network has the clear potential to beat the commodity HDD for bulk transfers, which is the strong suit of the hard drive. For random access, we would expect the performance gap to widen considerably.

5.2 Software Setup

IOzone (http://www.iozone.org/) is a file system benchmark tool for Linux. It has a number of different options including 15 different tests that can be run: read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants, aio_read, aio_write, and mmap. While testing we used a record size of 4 KB and a maximum test file size of 8 GB due to a limited 10 GB of total RAM space available on our LAN setup.
We also wanted to get results using applications that people use commonly and applications that forensics investigators might use. md5sum proved to have one of the best results, and it displayed to have much more disk use in the application rather than CPU usage. tar also proved worthy since it also is disk intensive. tar was tested by compressing many small files (images) into a tar file and then extracting them.

There were two forensic applications used. The first was from Sleuthkit, a free Linux digital forensics tool set. The tool used from this set is named Sorter. Sorter looks into a disk image, and carves out various known formats such as Microsoft Office documents, text files, pictures, sound files and so forth. Sorter also has many options available, such as multiple file system support, md5sum checking, sha1 checking, html output, an option to list the files and not extract them, and many other options. Sorter is a great tool to use, but unfortunately Sorter proved to have the least best results because it is not as disk intensive as the other applications and is more CPU intensive. The results will be explained in further detail in the following section.

Disk images can be saved in raw format into a single file using the dd tool. These dd made images are very popular and used commonly in digital forensic applications. On hand we had one NTFS 4.3 GB image and another NTFS 6.4 GB image to run experiments with.

5.3 Results

Below, we summarize the benchmark results, as well as some digital forensic tools. Since our project is focused on digital forensics, and since we focus on lowering latency, the selected applications were highly I/O intensive. We examined the general intensity by monitoring the CPU, disk, and network usage by simply using the built in resource monitoring application in Red Hat 9. Sorter was the only application that was not as high disk-intensive as the others, and this tool was selected to show that results are still not downgraded with Cheetah if the application uses other I/O resources as well as the disk.

5.3.1 IOzone

Since IOzone is a disk benchmarking tool, it has much more I/O intensive operations than the other test applications selected. Because of this, the improvement of performance results is far more apparent. Below is Table 5.1 that displays them summary of numeric results and brief descriptions of the IOzone benchmark measurements.

Test	Performance (KB/s)		Relative Speedup
	HDD	RAM disk	
Write	24,026	92,309	284%
Rewrite	25,788	92,628	259%
Read	26,568	88,768	234%
Reread	26,487	88,357	234%
Random Read	396	9,065	2189%
Random Write	495	10,501	2021%
Backwards Read	5,071	18,216	259%
Strided Read	5,243	7,834	49%

Table 5.1 A table that shows Cheetah's percent increases over the hard drive

- **Write**: Sequential writing to a new file
- **Re-Write**: Sequential writing to an existing file
- **Read**: Sequential reading of an existing file
- **Re-Read**: Sequential reading of a file that has already been read
- **Random Read**: Reading from random locations with a file
- **Random Write**: Writing to random locations with a file
- **Backwards Read**: Sequential backwards reading of a file
- **Strided Read**: Reading a file with a strided access, e.g., a 4KB read followed by 200KB sequential seek, another 4KB read, and so on

Cheetah was able to achieve 88-92% of the sustainable IP bulk transfer rate over our network. Clearly, the write performance is not a function of any optimizations on our part but is an artifact of the ability of TCP to sustain the measured rate. In the other hand, the read performance demonstrates that our aggressive adaptive scheme is able to feed enough data to keep TCP busy at close to that same rate.

Figure 5.2 shows the Write/Read and Re-Write/Re-Read performance results. We can clearly see that Cheetah performs substantially better than our hard drive using these common, basic file system operations. The distributed RAM disk performed about 3.5 times faster than the IDE drive used. Recall that, from the initial benchmarking, the raw network transfer rate (our practical limit) was 4 times the HDD one.

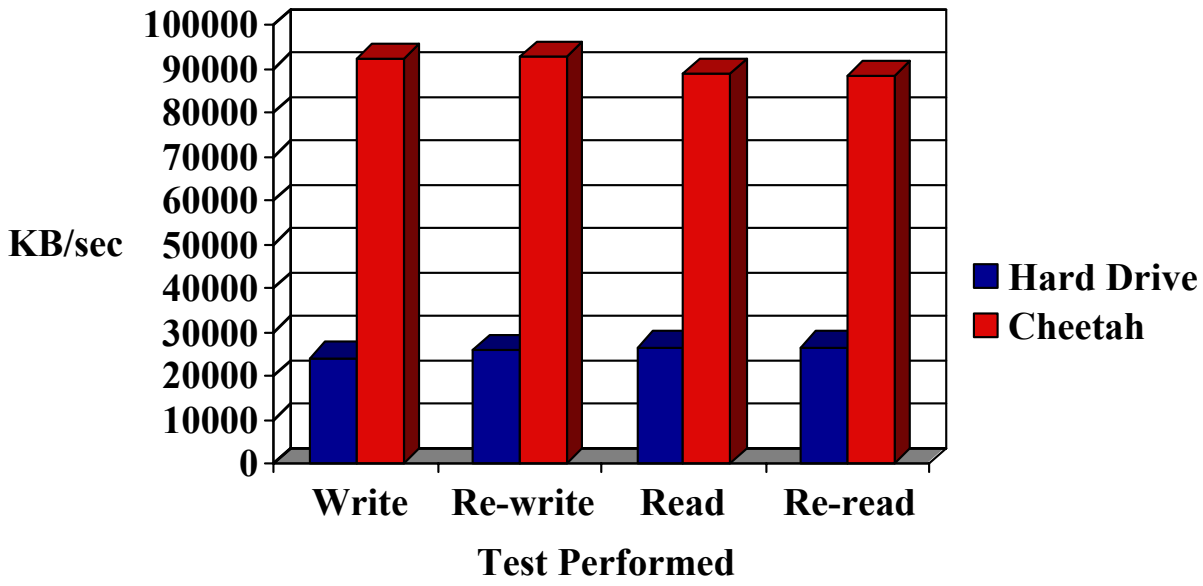

Figure 5.2 IOzone's read/write and re-read/re-write performance results

Figure 5.3 shows us the results of random writing/reading on both the hard drive and Cheetah's block device. The RAM disk showed an average improvement of 22 times over the mechanical drive. At the same time, the observed transfer rate (~10 MB/s) was 10% of the maximum, whereas for the hard drive that number is well under 2%.

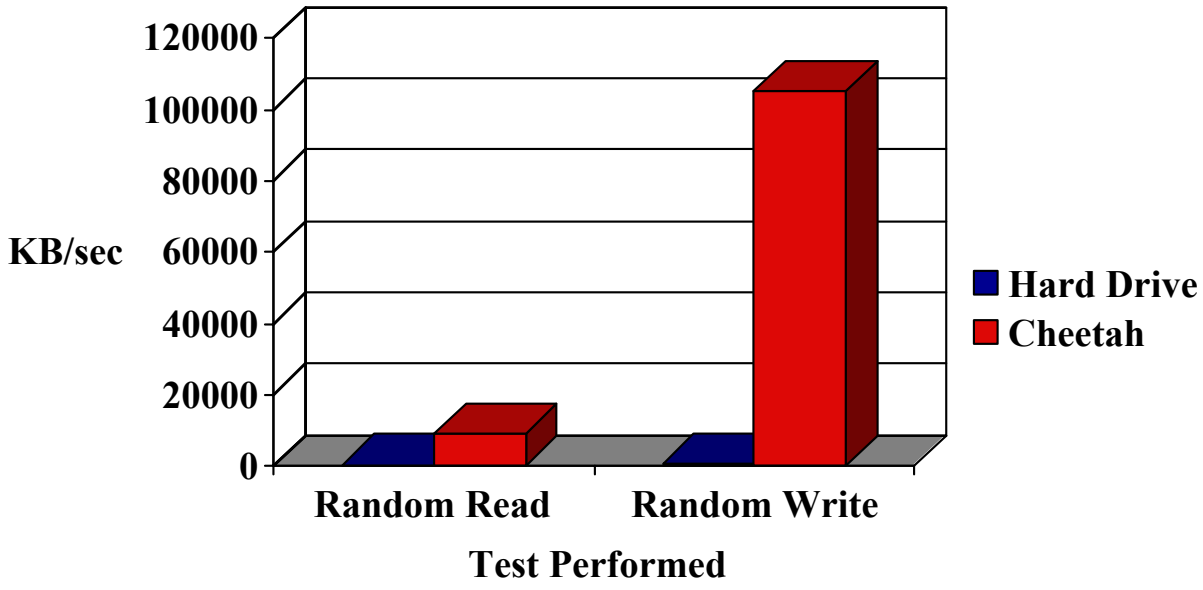

Figure 5.3 IOzone's random read/write performance results

Figure 5.4 shows the results of two types of reading, backwards and striding. The 100% improvement of backwards read over random read for Cheetah's RAM disk is entirely due to the read-ahead policy of the kernel – we did not tweak our read-ahead algorithm to handle this the way we handle forward read for the sake of the test. We find the result interesting as it gives an idea of the relative effects of file system read-ahead and block device read-ahead policies.

One relatively minor discrepancy are the stride read results for the RAM drive – they are somewhat lower than the random access results, which we would expect that to be the absolute floor of performance. Since the block device does not do anything differently, our best guess is that the file system issues read-ahead requests that are eventually not used and not counted by the benchmark application.

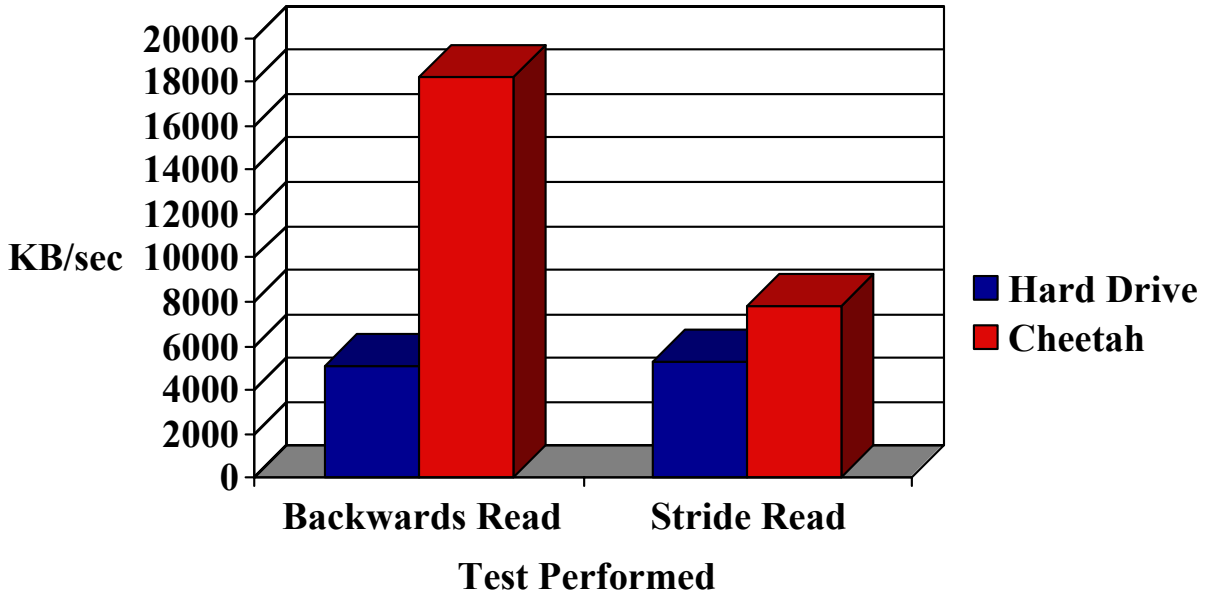

Figure 5.4 IOzone's backwards and strided read results

5.3.2 Tar

Figure 5.5 shows that the standard Unix archiving utility, tar, has good performance in Cheetah. tar was not performed on the 6.4 GB disk image target, but only the 4.3 GB target. The

reason is that our setup provided only 10 GB total, while tar needed 12 GB to successfully perform the archive operation on the 6 GB disk image.

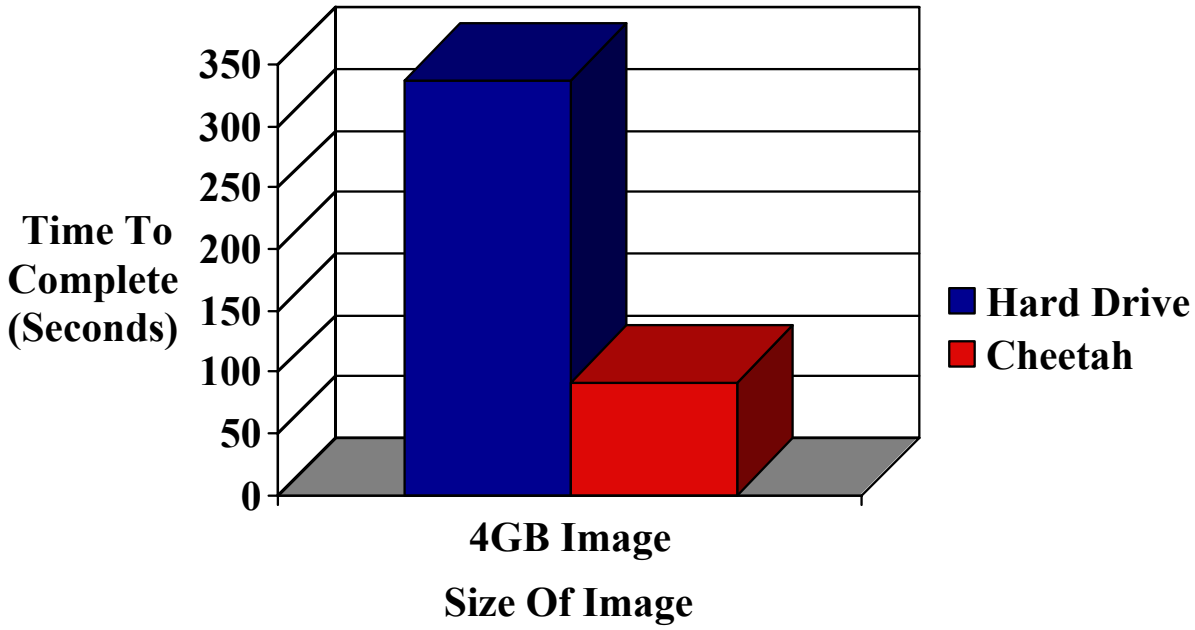

Figure 5.5 A chart comparing the time to complete an operation of creating a tar archive using the standard tar tool

5.3.3 Sorter

Sorter is a Perl script that analyzes a file system to organize the allocated and unallocated files by file type, and is found in the digital forensics Sleuth Kit package. It is clear to see in Figure 5.6 that Sorter does not have as great of improvements as the other test applications, but this is because Sorter uses utilizes more of the CPU resource than the disk resource. The reason

for presenting this result is to show that even under the condition that the application is more CPU bound than I/O bound, the application's performance will not worsen.

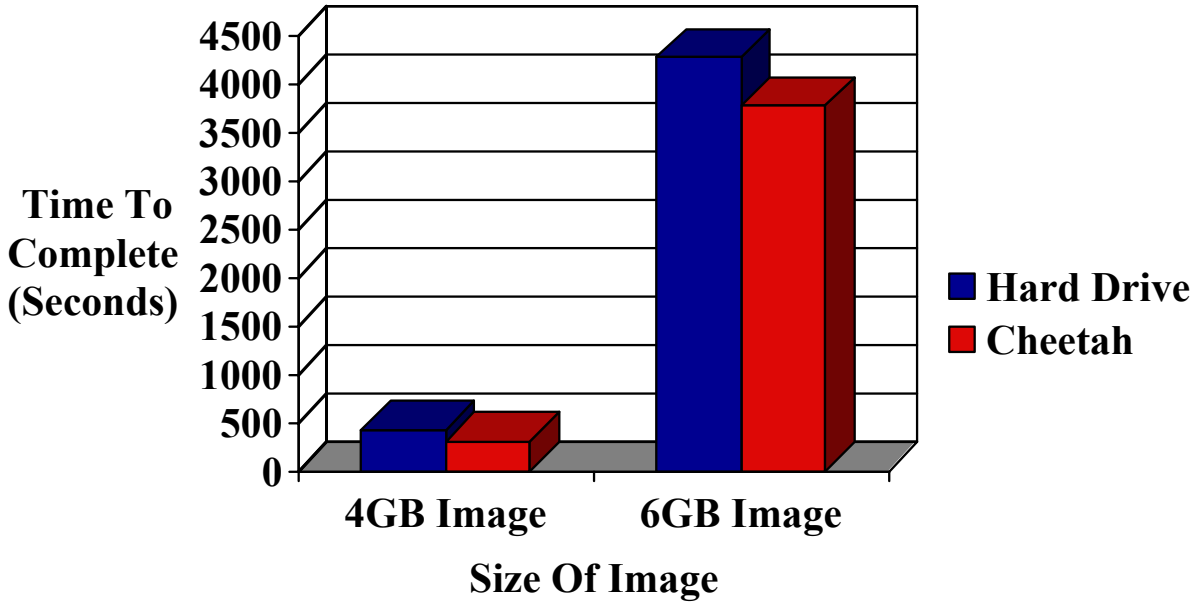

Figure 5.6 A chart comparing the results of two disk images using the Sorter tool, taken from the digital forensics Linux Sleuthkit.

5.3.4 Scalpel

Scalpel is a digital forensics file carver [11], similar to sorter. The main difference for our testing purposes is that Scalpel showed signs of higher disk intensity from the resource monitor. This allowed Cheetah to outperform the hard drive rather well. The reason why the two disk images have different performance increases (156% and 80%), is that for the 6 GB disk image, different parameters were used since the two disk images are totally different in both the amount

of files, and the types of files contained on each of the images. Figure 5.7 shows the results of Scalpel.

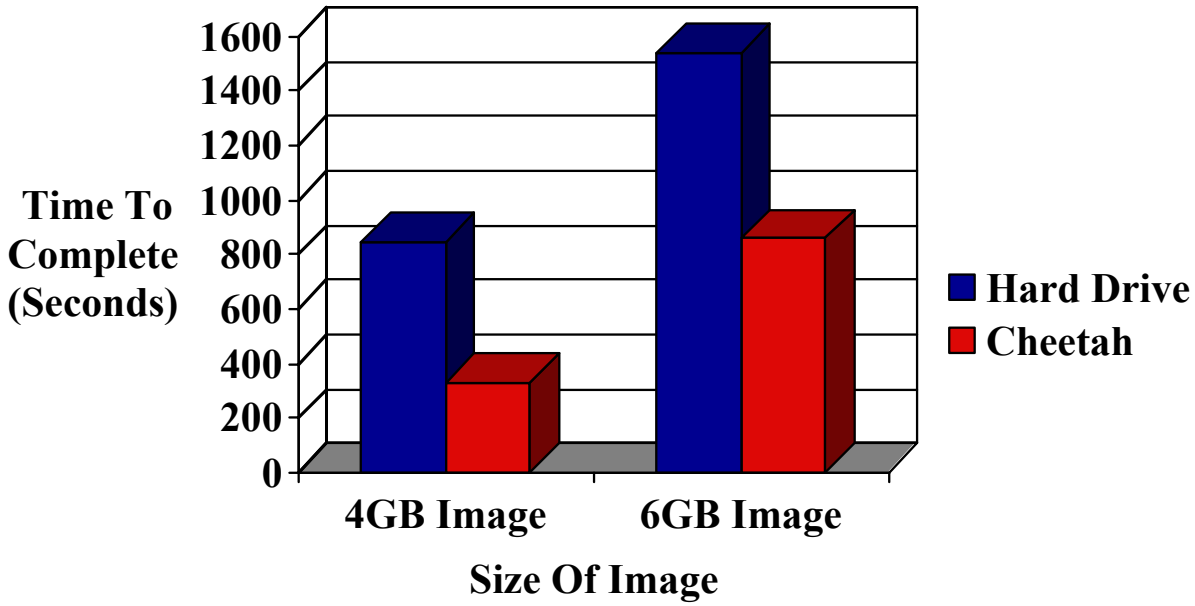

Figure 5.7 A chart showing the performance comparisons from using the Unix MD5Sum tool

5.3.5 MD5Sum

Here we test using the Unix MD5Sum application. In digital forensics, it is very common to use this tool to both

1. Identify and give file signatures easily and,
2. Accurately checks to see if a file or disk image was modified from the start of the investigation.

The results, displayed in Figure 5.8, were at first glance very interesting to us. MD5Sum revealed to be strongly utilizing the disk resource much greater than the CPU.

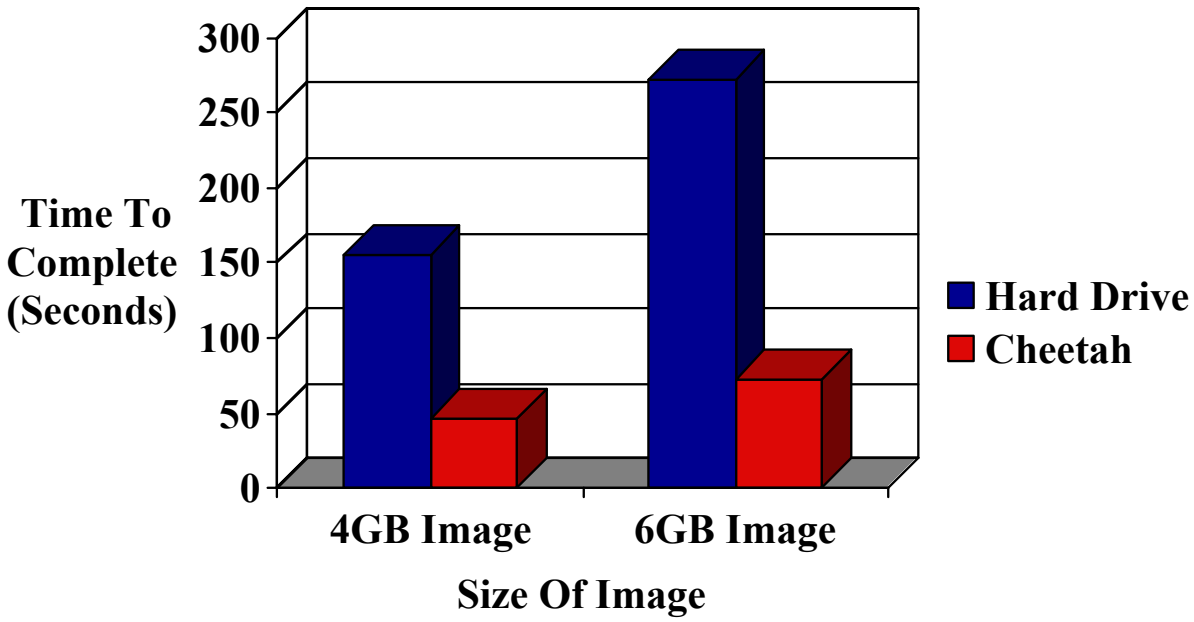

Figure 5.8 A chart showing the performance comparisons from using the Unix MD5Sum tool

5.3.6 Results Correlation

The IOzone benchmarking results showed to have a better performance increases than the test applications. This proves that Cheetah's increase in performance is most apparent for

applications that are highly I/O bound. The test applications had trivial increases in performance by using Cheetah, but results clearly showed that these applications were using other resources as well as the disk.

5.3.7 Comparative Evaluation

To place our results in the context of previous work we compare them with respect to NRD [9], which has the closest goals and performance metric to ours. A direct head-to-head comparison is not possible due to the varying technologies used, so our main basis for comparison is *efficiency*. One way to measure efficiency is to compare how well does each of the two implementations realize that available network bandwidth.

For sequential read/write operations, Cheetah utilizes 71 and 74%, respectively, of the theoretical maximum of 1GB/s. On the other hand, the respective numbers for NRD we derive to be 22 and 25%, for the 10 Mb/s Ethernet quoted. We deduce the sequential NRD read throughput from the performance for the "find" tool presented in Table 2 [9]; a 28 MB read is completed in 104 seconds (~276KB/s). The sequential write performance comes from Figure 11 [9], which shows a 30 MB sequential IOzone write to take about 100 seconds (~307 KB/s).

Clearly, other factors play into these end-to-end performance measurements – quality of hardware, NIC drivers, TCP/IP stack, etc. However, they cannot account for the threefold improvement in efficiency. Further proof can be found in the fact that NRD achieved only 25% improvement in sequential read performance relative to the hard drive. If we extrapolate for a network that is 4 times faster than the HDD (our setup) the speedup would not exceed 100%, whereas ours stands at 234%.

5.3.8 Summary

Our benchmark results have shown that the typical read() and write() file system calls can be sped up about 3.5 times faster than a commodity IDE hard drive. By running these tests on a commodity gigabit network, we have achieved speeds that were approximately 90% of what was achievable on our network's bandwidth. We have successfully manage to meet our original requirements:

1. *LAN Scalable*

   This requirement was met by using RAM from all of the available machines in our lab together to form the block device.

2. *Commodity Solution*

   Since our entire hardware setup consisted of a commodity gigabit switch and commodity machines, this requirement was clearly met.

3. *Lightweight*

   The user manual for using Cheetah consists of only 2 commands for starting the block device, and one command for starting the cache servers. If desired, a Unix script can be written to perform these operations automatically. Thus, Cheetah is very lightweight and easy to deploy.

4. *Digital Forensics Support*

The distributed block device is below the file system layer in the system architecture and its implementation details are transparent to the file system layer, allowing for multiple file system support. Forensic applications are also not limited, since the device details are transparent to them as well.

Chapter 6: Conclusion and Future Work

In this thesis, we presented a practical solution for sharing of RAM resources on a commodity gigabit cluster. The solution is based on a system containing a distributed block-level device and its connected cache servers called Cheetah. Unlike previous work, our solution is targeted at improving sequential read/write operations, which are the dominant disk access pattern. Our experiments show that sequential read/write operations can be sped up approximately 3.5 times relative to a commodity IDE hard drive. Furthermore, this speedup is approximately 90% of what is practically achievable for the tested system. To achieve this performance, we employ an adaptive read-ahead scheme that exponentially expands the read-ahead window during sequential reads.

Relative to previous work, our system is approximately 3 times more efficient in its ability to use available network bandwidth and is able to utilize 71-74% of the theoretical LAN capacity. For random access patterns, the measured speedup is over 20 times. Thus, for mixed loads, such as the ones experienced on a server, the speedup can significantly exceed the baseline 3.5 factor.

From these results, we show the effectiveness in digital forensics of Cheetah achieving a higher performance than the average hard drive. This is the most important goal of Cheetah, and the main purpose of the project. A popular problem that digital forensic investigators often encounter is that there is a great amount of precious time wasted while waiting for the digital forensic application to finish processing a disk, but due to thrashing and heavy loaded resources, performance penalties become apparent quickly. The only results available are the ones after the processing is done, and they could arrive too late. We wanted to downsize the time waiting for results by lessening the end-to-end latency for disk processing.

Theoretically, Cheetah's block device can be scaled up to a 2 TB maximum size, although this has not been tested since our lab only had around 10GB of free RAM. Otherwise, the amount of RAM available on the LAN determines the maximum space allocated for the distributed block device. RAM is a commodity piece of hardware, its speeds are much faster than disk speeds, and its capacity is larger than CPU cache, making RAM a very useful resource in our system. The problem of small capacity in RAM can be solved by distributing all of the available RAM on the network, which could lead up to a much bigger storage domain. Networks that have high-speed clusters greatly benefit in Cheetah's architecture. In order to achieve efficient performance, a gigabit network is needed. Fortunately, gigabit LAN speeds are getting more common and cheaper among households, businesses and institutions.

Cheetah is lightweight and designed to operate without difficulty for an average computer user. One of the main goals from the start of the project was to give digital forensic investigators less time trying to figure out how to use our system and more time using it to efficiently solve digital forensic cases.

With the capability of handling multiple file systems and the support to run any application, this should give many options to investigators. Since Cheetah deals only with the block device layer in the operating system architecture, it is completely transparent to any applications on the application layer.

Future Work

Since Cheetah is only a prototype, there can be many improvements for future work. Some improvements include robustness, multi-threaded reads, a forensic file system, and an upgrade for the 2.6 Linux kernel. Also, more testing of the system is necessary for use in fields other than digital forensics.

The robustness of Cheetah is presently not up to par for commercial use. If one of the cache servers shuts down while in use, then the module will likely crash and freeze the Linux kernel. All of the cache servers must be shut down before the module is restarted.

The adaptation of block reading gains performance in some tests, but overall performance can be greatly improved if the module didn't have to wait to get the blocks back. If there were a thread that retrieved and queued blocks together, then sending them to the system, the module would never have to wait for anything thus improving speeds.

When reading sequential blocks, the device really has no idea which file it is currently processing and thus has no idea of how many bytes remaining in the file. If the block device knew more about the files it was processing (where the next blocks are, size of file, etc.), then there would be greater performance because the device could just cache the rest of the file depending on the file size. A forensic file system would greatly benefit the device, linking the block device layer with the file system layer.

Cheetah was developed for any 2.4.x kernel, while the current popular Linux kernel is the 2.6 series. Since the small number of changes between block device drivers are widely known, it would not be a difficult task to upgrade Cheetah for the 2.6 Linux kernel. Many Linux users argue that the 2.4 kernel is more stable for servers than the current 2.6 version, but it would still

be reasonable to make the change since many popular Linux distributions are now designed around the 2.6 kernel.

We have tested with multiple applications including digital forensic tools, but it would be comforting to test with many other applications, including more forensic applications. These applications should be disk intensive to achieve the highest results. Some possible application areas include Bioinformatics and Geographic Information Systems (GIS). By testing more it is possible to find more audiences who could find Cheetah to be useful in their work.

References

[1] V. Roussev and G. Richard. “Breaking the Performance Wall: The Case for Distributed Digital Forensics”. In *Proceedings of the Fourth Digital Forensics Research Workshop*, (DFRWS) 2004.

[2] D. Patterson, “Latency Lags Bandwidth”, Communications of the ACM, 47(10), 2004.

[3] M. Dahlin et at. “Cooperative caching: Using remote client memory to improve file system performance”. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994.

[4] A. Acharya and S. Setia. “Availability and utility of idle memory in workstation clusters”. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1999.

[5] L. Xiao, X. Zhang, and S. A. Kubricht. “Incorporating Job Migration and Network RAM to Share Cluster Memory Resources”. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, 2000.

[6] J. Oleszkiewicz, L. Xiao, and Y. Liu. “Parallel Network RAM: Effectively Utilizing Global Cluster Memory RAM: Effectively Utilizing Global Cluster Memory”, In *Proceedings of the 33rd International Conference on Parallel Processing*, 2004.

[7] S. Ghemawat, H. Gobioff, and S. Leung. “The Google File System”, In *Proceedings of 19th ACM Symposium on Operating Systems Principles*, 2003.

[8] J. Liu, J. Wu, and D.K. Panda. “High Performance RDMA-Based MPI Implementation over InfiniBand”. *International Journal of Parallel Programming*, 32(3), 2004.

[9] M. Flouris and E. Markatos. “The Network RamDisk: Using remote memory on heterogeneous NOWs”. Journal of Cluster Computing, 2(4): 281-293, 1999.

[10] Kangho Kim, Jin-Soo Kim, and Sung-In Jung. “GNBD/VIA: A Network Block Device over Virtual Interface Architecture on Linux”. *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, 2002.

[11] G. Richard and V. Roussev. “Scalpel: A Frugal, High Performance File Carver”, In *Proceedings of the Fifth DFRWS*, 2005.

[12] S. Liang, R. Noronha and D.K. Panda. “Swapping to remote memory over InfiniBand: An Approach using a High Performance Network Block Device”, *Proceedings of the IEEE Cluster Computing*, 2005.

[13] L. Iftode and J. Singh. “Shared Virtual Memory: Progress and Challenges”, In *Proceedings to the IEEE*, Vol 87(3), 1999.

Vita

Daniel Tingstrom was born in Thibodaux, Louisiana in 1982. He received his Bachelor Degree in Computer Science from University of New Orleans in May 2004. He started his graduate program in June 2004 and became a teaching assistant instructing labs (CSCI 1581 and CSCI 2121) and also a lecture course (CSCI 1583). He completed his studies in August 2005, and currently does research at ATC-NY in Ithaca, New York as a Computer Scientist.