2006

Latency Reduction Techniques for Remote Memory Access in Anemone

Mark Lewandowski

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

LATENCY REDUCTION TECHNIQUES FOR REMOTE MEMORY ACCESS IN ANEMONE

By

MARK LEWANDOWSKI

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2006

The members of the Committee approve the Thesis of Mark Lewandowski defended on December 12, 2005.

Kartik Gopalan
Professor Directing Thesis

Ted Baker
Committee Member

Sudhir Aggarwal
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

I would like to dedicate this work to everyone who has supported me in this pursuit. First, I have to thank my parents. Their love, support and patience with me has given me the chance to do something great. Next, I have to thank my brothers Steven and Greg. Even though they didn't write this paper for me, they still deserve some thanks. Finally I have to thank Kim. It is her support that got me through this whole ordeal. Without her this paper wouldn't have happened.

ACKNOWLEDGEMENTS

I would first like to acknowledge my committee members, who have always stood behind me in my pursuit of a higher education. I would particularly like to thank Dr. Baker for understanding my commitment to completing my thesis, while supporting me on NSF Grant: 0509131. The opportunities he has continued to provide me throughout my career at Florida State never cease to prove themselves invaluable.

Second I would like to thank Dr. Aggarwal for his valuable insight. I can always count on him to expect my best work.

Next, I can't say enough for the continued help and patience that Michael Hines has showed me throughout the life-cycle of this project. I would like to refer to Mike as the father of Anemone, but on the slight chance that that makes me the mother, I will refrain. Even on those off days, when both of us had spent 48+ consecutive hours in the LENS Lab, Mike's patience with me never wavered. I could always count on him to push the development of this project when no one else would.

Last, and certainly not least, I have to especially thank Dr. Kartik Gopalan for everything he has done for me since I first enrolled in his first class at FSU (my first graduate level course), Advanced Unix Programming. Not only has Kartik been instrumental in the development of this project, but he has shown more patience and dedication towards the improvement of myself and my academic goals (even the delusional ones). Even our epic strife apropos reducing one's vocabulary to a small set of acceptable monosyllabic grunts, of which only simple subject-predicate combinations are considered admissible, has proven a useful edification (It improved my technical writing skills.). Thanks to you I have become a better research scientist, and more scholarly under your tutelage.

— Mark

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Memory system hierarchy has remained unchanged for many years, leading to a growing gap between main memory access times and a local disk's paging latencies. This trend has especially become a performance bottleneck for memory intensive applications. These applications can quickly eat up all available main memory, forcing the kernel to start swapping to the disk. One solution to this problem is to insert a new level – the remote memory – in the traditional memory hierarchy between local main memory and local disk.

Earlier work on the Adaptive NEtwork MemOry engiNE (ANEMONE) system demonstrated that remote memory access is a viable and attractive solution to this problem when the paging process exhibits a random block access pattern. This thesis evaluates the network communication latency of the Anemone system using three mechanisms: 1) a kernel-level lightweight reliable datagram protocol to replace NFS, 2) an aggressive page acknowledgment policy, and 3) a two-level caching mechanism. Collectively, these three techniques reduce the average network paging latency from 800 microseconds to 500 microseconds and speed up the average application execution time by a factor of 3 to 9.

CHAPTER 1

INTRODUCTION

Input/Output (I/O) has long remained the primary bottleneck in most modern computer systems. Over the last 10 years the computer industry has produced significant improvements in processor speed and memory access times, but only minor improvements in disk latencies. This has turned out to be a growing performance gap with an ever increasing effect on current applications. One popular analogy compares memory access times in a system to time taken to prepare a meal. In this case accessing on-die cache is like going to one's pantry and getting a snack. When the data cannot be found one must access main memory. This could be compared to going to the store and buying groceries. Finally, if the correct data still cannot be found in main memory, one must page out to disk. This is more like growing the food for oneself.

1.1 Motivation

The Adaptive Network Memory Engine (ANEMONE) system [1] is an effort to minimize the performance impact of page faults on execution times of memory-intensive applications. This system is built on the insight that paging out over a high speed gigabit network can be faster than paging out to disk. Earlier work in [1] demonstrates that ANEMONE can deliver significant speedups in execution times for multiple concurrent memory-intensive processes. However, for single process memory-intensive applications, ANEMONE delivers execution times that are competitive but not significantly faster than paging to local disks with on-board caches.

It has been shown in the past that such a system can be successful for a machine performing multiple memory needy applications at one time, but has so far been unable to improve process performance for a single process experiment.

1.2 Problem Statement

The goal of this research is to improve ANEMONEs performance for both single and multiple processes, by devising new latency reduction techniques. After analyzing data from experiments with the first version of the ANEMONE system it became evident that the primary sources of read and write latencies are the client-side Network File System (NFS) protocol implementation and the absence of page caching in ANEMONE.

Typically most commodity disks today have average seek times anywhere between 4ms - 11ms. This is slightly misleading. When most applications make requests for pages that are not contained in main memory, they request some number of sequential pages. Disks have a high average seek time due their mechanical parts. When a read or write request arrives to a disk, the disk must first move its head to the correct cylinder, then wait for the platter to rotate to the correct sector before the disk can begin transferring data. However, once the head has begun a transfer the disk performs very well for sequential reads and writes. The performance of disks can be further improved by use of on-board caches, typically between 8MB - 16MB, that now come standard on most inexpensive commodity disks.

1.3 Summary of Contributions

This thesis proposes fundamental changes in the design of the ANEMONE system to reduce paging latencies and improve both single and multiple process execution times. The specific contributions are as follows: 1) a lightweight Remote Memory Access Protocol (RMAP), 2) a two level caching scheme, and 3) an Early Acknowledgment system. Implementations of the new ANEMONE system have show that it can outperform the old ANEMONE system by up to 120%. This is enabled through large latency gains against disk and NFS writes.

A two level LRU caching system has also improved the performance of ANEMONE. Cache hit rates regularly hit up to 20%, and early trials suggest that higher hit rates can be achieved through aggressive prefetching.

1.4 Outline of Remaining Chapters

The following sections are laid out as follows. Section 2 discusses the background behind this project, and related research in remote memory paging. Sections 3 and 4 discuss the ANEMONE system's architecture and implementation details. Section 5 outlines

the experimental results of this work. Section 6 looks to the future and discusses some possibilities for further analysis. Finally Section 7 summarizes the work done on ANEMONE, and concludes the paper.

CHAPTER 2

RELATED WORK

The idea of remote memory is not new. Extensive studies have been done on the subject. This chapter outlines some of the common performance enhancements utilized in other projects.

Some efforts to utilize remote memory involve modifying the operating system. The Global Memory System[2] (GMS) assumes a reliable network, and is implemented on a network with built in congestion control to prevent most packet loss. In contrast, ANEMONE is built on a commodity gigabit ethernet LAN, where packet drops due to congestion have been observed.

The Remote Memory Model [3] architecture also modifies the operating system, but chooses to ignore kernel sockets and instead implements a lightweight data transfer protocol. This allows data to bypass unnecessary latencies added to data transport through kernel network stacks. Additionally, this protocol guarantees in-order delivery, a feature that is not necessary in remote paging. ANEMONE does not guarantee in-order delivery in its protocol.

Another popular remote memory strategy involves building complex application interfaces, requiring programs to be written to capitalize on remote memory benefits. Both Ageis[4] and Dodo[5] provide application level libraries for access to remote memory. Ageis improves remote memory performance by directly altering the translation look-aside buffer. Dodo implements a central memory manager daemon, similar to ANEMONE's memory engine. Unlike ANEMONE, Dodo's memory manager does not touch data directly. To access remote memory a client must query the memory manager, then query the memory server. The Pegasus [6] system and, the distributed shared memory version of Cashmere [7] also provide application level libraries for programmers to utilize. This requires software to be engineered to make use of remote memory paging. ANEMONE does not embrace this philosophy, and instead opts for a system transparent to existing applications.

Reliable Remote Memory Pager[8] (RRMP), Network Ramdisk[9] (NRD), and Samson[10] use client block device drivers to provide an interface between the client and the remote memory. RRMP and NRD do not use this to implement a custom protocol, and instead choose to execute through TCP/IP. The Network Ramdisk focuses on reliability in contrast to focusing on speed like most other work. NRD uses data replication and adaptive parity caching to ensure data consistency. ANEMONE also performs data replication when data reaches its engine cache.

Samson chooses to use a custom protocol built on top of the Myrinet link layer protocol. Samson utilizes extensive operating system and device driver modifications to implement a dedicated memory server. The Samson memory server communicates directly with the client; there is no engine in the Samson hierarchy. Samson does implement a LRU-approximation client side cache similar to ANEMONE's pure LRU based cache. The Samson project relies on a timer to check cache entry timestamps to determine if they should be removed from the cache. ANEMONE's use of a FIFO queue to track LRU pages is more subtle, but runs in O(1). Samson does not implement a memory engine, so it does not include a second cache.

The Berkeley NOW project[11] performs cooperative caching through a global cache file, but does not directly address remote memory paging. [12] and, [13] both propose systems based on NOW, but focus on extending the existing system and not performance.

CHAPTER 3

ARCHITECTURAL DESIGN

This chapter discusses the factors that affect the performance of the Anemone system and the design of three latency reduction techniques proposed in this thesis.

Section 3.1 discusses the original design of the ANEMONE system. Section 3.2 compares and contrasts the pluses and minuses of the original design decisions. Finally, Section 3.3 describes the design of three latency reduction techniques proposed in this thesis and their justifications.

3.1 Original ANEMONE Architecture

We can visualize a typical virtual memory system as a hierarchical pyramid, with each level representing a different memory subsystem. The pyramid is organized in such a way that the faster memory types are located towards the top of the pyramid, and slower types nearer the bottom. Furthermore, the largest capacity memory subsystems are towards the bottom, and the smallest are closer to the top. The standard hierarchy of a virtual memory system is represented in Figure 3.1.

The original ANEMONE system inserts another layer of remote memory between main memory and disk, as shown in Figure 3.2. In this modified hierarchy, remote memory is slower than local main memory, but faster than local disk.

Figure 3.3 shows the system architecture of the Anemone system. There are three main components in Anemone: 1) Memory Engine, 2) Memory Client and 3) Memory Server. The rest of this section describes each of these components.

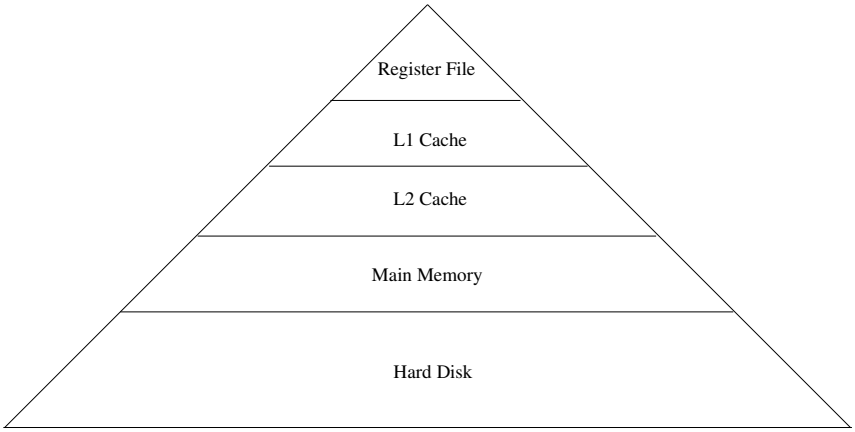

Figure 3.1: Standard hierarchy of a typical virtual memory system

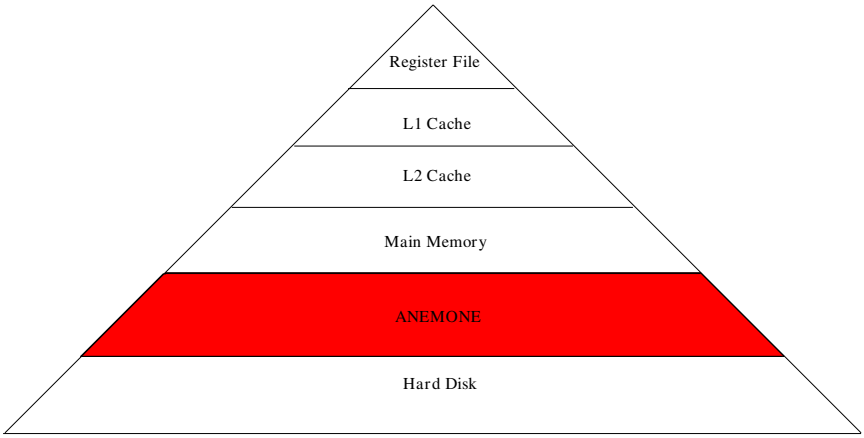

Figure 3.2: The ANEMONE system's virtual memory hierarchy

3.1.1 Engine

The Memory Engine is implemented as the primary control entity in the ANEMONE system. It exists to manage the interactions between client machines and the remote memory pool. The engine occupies a centralized server role for simplicity reasons. Once system feasibility has been shown, the centralized server role will be phased out in favor of a fully distributed architecture. When this happens, the client and server modules will have to be extended to

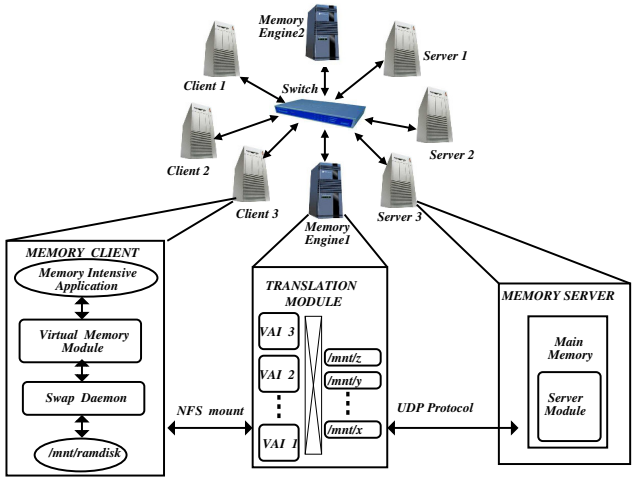

Figure 3.3: The original ANEMONE system's architectural design

include the engine's functionality.

To communicate with a client, the engine must use its own implementation of the Network File System protocol (NFS) [14]. When a client connects to a Memory Engine it expects access to a pool of memory resources. The engine does not provide these resources directly, but instead maps free memory from the server memory pool to a Virtual Access Identifier (VAI) that has been allocated specifically for the client in question. Each VAI can be thought of as a logical device allocated to allow access on remote memory. Abstraction to the VAI level allows the device to transparently handle read and write operations for memory mapped from multiple servers.

The Memory Engine acts as the central authority in the ANEMONE architecture. It is charged with managing the individual VAIs, and controlling their interactions with the registered memory pool. Clients are able to interact with individual VAIs which transparently virtualizes remote memory.

3.1.2 Client

The primary concern for the ANEMONE client is to transparently handle client-engine communication with a well defined interface to the client operating system. NFS, which

arrives bundled with commodity operating systems, provides such an interface. [1]

Adding this layer of abstraction to the client allows memory intensive applications to operate as if oblivious to the presence of the ANEMONE system. For instance, running STL quicksort on a large array does not require any modification of existing code. Instead, the application counts on the operating system's swap daemon to efficiently handle all swap interactions between the application and the NFS mounted file.

3.1.3 Server

The Memory Server stores the client pages in its main memory. Upon initialization a server designates a portion of its own memory to contribute to the ANEMONE cluster. This memory is considered reserved, and its data cannot be paged out. When a server registers with the Memory Engine it tells the engine how much free memory it is contributing to the cluster. The contributed memory can be multiplexed across many VAIs, which act to hide information about the client from the server.

A server effectively acts as a dumb node in the system. It maintains no knowledge of any clients or other servers, relies on the engine to perform any load balancing, and only performs minimal accounting to maintain pages correctly. It provides no supplementary computational power. The server's only requirement is to store and retrieve data as quickly as possible. It communicates through a reliable, stop and wait, UDP based protocol with the engine.

3.2 Pros and Cons

Previous work on ANEMONE has shown viability of a large scale, clustered, memory virtualization system to improve the performance of large-memory applications. The first implementation of ANEMONE demonstrates a performance increase in multi-process experiments [1]. The flexibility provided to the system through its reliance on the NFS protocol is a strong advantage, as it does not limit client machines to a specific operating system. Any OS that provides support for NFS swapping is able to connect to the ANEMONE system, including SunOS, FreeBSD, Linux 2.4, and even Windows.

[1] The implementation of NFS in Linux requires a kernel patch to permit swapping operations using NFS mounts.

Although the ANEMONE system shows promise against disk systems running multiple large memory processes, it has failed to best these systems when executing a single large memory process. This can be attributed to the different memory access patterns in the two situations. A system running a single application tends to produce page requests in a sequential manner, while a system running multiple applications accesses pages in a random fashion. ANEMONE fails to take into account small, highly effective on-disk caches that come standard on most disk devices today. These caches are designed to exploit sequential access patterns like the ones seen in single large memory processes.

While its on-disk caches are effective for boosting the disk's performance on sequential reads, the cache's greatest achievement is in improving random write times. In initial ANEMONE studies it has been shown that a random access write can be sped up by a factor of 400 - 700 by including an on-disk cache. This means that a typical random disk write operation that used to take ∼4000 microseconds can now be completed in ∼10 microseconds. This improvement also holds for sequential writes, however the performance improvement is not nearly as significant. The largest disk performance increase comes from eliminating I/O wait time while the disk performs some I/O. Through use of the cache the disk is able to signal to the OS that an I/O operation has completed before the data is actually written to the disk. Disks used to accrue rotational delay and disk head seek penalties for every I/O, but the addition of an on disk cache has eliminated this.

The original ANEMONE architecture is also slowed down by network communication overhead. The IP stack can be blamed for some of the delay, but use of the NFS protocol produces the largest penalty. The standard NFS implementation is far too heavy to provide the performance that ANEMONE is striving to achieve. This is due to the protocol's complex embedded flow control, and excessive reliability assurance. Eliminating the NFS protocol limits ANEMONE's transparency to client operating system, but the benefits gained from implementing a lightweight communication protocol far outweigh the advantages that NFS provides. Chapter 5 elaborates on performance improvements as ANEMONE transitions away from NFS.

We recorded the traces of paging activity of the swap daemon for one execution each of POV-ray and Quick-sort. Figures 3.4 and 3.5 plot a highly instructive plot of the paging activity count versus the byte offsets accessed in the pseudo block device as the application execution progresses. The graphs plot three types of paging events: a write (or page-out),

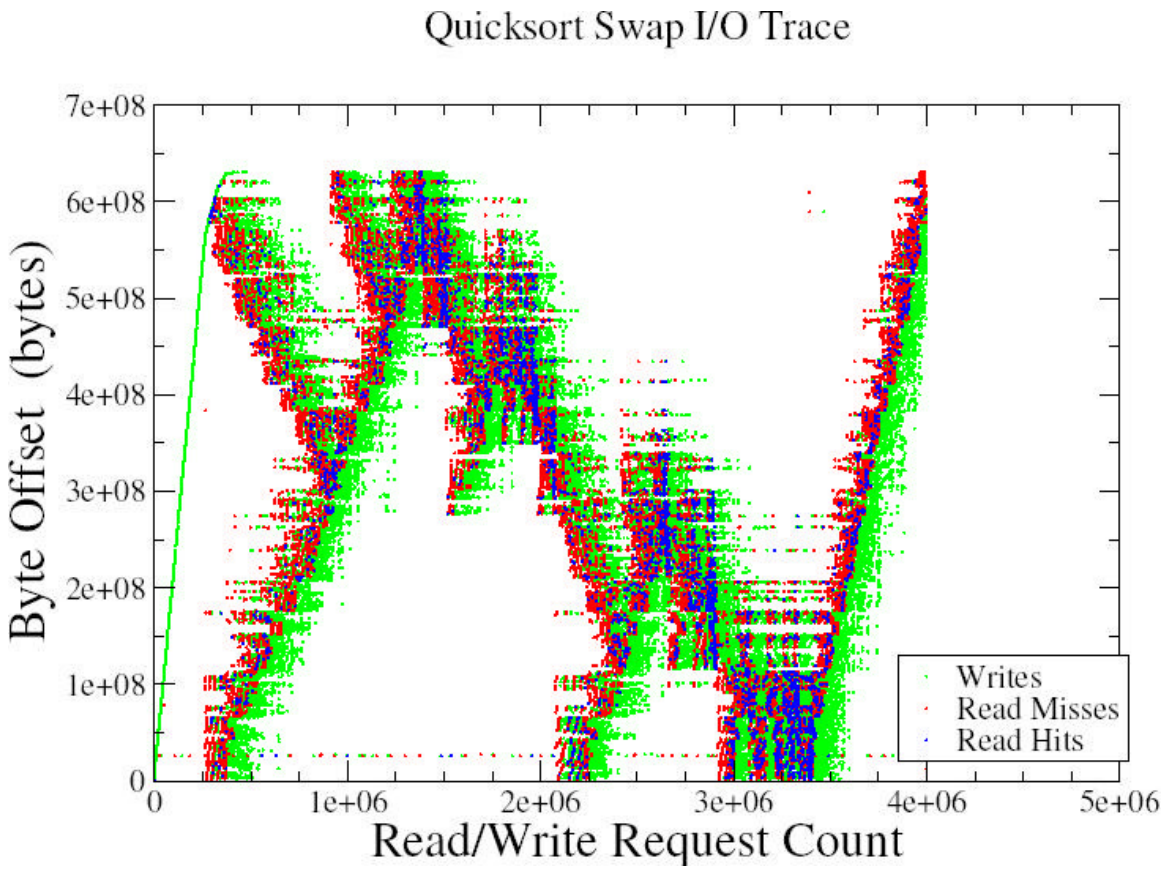

Figure 3.4: Quicksort memory access patterns

read (page-in) hits in the cache, and read misses. Both traces show signs of regular page access patterns. The level of sequential access pattern seems significantly higher for POV-ray than it is for Quick-sort. Upon close examination however, we find that there is significant reverse memory traversal pattern in POV-ray, even though it is sequential. On the other hand, Quick-sort has a tendency to access non-local regions of memory due to the manner in which it chooses pivots during the sorting process. This seems to indicate that both applications tend to defeat any read-ahead prefetching performed by the swap daemon, but suggests that an advances caching and prefetching scheme will improve performance.

3.3 Proposed Latency Reduction Techniques

Section 3.2 outlines a number of weaknesses in the original incarnation of the ANEMONE system. It is apparent that these shortcomings must be addressed to enable the system to

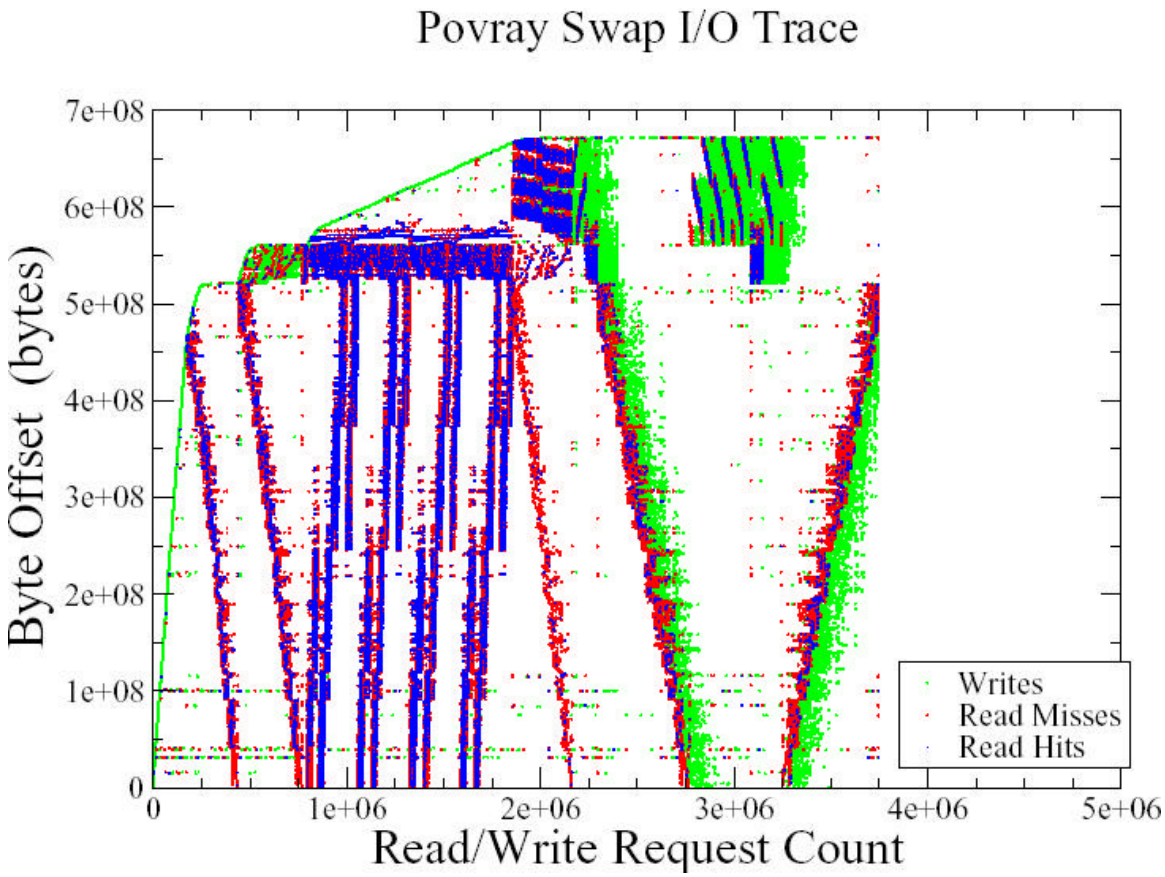

Figure 3.5: POV-Ray memory access patterns

outperform a disk-based swapping scheme. The following subsections outline the steps taken to overcome the shortcomings of the original ANEMONE system.

3.3.1 Remote Memory Access Protocol

The original ANEMONE system appears to interact with clients as a black box NFS server in order to permit clients with any operating system to use Anemone. As a first latency reduction technique, this thesis proposes to replace the role of an NFS based client with a kernel module in the client specific to the Anemone system. Although this proposed solution moves away from the NFS protocol, the replacement protocol works to emulate NFS in a way that a user not concerned with optimal performance is still allowed to interface with the system through its NFS swap interface.

The decision to move away from NFS is really what liberated the ANEMONE system

from somewhat limiting constraints. Implementing a new protocol in a separate kernel module allows the client to bypass the somewhat slow processing of the Linux IP layer. The kernel module presents developers with the ability to implement a cache on the client side of the ANEMONE cluster. Utilizing this cache allows the client to better compete with cached disk writes. Section 3.3.3 outlines the improvements that were made to augment its overall performance.

To develop a better protocol suited for ANEMONE's needs it is necessary to examine the behavior of ANEMONE over the network. It quickly becomes obvious that many features NFS carries with it are not necessary to allow ANEMONE to function over a LAN. For instance, NFS supports close to forty different operations in the protocol, but the engine only implements a small subset of these, specifically only READ and WRITE. By reducing the number of commands that a customized protocol must handle it is possible to reduce the processing complexity. [14] requires that NFS implementations provide congestion control similar to that of TCP. This is a major obstacle to overcome, and guarantees that NFS will never perform much better than TCP, even when using NFS over UDP.

To eliminate unnecessary overhead ANEMONE introduces a light weight *Reliable Memory Access Protocol* (RMAP). RMAP takes measures to reduce the factors that slow NFS. RMAP is a window based reliable protocol. It does not guarantee in-order delivery, because block devices typically do not guarantee in-order delivery. Only five commands are implemented at this point, REG, UNREG, READ, WRITE, and STAT, but care has been taken to maintain backwards compatibility with NFS.

3.3.2 Early Acknowledgments

In the initial implementation of the ANEMONE memory system, the time for a client to send a page out/page in request was observed to be ∼500 microseconds. This is due to an overly conservative acknowledgment system. The client machine spends large amounts of time waiting for outstanding network traffic. To complete a page out operation the client sends a write request to the memory engine, waits for the data to be propagated to the Memory Server where an acknowledgment is generated, then has to wait for the acknowledgment to be propagated back up to the client. Clearly this is unnecessary, since the engine receives a copy of the page from the client before forwarding it onto the server.

Allowing the engine to generate and send an acknowledgment immediately to the client

not only reduces the latency seen by the client, but it also offloads the retransmission responsibilities onto the engine. To handle client writes in an optimal fashion, the engine generates and sends the client an ACK upon reception of the client data, storing a local copy for retransmission until it receives an acknowledgment from the server.

3.3.3 Two-level Cache Design

In comparrison to the performance of the initial ANEMONE system versus the performance of traditional disk based page swapping, ANEMONE was not providing the performance that was initially expected. Upon closer examination of read/write latencies for ANEMONE and for disk it was observed that on-board caches on disks were greatly reducing the latency seen during writes.

When the Linux operating system tries to write a page to disk it sends the data to the disk's device driver, which then forwards the information on to the disk hardware. Originally ANEMONE expected the data to be written to the physical disk before returning a completion message to the operating system. The introduction of a cache on the disk makes that assumption false. Instead, the disk cache returns a completion message as soon as the cache receives the data, queuing it to be written to the disk at a later time. This pattern is observed for many writes, until either the cache deems it necessary to stall and write to the disk, or has enough time between disk accesses to flush the cache.

After comparing the latencies between disk systems ($\sim$7 microseconds), and ANEMONE ($\sim$40 microseconds) it seemed necessary to include a method to reduce this performance gap. A simple method for reducing page lookup times is to introduce caching throughout the ANEMONE system. Two levels of caching are introduced, one at the client and one at the engine. Both levels are built from the same foundation, but function slightly differently.

The basic ANEMONE cache design uses a FIFO queue to track the use of pages in the cache. When a page is inserted in the cache, it is inserted at the tail of the FIFO queue. As succeeding pages are inserted, the original page moves towards the head of the queue. If a page is referenced while it is in the cache, its reference in the queue is removed, and reinserted at the tail. When a page needs to be evicted from the cache, the page at the head of the queue will be the least recently used page, and is removed.

The cache also employs a hashtable for quick indexing of the cache. On a read or write, it is important to search the cache for the page. The hashtable is used to allow quick indexing

of the cache, avoiding a linear search of the FIFO queue. When a page is inserted or removed from the FIFO queue, it is correspondingly inserted or removed from the hashtable.

Client Cache Design

The client cache is built to be a small cache to improve temporal locality hit rates. It retains the most recently used pages for a small number of pages (typically 4 MB - 16 MB). The client cache operates as a write-back cache in an effort to reduce client-engine network traffic.

Memory Engine Cache Design

The engine cache operates as a write-through cache, and is much larger than the client cache. The engine cache can be as large as several hundred megabytes since main memory in the engine is not under the same pressure as the client's main memory.

3.4 Summary

This section has outlined the proposed changes to the ANEMONE system. First, eliminating the NFS protocol will effectively reduce network communication overhead. This provides a significant speedup on client-engine communication. Second, using early acknowledgments to shortcut writes (Page-Out requests) will reduce ANEMONE's latency on a per-write basis. A two level caching architecture will also reduce network traffic, and speed up read and write latencies. Finally, the introduction of the client module will eliminate ANEMONE's dependence on rarely updated kernel patches.

Chapter 4 will further discuss the implementation issues behind these techniques, while Chapter 5 will show the performance gains of each solution.

CHAPTER 4

IMPLEMENTATION

This chapter describes the implementation of the ANEMONE architecture presented in the previous chapter. Section 4.1 discusses the development and configuration of the ANEMONE testbed. Section 4.2 discusses the implementation of a lightweight remote memory access protocol (RMAP), section 4.3 outlines the ideas in the ANEMONE cache hierarchy, section 4.4 details the implementation of the Early Acknowledgment strategy, and finally section 4.5 describes the implementation of the ANEMONE client module.

4.1 Testbed Setup

The Anemone system is built over a cluster of machines with a wide range of memory capacities. Table 4.1 shows the configuration of each of the 8 machines in the ANEMONE testbed. It is important to notice that each machine in the testbed is equipped with an Intel Pro 1000 MT gigabit network card capable of transmitting jumbo frames (greater than 4KBytes per frame). Standard ethernet frames are too small to transmit an entire page in one 1500 byte frame. ANEMONE uses jumbo frames to fit one page into each frame.

The machine with the smallest amount of physical memory (anemone4) is designated as the client. This maximizes the effect swapping has on the machine. In the real world the client machine may have a substantially larger amount of memory. Running tests in this configuration is designed to highlight measured performance difference between ANEMONE and disk based systems.

The machines are all connected through a SMC EZSwitch 8508T gigabit ethernet switch. This switch is chosen because it is an affordable gigabit switch that supports jumbo frames.

During periods of high network traffic, the e1000 network driver may generate a high volume of interrupts. This can effectively overwhelm the CPU and slow down the

Table 4.1: ANEMONE testbed

Machine Name	Processor	RAM	NIC	Role
anemone1	Pentium 4 2.6 Ghz	2.0 GB	Intel Pro 1000 MT	Server
anemone2	Pentium 4 2.4 Ghz	3.0 GB	Intel Pro 1000 MT	Server
anemone3	Pentium 4 (Hyper Threading) 2.8 Ghz	1.0 GB	Intel Pro 1000 MT	Engine
anemone4	Pentium 4 2.6 Ghz	256 MB	Intel Pro 1000 MT	Client
odo	Pentium 4 (Hyper Threading) 3.0 Ghz	1.0 GB	Intel Pro 1000 MT	Server
omo	Pentium 4 (Hyper Threading) 3.0 Ghz	1.0 GB	Intel Pro 1000 MT	Server
slavei	Pentium 4 (Hyper Threading) 3.0 Ghz	1.0 GB	Intel Pro 1000 MT	Server
pmp	Pentium 4 (Hyper Threading) 3.0 Ghz	1.0 GB	Intel Pro 1000 MT	Server

performance of the machine. The Linux kernel has the ability to package hold off on handling an interrupt until a group of interrupts have occurred. This will reduce the amount of CPU-interrupt interference observed on a given system. Interrupt coalescing will also produce higher latencies for network traffic, as packets will arrive at a destination and wait for some amount of time before being handled. It is more important to cut down on packet latencies for the ANEMONE system, therefore all machines in the testbed have interrupt coalescing disabled by default. In our experiments, we observe that interrupt rates are not a problem.

In this configuration, preliminary tests have yielded a throughput of 900 Mbits/second for systems with PCI NICs, and 944 Mbits/second for systems with on-board NICs.

4.2 RMAP: Reliable Memory Access Protocol

The translation module, shown in Figure 3.3, handles the client request processing at the engine. It needs to be fast so that page-fault latencies are kept short. The Memory Engine is dedicated solely to the task of handling memory requests. Use of TCP for reliable communication turns out to be too expensive because it incurs an extra layer of processing in the networking stack. In addition Anemone does not need the congestion control and in-order byte-stream services of TCP. Because Anemone operates within a LAN, there is no need for any network-layer routing functionality. Thus Anemone can avoid the extensive IP layer processing as well. The need for packet fragmentation is eliminated because gigabit networking technology provides support for the use of *Jumbo* frames which allows MTU

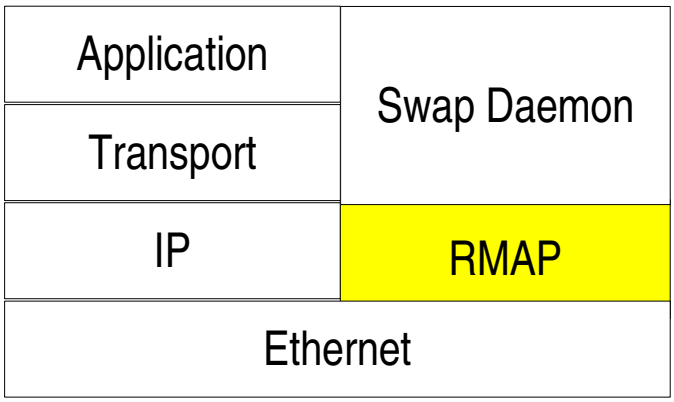

Figure 4.1: The ANEMONE network protocol stack

(Maximum Transmission Unit) sizes larger than 1500 bytes, typically between 9KB and 16KB. An entire 4KB or 8KB memory page, including headers, can fit inside a single Jumbo frame. Consequently, *Anemone uses its own window based reliable datagram protocol with both clients and servers, but without the requirement of in-order delivery.* To accomplish this, all the communication modules of Anemone (in the clients, engine and the servers) operate above the network device driver, as shown in figure 4.1, bypassing all other protocol stack layers.

The RMAP implementation includes five network functions. REG{ister} and UNREG{ister} establish/tear down a connection with the memory engine. When the engine receives a REG request it allocates a VAI and returns a file handle on it. The client can use this handle to communicate with the VAI for the duration of the connection with the server. As one would expect, when the engine receives an UNREG request it deallocates the VAI and all metadata associated with that client, frees remote pages belonging to those clients, and returns success to the client. READ, WRITE, and their associated ACKs provide basic reliability for reading and writing between the client and engine. A simple per-request timeout is used for retransmissions of READ/WRITEs. If a READ/WRITE communication does not receive its associated acknowledgment after a given amount of

time, the request is retransmitted. Retransmissions at the engine are also bounded, and if a request is retransmitted more than the retransmission bound it is silently dropped, leaving it up to the client to retry. Finally a STAT function is made available to allow the client to gather information on its corresponding VPI(s) in the engine, such as available memory and average workload.

A lightweight flow control strategy has also been implemented in the protocol. Upon loading the client module, a fixed size FIFO transmission queue is allocated within the module itself. As BIO requests arrive at the client module, the transmission queue is allowed to fill. When the queue is full, the client will stall any upper layer paging activity until space has been freed in the queue.

The transmission queue is allowed to transmit packets within a fixed length window. The window data structure encompasses a fixed number of packets in the transmission queue. When some data in the transmission queue is shifted into the scope of the window the data becomes transmittable. Once data contained within the range of the window has been transmitted, the data will remain within the range of the window until the data's acknowledgment arrives at the client. At this point the data will be removed from the transmission queue, freeing space in both the queue and the window's range.

It is important to note that acknowledgments may not arrive at the client in the order of transmission. The client is able to handle this situation flawlessly. One can think of the transmission queue as a stack of blocks. If the window size is set to n, the only blocks that are eligible for removal are within n blocks from the bottom. When a block is removed the higher blocks shift down one position. Thus, as long as there are requests (blocks) in the queue there are guaranteed to be 1 to n requests within the window's range. This strategy allows the window to be used to limit the number of outstanding transmissions. Future implementations of the client module may include dynamic resizing of the window, but this is not realized in the current RMAP protocol.

The RMAP implementation also provides four user tunable parameters; three timers and a window size. Below is quick summary on how each parameter can affect the overall client module.

Setting the window size adjusts the network transmission window's size. Care should be taken when adjusting this parameter to ensure the window size will not overwhelm the link or engine(s). If the window size is set too large, the client may overrun the network interface's

queue, causing packet drops. Increasing the window size has a tendency to increase the number of page retransmissions between the client and engine. Current experiments have found the optimal window size to lie somewhere between 10 and 12 page-requests. This appears to mainly be due to the swap daemon's prefetching strategy. This can be observed during the quicksort experiment, and will be discussed further in Chapter 5.

There are also three user configurable timers. The first is a timeout value, counted in jiffies (ms). Each time the timeout expires RMAP checks to see if there are any requests that need to be retransmitted. Setting this timer to a very low value can degrade performance of the system. Every time the timeout expires a utility function compares the current time with the timestamp of every request within the transmission windows range, retransmitting packets with older timestamps. Setting the timer parameter to too large a value will degrade the system in a different manner. This will increase the retransmission latency, causing requests waiting to be retransmitted to wait in the queue for an undesirable amount of time.

The second timer is the per packet retransmission latency. This specifies the minimum amount of time a request must wait in the queue before being retried. The larger this is, the longer it takes to begin retransmission of missed data. Larger values also increase the time the swap-daemon must wait for this page to be removed from the queue. Depending on the network and application workload, this must be chosen carefully. In a future implementation this may become dependent on the current request RTT.

Finally, the third timer is a spacing parameter that specifies how long to wait between any individual transmissions. Again, depending on the workload, this can put a lower bound on the paging-rate of the client, as it will upper bound the maximum transmission bandwidth. This timer can also be used to provide very basic flow control. Future work involves implementing this and the other four RMAP parameters as dynamically updating values.

4.3 Caching

Section 3.3.3 mentions that both the client and engine modules both include LRU-based caches. Both caches are engineered from the same basic foundation since they both provide similar functionality to ANEMONE, and are only separated from each other by one level of indirection. The basic ANEMONE cache implementation will be covered directly, while specific engine and client cache details will be discussed in Sections 4.3.1 and 4.3.2.

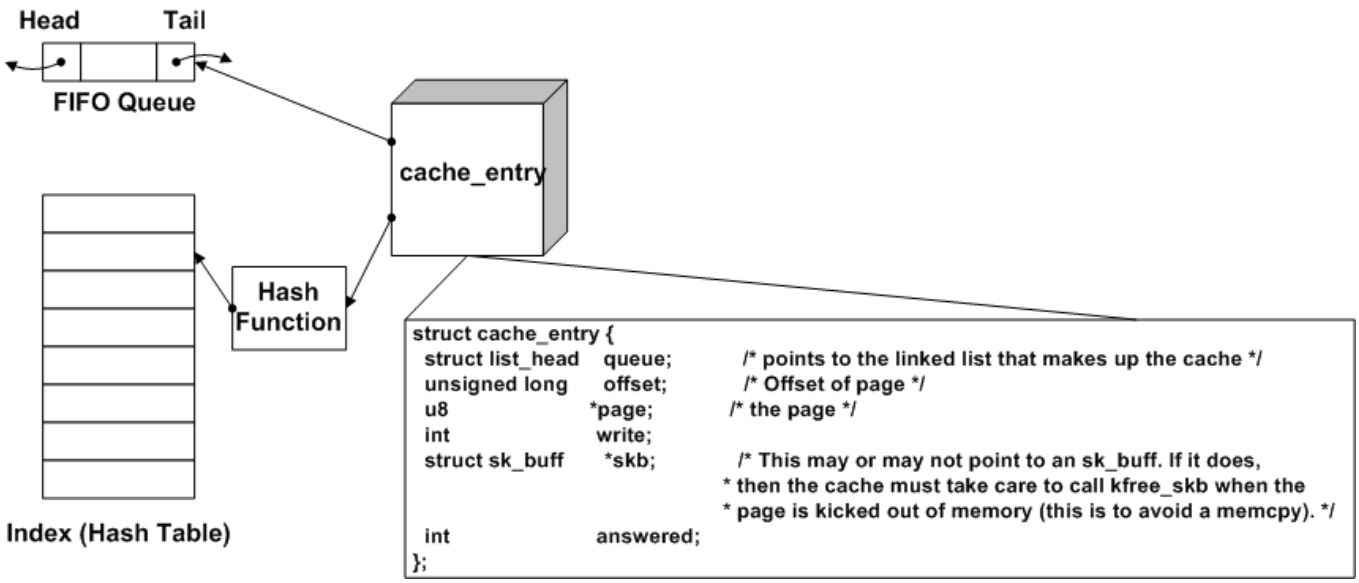

Figure 4.2: The ANEMONE client's cache architecture

ANEMONE's basic cache design implements both a FIFO queue and a hashtable. Figure 4.2 shows the structure of the basic design. The cache utilizes these basic data structures to allow rapid page lookup in two specific cases: 1) finding the LRU page in the cache, and 2) finding a specific index in the cache.

The cache uses a FIFO queue to keep track of page usage in the cache. The queue is based on a kernel implementation of a doubly linked list. When a page is inserted in the cache, a reference to the page is inserted at the tail of the cache. When the cache deems it necessary to evict a page, it can find the LRU candidate at the head of the FIFO queue. This allows LRU lookup times of O(1).

The cache also implements a hashtable data structure to quickly index stored pages. When a page is inserted in the cache, a reference is also stored in the hashtable. If inserting the page causes a collision in the hashtable, the buckets are stored as linked lists to allow multiple pages to be stored in each bucket. Since the hashtable allows collisions, its lookup time is $\Theta(1+n/m)$, where n is length of the linked list being searched, and m is the number of buckets. This improvement yields performance far better than the linear lookup time, O(n), of a linked list by itself.

The following subsections outline the differences between the two caches.

4.3.1 Client Cache Implementation Details

The client cache operates as a write-back style cache. It is typically between 4 MB -16 MB in size. The cache preallocates all memory on load, storing free cache entries in a linked list. Free resources are utilized in a LRU fashion. When a new entry is placed in the cache, free resources are allocated from the head of the free resource list. When a page is evicted the resources are placed back at the tail of the free resource list.

It is important to note that allocation of memory in the kernel is inefficient. If one were to allocate memory every time more memory was needed, the kernel's memory space would quickly become fragmented. it is much more cost effective to allocate all memory at once if an upper bound on the maximum amount of memory can be established ahead of time.

The ANEMONE client code takes advantage of this and allocates a large chunk of memory, that is locked into main memory (cannot be swapped out) at load time. Through intelligent memory use, a page's memory can be reclaimed by the block device when it is kicked out of the cache. Not only does this speed up the implementation of the client module, but it also allows the client to maintain a consistent memory footprint in the kernel. This is important because it allows other applications to see a constant amount of usable memory while they are running. Without this, the swap daemon would be forced to swap pages in and out of the cache needlessly.

4.3.2 Memory Engine Cache Implementation Details

The engine cache operates as a write through cache. This is because when a page arrives at the engine it already sits in a *sk_buff* structure, making it simple to allow the page to transmit to a server. The engine cache preallocates memory to store cache entry metadata, but does not allocate space for the pages themselves.

When the engine caches a page it stores an independent copy of the data, to preserve socket buffer state. This means that a new socket buffer must be allocated to store the copy. The function call *skb_copy* allocates and copies the socket buffer correctly. Without preallocation of these new socket buffers, the engine cache is susceptible to kernel allocation/deallocation problems. When a socket buffer is deallocated it is not freed by the kernel right away. This imposes a limit on the size of the cache. Future work will include avoiding this issue.

4.4 Early Acknowledgments

Implementation of an Early Acknowledgment system at the Memory Engine is easy to accomplish. The engine maintains a small buffer of sk_buff structures for exclusive use by the early acknowledgment system. The buffer is allocated when the module is loaded. The buffer is typically ∼20 sk_buffs in size. When a page arrives from a client a free page is claimed from the queue of free sk_buffs and is populated with information confirming the receipt of the page. The sk_buff is then explicitly sent back to the client. It does not interfere in any way with the normal processing of the engine.

4.5 Anemone Client Implementation

The memory client is designed to provide a transparent interface between the client machine's swap daemon and the Anemone memory cluster. It is loaded into the Linux kernel as a standard module, where it acts as a *pseudo block device* (PBD), i.e. providing a regular block device interface for accessing remote memory.

When the module is loaded it registers with the kernel's default block device interface. Standard block devices interact with the kernel through a request queue. The request queue provides the means for the kernel to arrange I/O requests in byte offset order, thus arranging reads and writes with similar offsets to be grouped together into one *request*. The request is then placed on the request queue using an elevator algorithm. As is mentioned in [15], kernel developers have taken extreme measures to optimize the request queue to fully exploit any sequential properties in a set of block I/Os (BIOs) in an effort to speed up disk response times. Unlike disks, Anemone is not reliant on sequential access properties to maintain maximum I/O transfer rates. Since it is built on solid state memory, access times remain constant for any I/O given to the block interface. As a result, the client module is implemented to bypass the request queue and instead deals directly with individual BIOs. The kernel provides block devices the ability to register their own function in the block interface to handle read/write requests. By intercepting BIOs before they are placed on the request queue, the Anemone client is able to skip any unnecessary processing that is normally associated with placing an I/O request on the queue.

Dealing with BIOs directly provides other advantages. It allows the client to handle BIOs in the order they arrive. Additionally some clever queuing of incomplete BIOs allows out of

order BIO completion. In contrast, to complete a similar action while using a request queue requires an additional queuing data structure, and a function to allow the module to break a request apart into individual BIOs.

The client module also includes several tunable parameters (outlined in section 4.2) that can drastically affect the performance of the client. Within the current implementation the user is allowed to specify the upper bound on the transmission queue. The purpose of the transmission queue is to receive BIOs from the block interface as quickly as possible. It is best to set this queue size to a value that is large enough where the swap daemon will not be able to keep it fully populated, however caution needs to be taken to keep its memory footprint at a minimum; usually a value around two hundred blocks is found to be acceptable. The size of this queue detracts from memory that should be available to current processes, so it is important to keep the queue's size to a minimum.

4.6 Summary

This chapter has provided a look into the implementation of the ANEMONE system's testbed and latency reduction techniques. The Remote Memory Access Protocol (RMAP) is a window based protocol with simple built in flow control mechanisms. RMAP does not guarantee in-order delivery. Both the Memory Engine and ANEMONE client module have caches built on a common LRU-based cache. The cache uses a hashtable for quick indexing, and a FIFO queue to track LRU entries.

While multi-level caching speeds up read requests, a simple Early Acknowledgment strategy improves the system's write request response time. Finally, the client module's *pseudo block device* (PBD) interface is shown to provide a transparent convergence between the client machine's virtual memory system, and ANEMONE.

CHAPTER 5

EXPERIMENTAL RESULTS

This chapter describes the performance changes observed in ANEMONE after implementing the latency reduction techniques described in Chapter 4. Section 5.1 describes the experimental setup used in the experiments. Section 5.2 discusses the changes observed in single read/write latencies for each technique implemented. Section 5.1.1 describes the test applications used to perform the experiments in sections 5.3 and 5.4. Finally section 5.5 discusses the performance of both levels of cache.

5.1 Experimental Setup

The experimental setup consists of setting up the Anemone cluster on the testbed described earlier. Machines anemone1, odo, and slavei are designated as memory servers who provide 200,000, 100,000, and 100,000 blocks respectively to the engine. Local machines anemone3 and anemone4 are specified as the memory engine and client. Table 4.1 shows these machine's physical hardware statistics. This provides an Anemone cluster with 400,000 pages in the memory pool. This roughly translates to ∼1.6 GB of remote memory.

Tests using the client module set the cache size to 16 MB, and for engine cache experiments the cache size is set to 80 MB. Section 4.3.2 explains the size restrictions of the engine cache. RMAP parameters are set accordingly:

- Window Size - 12
- Retry Timeout - 10 ms
- Packet Retransmission Timeout - 3 ms
- Spacing - OFF

Tests using the disk are configured on a Western Digital (WDC WD400BB-75FJA1) hard drive. Table 5.1 shows the disk's full performance specifications.

5.1.1 Test Applications

The second phase of experiments tests the ANEMONE system on real world applications. The test suite includes programs chosen to display a varying range of memory access patterns (Figures 3.4 and 3.5). All application experiments test disk against the original ANEMONE system and the augmented version (ANEMONE with RMAP, both levels of caching, and Early Acknowledgments).

Quicksort

The use of STL Quicksort[16] highlights ANEMONE performance for applications who's memory access pattern is not prefetched easily. This is the standard implementation included in the STL C library. In this experiment an array of randomly generated numbers is sorted using quicksort. The array size varies, depending on the experiment. Tests track the start and end times of the application, as well as the number of cache hits and misses for both the client and engine cache.

POV-Ray

POV-Ray[17] is a graphics rendering application that displays a more predictable memory access pattern. Experiments using this application render a preconfigured scene of 1/4 unit spheres. Start and end times are recorded, as well as client/engine cache hits and misses.

5.2 Per-Block Latencies

It is important to do a raw comparison between read and write latencies for Anemone versus Disk. Results based on Anemone and disk performance are displayed in Figures 5.1, 5.2, 5.3 and 5.4. These four charts plot cumulative distributions for observed read and write latencies on sequential and random I/O schemes. This experiment runs 100,000 random access, and 100,000 sequential access read/write requests on the disk and various configurations of the ANEMONE system, and compares their latencies. Experimental results are also summarized in Table 5.2.

Table 5.1: Western Digital (WDC WD400BB-75FJA1) performance specifications

Rotational Speed	7,200 RPM (nominal)
Buffer Size	2 MB
Average Latency	4.20 ms (nominal)
Contact Start/Stop Cycles	40,000 minimum
Seek Times	
Read Seek Time	8.9 ms
Write Seek Time	10.9 ms (average)
Track-To-Track Seek Time	2.0 ms (average)
Full Stroke Seek	21.0 ms (average)
Transfer Rates	
Buffer To Disk	400 Mbits/s (Max)

It is somewhat unrealistic to compare the two for purely random or purely sequential memory access patterns, because they do not correlate to real world memory access patterns. Fortunately, analysis on the CDFs still provides useful insight into fundamental application execution behaviors.

5.2.1 Disk Latencies

Random read latencies observed on disk are predictably slow. The average read latency on a random block is ∼8ms. This is close to the average seek time of the disk. This makes sense, since a random read requires the disk to reposition its head on every read, and nullifies the effort of the on-disk cache. Sequential reads produce surprising results. An average read operation is ∼2μs. This is because both the kernel and the disk work to prefetch data before it is requested. For 100,000 sequential reads the first request takes 31,744μs, but succeeding reads only take 1 μs - 2 μs due to prefetching.

On disk caching enables quick write times. When the on-disk cache is not full, disk writes average ∼10μs. For streams of write requests greater than 20,000 the disk performance degrades significantly. The average disk write time for this 100,000 write experiment is actually ∼2ms. This is because write latencies of ∼1s occur in situations where the disk cache is full. This is important since page swapping for memory intense applications performs several million writes, and may fill the disk cache. Sequential disk writes produce similar

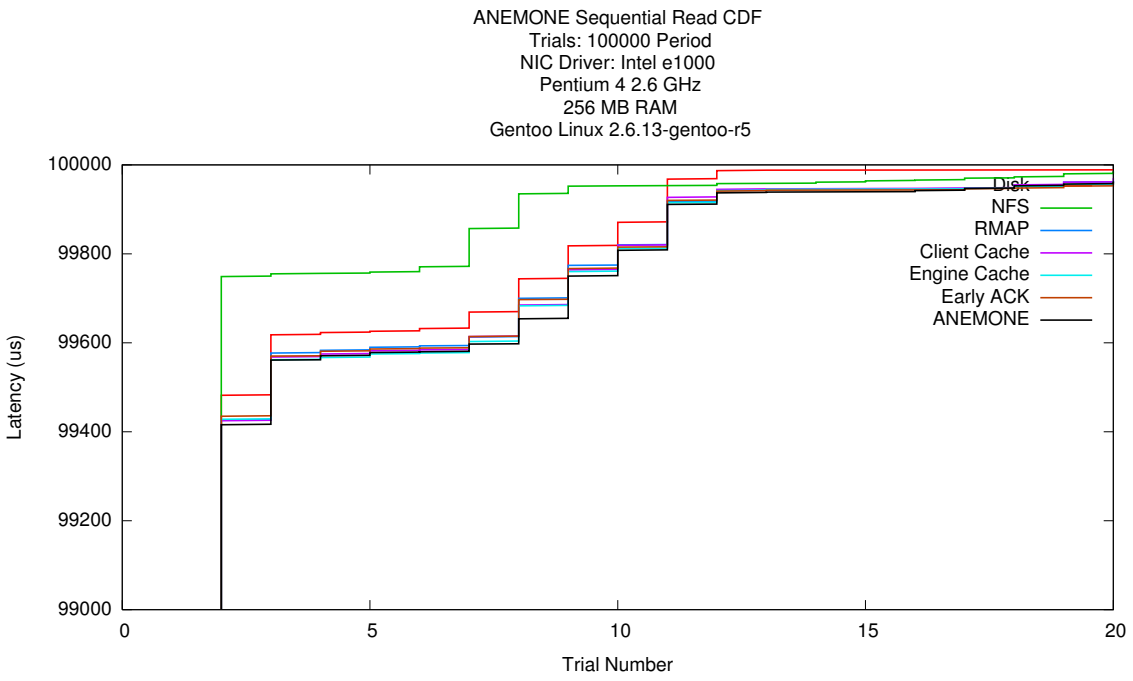

Figure 5.1: Sequential Read latency CDF for Anemone versus Disk

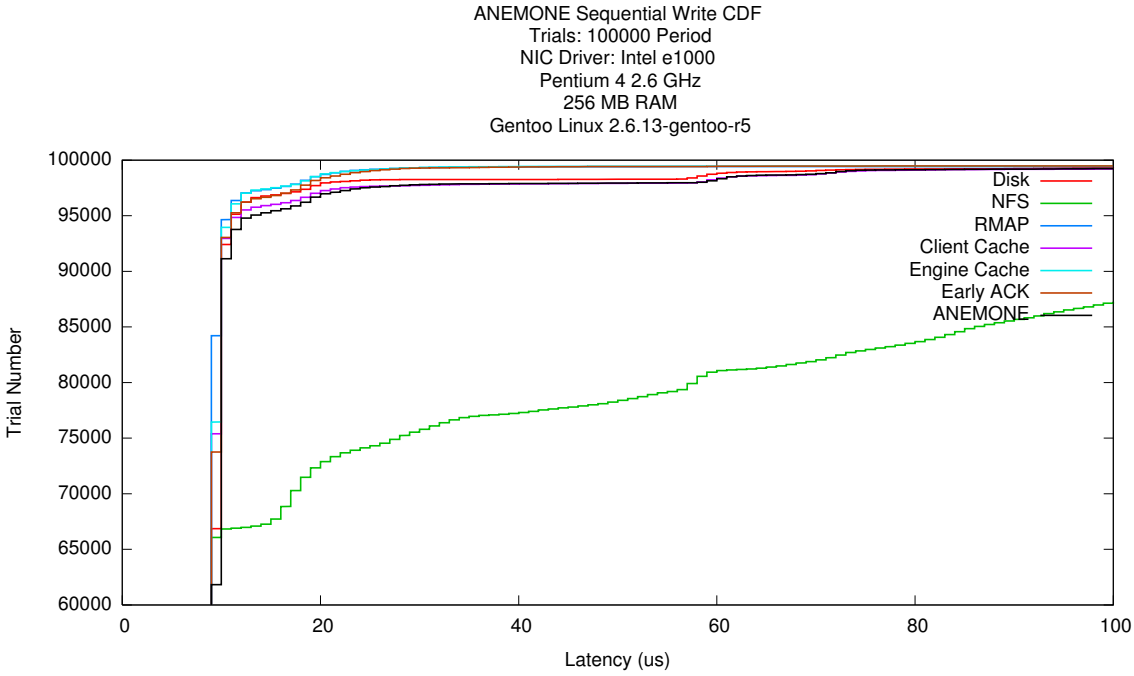

Figure 5.2: Sequential Write latency CDF for Anemone versus Disk

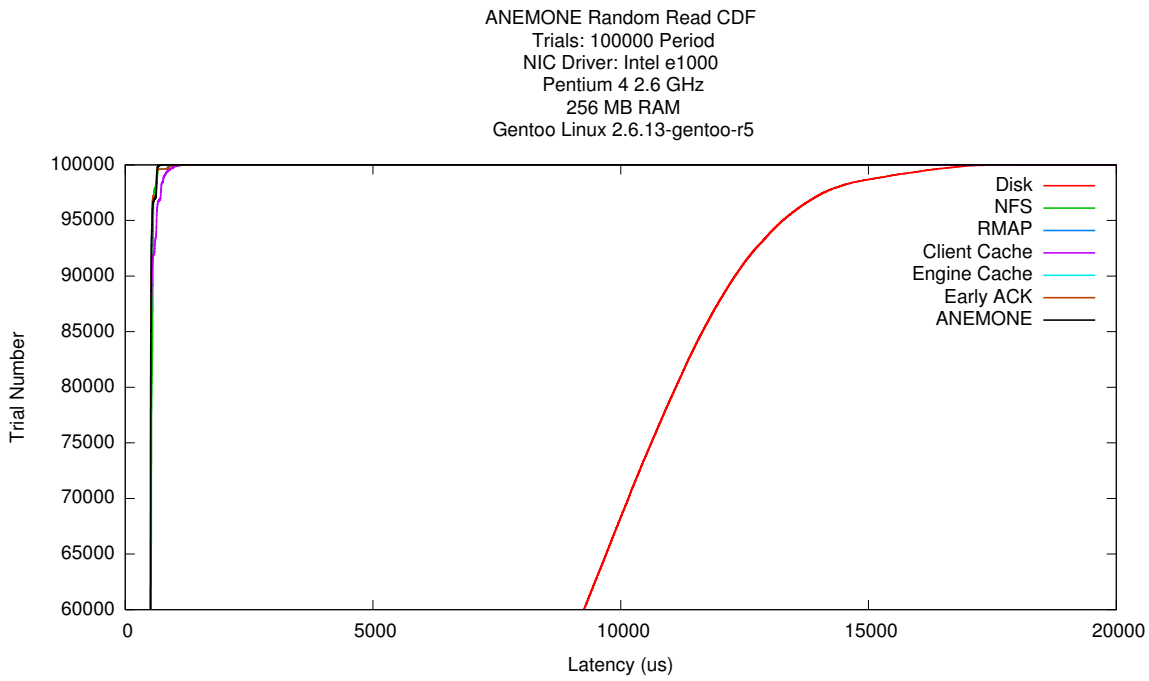

Figure 5.3: Random Read latency CDF for Anemone versus Disk

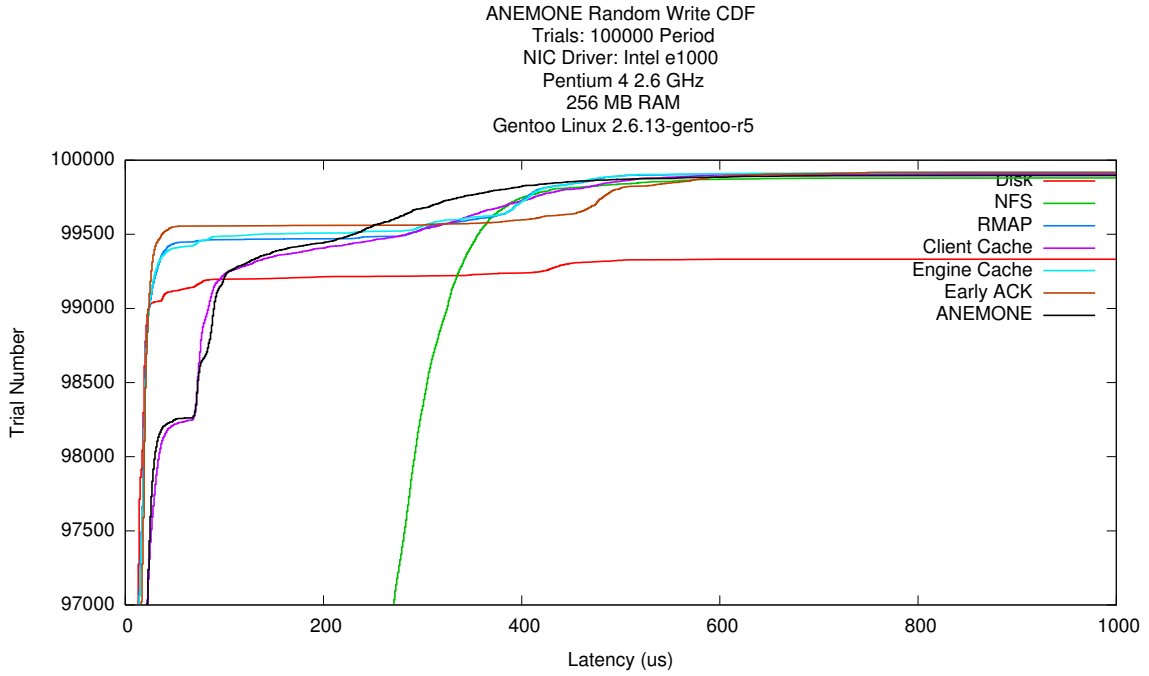

Figure 5.4: Random Write latency CDF for Anemone versus Disk

performance.

5.2.2 ANEMONE Latencies

ANEMONE is able to reduce disk paging latencies significantly. Figure 5.1 shows the cumulative ditribution of sequential read latencies. It is noticeable that disk and all ANEMONE implementations show similar latencies, with disk completing greater than 99% of its reads in less than 5μs. The original ANEMONE over NFS performs the best in this situation, with ~99.9% of all read requests completing in under 5μs.

Reducing sequential write latencies may be the largest factor of improvement for ANEMONE. Figure 5.2 shows that ~70% of sequential write requests are completed in under 20μs in the original NFS ANEMONE system. When RMAP is used to replace NFS all other implementations of ANEMONE produce similar latencies to that of disk based systems. This is due to RMAP's built in buffer space, which allows the ANEMONE client to complete a write request before it is sent out over the network.

Figure 5.3 shows latency reductions for random access reads. The figure shows that the disk performs very badly under this highly randomized situation. The disk is only able to complete ~7% of the requests in under 500μs, while all six implementations of ANEMONE can complete ~90%.

Finally the experiment testing random write latencies shows the greatest improvement for the new ANEMONE system. Figure 5.4 shows that all systems are able to complete ~99% of their write requests in under 300μs. The most important detail to note is that write latencies for the first 99% requests are comparable to disk (~10μs) on systems using RMAP. The NFS system shows write latencies ~300μs.

5.3 Single Process Performance Results

Single process experiments are meant to evaluate the case where there is high locality in memory access pattern and, as a consequence, the disk heads range of motion is limited when swapping to local disk. Single processes are also helped by swap daemon prefetching. When the swap daemon has more memory to work with, it performs more aggressive page prefetching.

This section compares disk performance to ANEMONE for systems running a single large-memory process. Application size varies for each trial. Trials begin with a 100 MB

Table 5.2: Read/Write Latency Test Results for 100,000 tries

	Sequential Read	Random Read	Sequential Write	Random Write
Disk	2 μs	8100 μs	10 μs*	80 μs
ANEMONE	1 μs	490 μs	87 μs	83 μs
RMAP	2 μs	486 μs	85 μs	84 μs
Client Cache	2 μs	510 μs	81 μs	79 μs
Engine Cache	2 μs	486 μs	87 μs	84 μs
Early ACK	2 μs	486 μs	87 μs	86 μs
ALL	2 μs	494 μs	82 μs	80 μs

application, and increment each succeeding application size by 100 MB. The final trial is a 1200 MB application.

5.3.1 Quicksort

Results for single process quicksort experiments are shown in figure 5.5. The gap between disk and the original ANEMONE system grows as application size increases. The gap between the original ANEMONE system and the augmented system also grows. At 1200 MB the original ANEMONE implementation increases performance by 130%. The augmented version improves on the disk's performance by 298%. This is a performance increase of 168% over the original ANEMONE system.

5.3.2 POV-Ray

Single process POV-Ray experiment results are shown in figure 5.6. Although POV-Ray has a more predictable memory access pattern, it performs more page swapping than quicksort. For the 1200 MB trial the original ANEMONE implementation is 150% faster than disk. After implementing its changes, the updated version of ANEMONE improves disk performance by 370%. The new ANEMONE system is 120% faster than the original implementation.

5.4 Multiple Process Performance Results

Multiple process experiments test a random-like memory access pattern. Although typical applications access memory in an almost sequential access pattern, running multiple large-

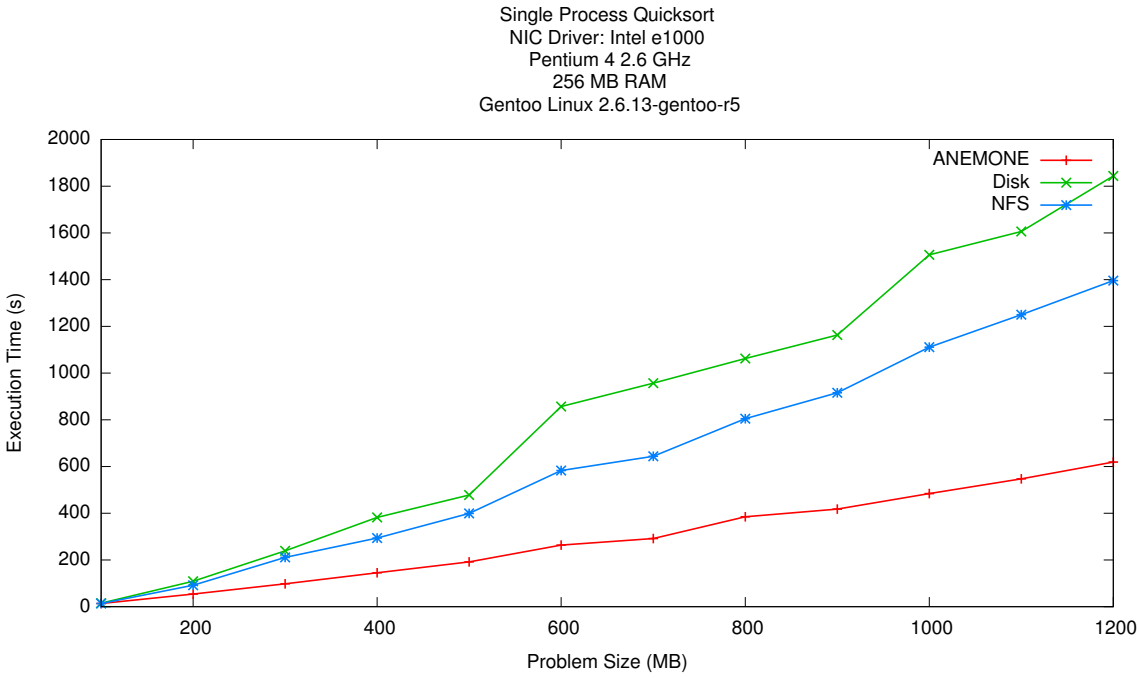

Figure 5.5: Single Process Quicksort: Anemone vs. Disk

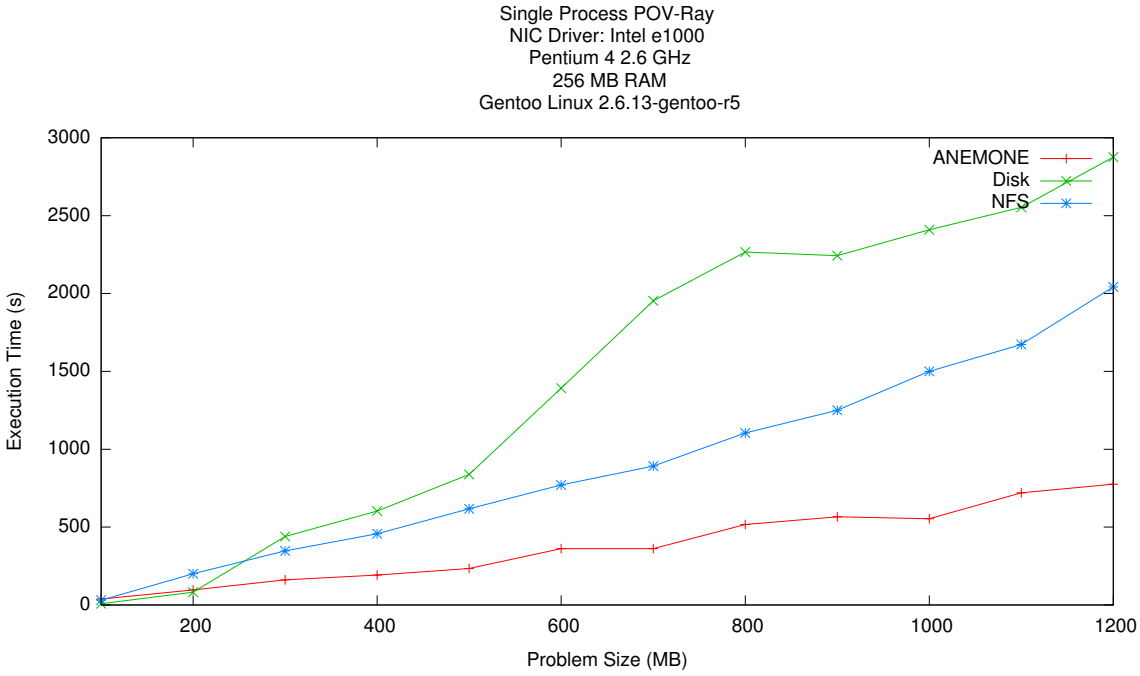

Figure 5.6: Single Process POV-Ray: Anemone vs. Disk

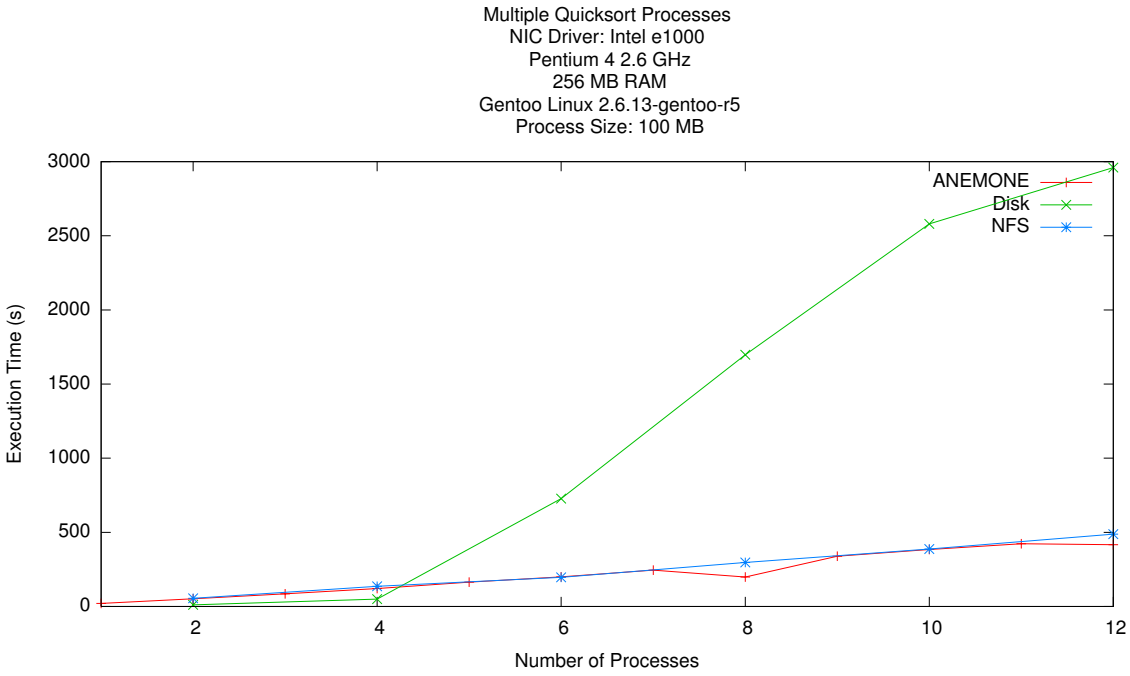

Figure 5.7: Multiple Process Quicksort: Anemone vs. Disk

memory applications forces the virtual memory system to swap pages in and out of different memory subspaces. A system accessing varying subspaces at the same time produces an access pattern that can be modeled as random.

In these experiments an increasing number of 100 MB processes are run against each other in each trial. The trials start with 1 process, and increase all the way up to 12 concurrent processes. The metrics for these experiments are: start time, end time, cache hits, and cache misses.

5.4.1 Quicksort

Multiple process quicksort results are shown in figure 5.7. Increasing the number of quicksort processes running in tandem shows a growing trend of improved ANEMONE performance. With 12 concurrent processes running on the test machine, the original ANEMONE implementation's performance is 600% faster than disk. The newer version is 710% faster than disk.

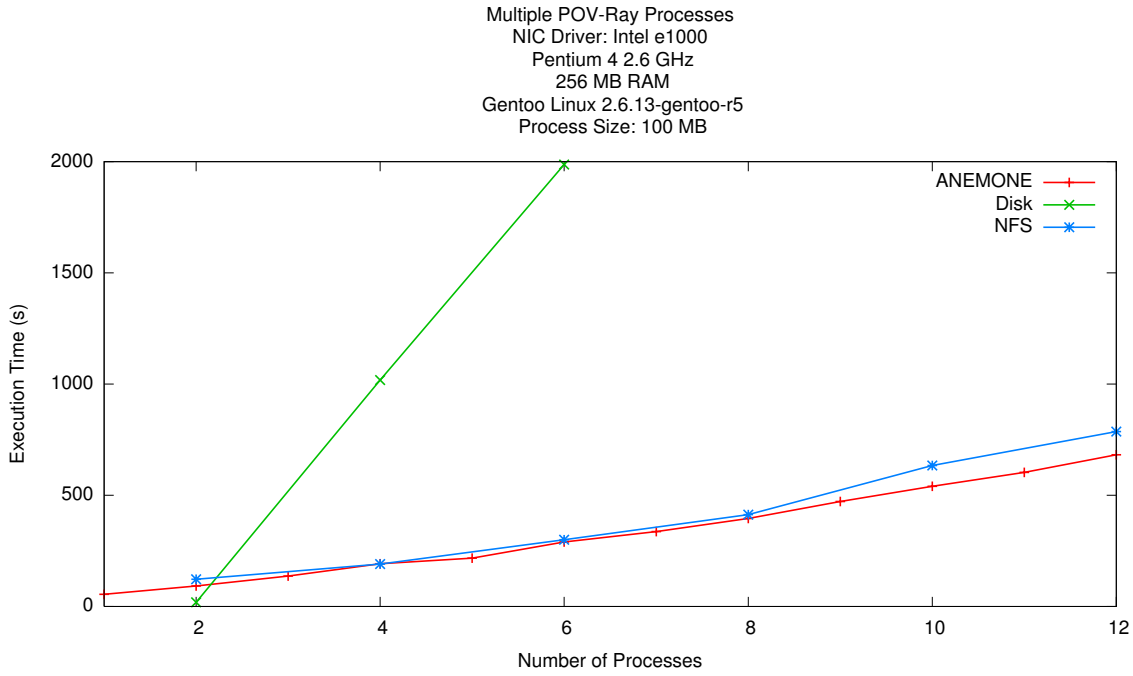

Figure 5.8: Multiple Process POV-Ray: Anemone vs. Disk

5.4.2 POV-Ray

Multiple process experiment results for POV-Ray are shown in figure 5.8. The performance improvements seen for POV-Ray shows promise. ANEMONE is faster than disk by 770%, while ANEMONE with its new features is 835% faster than disk.

5.5 Cache Performance Analysis

This section outlines the performance of the two levels of caching in the ANEMONE system. All cache performance data has been gathered during the execution of previously described application experiments.

5.5.1 Client Cache

Client cache performance can be seen in figure 5.9. Cache performance for both quicksort and POV-Ray begin between 15%-20% for problems of small size. These are the highest cache hit rates, since a larger fraction of the experiment's problem is able to fit in the cache. As application size increases quicksort performance drops to ∼4%, still increasing

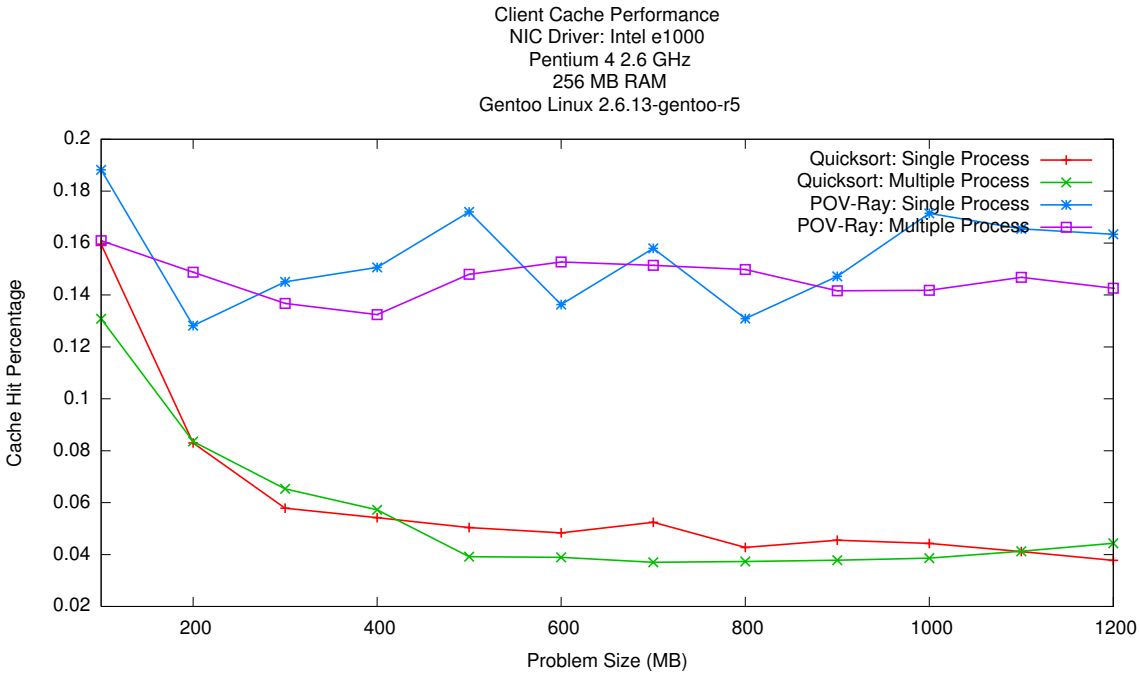

Figure 5.9: Client Cache Performance

performance by up to 45 seconds, while POV-Ray performance stays between 14%-18%, an overall performance increase of up to 270 seconds.

The increase in cache performance for the POV-Ray application is promising. It shows that a full production application can take advantage of a small, local cache in the ANEMONE client. Quicksort does not perform as well, because it does not have much temporal locality (Figure 3.4).

5.5.2 Engine Cache

The engine's cache hit rates can be seen in figure 5.10. The cache performs differently for both quicksort and POV-Ray applications due to their memory access patterns. For small problems, quicksort performs well, getting hit rates exceeding 75%. As the problem size increases for quicksort applications, cache hit rates approach 10%, then level out. Multiple POV-Ray applications generate a constant 10% cache hit rate, amounting to aproximately 1200 seconds in saved time.

It is probable that increased engine cache sizes will yield far greater cache hit rates. It

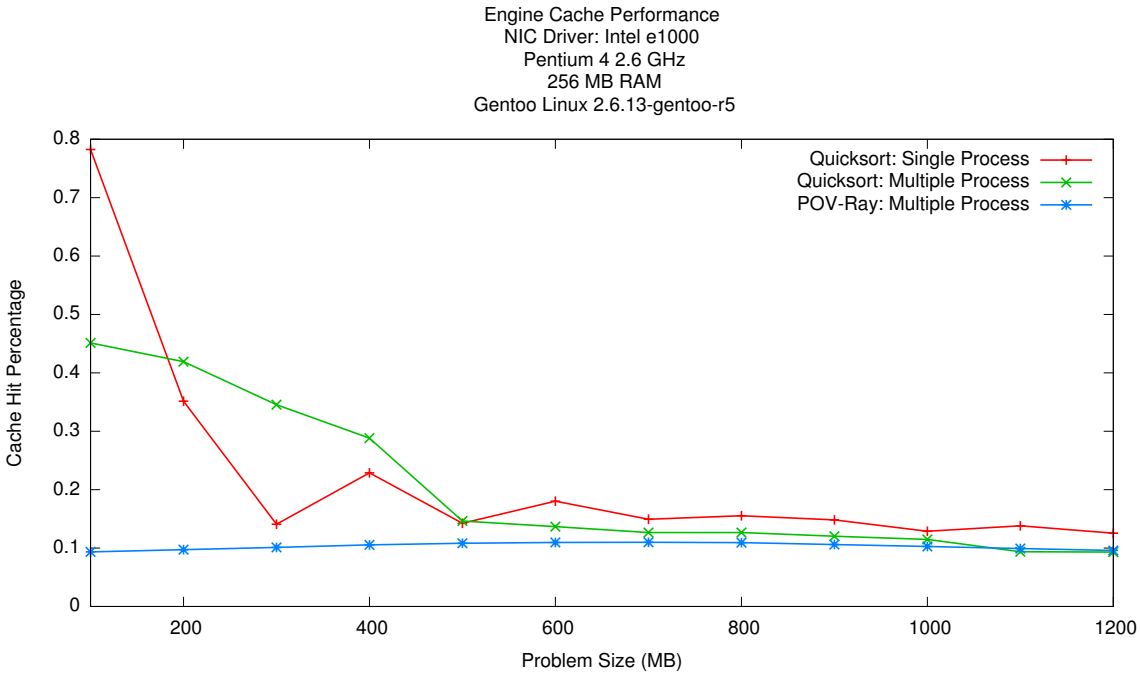

Figure 5.10: Engine Cache Performance

also shows that LRU caching works moderately well in caches under 100 MB in size.

5.6 Summary

Experimental results show that the improvements to the ANEMONE design yield a significant performance increase. An analysis of the per-block latencies in the ANEMONE system demonstrates that the proposed techniques have yielded a reduction in write latencies. This the largest factor in the application speedup.

Experiments for quicksort and POV-Ray show real world results for the enhancements. ANEMONE performance increases by as much as 18% over the original ANEMONE system, for single processes. The new implementation is approaching improvements on an order of magnitude for multiple process experiments.

Furthermore, an analysis on cache performance shows that caching can lead to a significant performance gain if cache hit rates can be improved. An implementation of more aggressive caching algorithms may increase hit rates.

CHAPTER 6

FUTURE WORK

There are several exciting avenues for continued research within the ANEMONE project. Future plans include evaluating the system under a more dynamic set of scenarios such as an increased number and variety of clients, severs, and engines, a larger test suite of memory-intensive applications including performance analysis on cache-oblivious algorithms [18], and real network implementation where background traffic may become an issue.

Cache performance improvements should garner a lot of attention in future development. Experimentation with different prefetching strategies has already suggested that improved cache performance is feasible. [1]Other implementations of caching strategies such as [19] may also increase cache hit rates. Since an increased memory footprint for the cache seems to directly correlate with decreased performance, page compression at the cache and/or server level is also an attractive option.

By compressing page data, ANEMONE can potentially realize two performance improvements: 1) reduction of network latencies, and 2) more efficient use of server space. Initial tests have shown that roughly 75% of all virtual memory pages can be compressed using simple Ziv-Lempel based compression algorithm. Compressing pages before they are transfered allows for smaller network propagation times, one of the largest contributors to current latency. Compressing pages only takes a few microseconds, but can save tens of microseconds per page in propagation.

Once a page has left a client neither the engine or the servers need to look into the data at any time. Current use of engine cache and server space is divided in fixed size chunks of memory because pages are of a fixed size. Shrinking some pages allows for more data

[1]Experimental disk-based versions of the ANEMONE client module have yielded cache hit rate improvements on the order of 10% - 50%.

dense memory storage. This allows ANEMONE to store more pages for the same set size of memory.

The current implementation of ANEMONE requires jumbo frame support throughout the network. Modifying the code to allow the system to run over standard 1500 byte ethernet frames should be done to allow greater flexibility. To enable this, pages would have to be broken up into pieces. Initially it might be best to allow split-paging only between the client and engine.

Developing Anemone into a peer-to-peer memory sharing cluster is also an appealing direction. Allowing users across a LAN, or perhaps even across a WAN to combine memory resources takes a large step towards providing a widely available, highly extensible, computing resource. This would present Anemone as a provocative platform for large dataset problem computing. A P2P network of publicly available memory could lead to the development of community owned, memory resident applications.

[20] shows that performing NFS transactions over RDMA has increased performance. Since ANEMONE transactions are NFS-like in their nature, it can be expected that ANEMONE would speed up over RDMA as well.

Work is currently underway to improve the scalability and fault tolerance of Anemone. Page replication would provide higher data availability. Multiple memory engines would help distribute large ANEMONE workloads. SPOF (Single Point of Failure) recovery must also be addressed.

When multiple servers are connected to a memory engine, the engine is must evenly distribute pages across them. Currently ANEMONE adopts a simple round robin strategy, sending pages to each server evenly. It is possible that during an application's execution, a core set of pages could get swapped frequently over ANEMONE. If these pages all reside on one or a few servers, these servers may see an increase in usage that is unfair to the rest of the ANEMONE network. To combat this, an adaptive load balancing strategy should be employed to monitor page usage across the system. If the engine observes an uneven workload it would have the ability to dynamically shift pages to less busy servers.

CHAPTER 7

CONCLUSIONS

This paper presents the design, implementation, and evaluation of Anemone – a transparent distributed memory virtualization system that can dramatically improve real world applications. ANEMONE is able to transparently swap pages over a gigabit network to a virtual pool of free remote memory. It has shown that it is possible to gather collective memory resources from a cluster of local machines, and provide a virtualized interface for client machines to interact with. ANEMONE improves application performance by factors of two to approaching an order of magnitude. The interface is transparent to the client machine, and allows access to a virtually unlimited union of remote memory. Anemone has demonstrated its feasibility without 1) modifying the client's operating system, or 2) modifying the memory-bound applications.

Implementation of a lightweight Remote Memory Access Protocol, two level caching scheme, and an Early Acknowledgment system have improved system throughput. Performance evaluations generated by two unmodified, real-world applications have yielded results indicating this. For single, memory-bound process experiments, the quicksort algorithm showed an improvement factor of 2.7 for a 1.2 GB array of random numbers. Meanwhile the POV-Ray ray tracing application improved by a factor of 3.7 for a problem of similar size. The differences in performance can be attributed to the application's virtual memory access patterns, and their resulting cache hit percentages.

As a client machine running ANEMONE is stressed by an increasingly heavy load, not an uncommon situation faced by many multi-user/multi-process systems, its difference in execution times versus those of competing disk-based systems balloon. Running twelve concurrent 100 MB quicksort processes demonstrates an execution improvement factor of 6.3. A similar POV-Ray experiment for twelve 100 MB scenes renders an improvement factor of

8.7. Further experiments on problem sets of larger sizes are expected to report performance increases surpassing an order of magnitude. ANEMONE's large jump in multiple process performance can be attributed to memory access patterns. When multiple processes page-fault within similar time frames, they produce a pattern similar to a constant stream of random page-in/page-out operations.

The Anemone cluster has proven to be a viable solution, competing with disk-based virtual memory systems. Affordable, high RPM, large cache disks show signs of significant effort to improve page swapping latencies. Their caches have been highly optimized to improve spatial locality prefetching. In contrast, the performance improvements outlined in this thesis show a noticeable gain for single application experiments, finally surpassing disk performance. ANEMONE's largest performance benefit is its ability to cut out lengthy disk seek times during multiple large-memory application execution.

Analysis on the Anemone system shows that as network speed, throughput, and latency continue to improve, a cluster based virtual memory pool provides a highly attractive solution to achieving a high level of performance for many applications.

REFERENCES

[1] M. Hines. Anemone: An adaptive network memory engine. Master's thesis, Florida State University, Tallahasse, FL, 2005. 1.1, 3.2

[2] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. *Operating Systems Review, Fifteenth ACM Symposium on Operating Systems Principles*, 29(5):201–212, 1995. 2

[3] D. Comer and J. Griffoen. A new design for distributed systems: the remote memory model. *Proceedings of the USENIX 1991 Summer Technical Conference*, pages 127–135, 1991. 2

[4] D. Engler, S. K. Gupta, and F. Kaashoek. AVM: Application-level virtual memory. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995. 2

[5] S. Koussih, A. Acharya, and S. Setia. Dodo: A user-level system for exploiting idle memory in workstation clusters. In *Proc. of the Eighth IEEE Intl. Symp. on High Performance Distributed Computing (HPDC-8)*, 1999. 2

[6] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996. 2

[7] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proc. of Intl. Parallel Processing Symposium, San Juan, Puerto Rico*, pages 153–159, April 1999. 2

[8] E.P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *USENIX Annual Technical Conference*, pages 177–190, 1996. 2

[9] M. Flouris and E.P. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Cluster Computing*, 2(4):281–293, 1999. 2

[10] E. Stark. SAMSON: A scalable active memory server on a network, Aug. 2003. 2

[11] T. Anderson, D. Culler, and D. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, 1995. 2

[12] F. Brasileiro, W. Cirne, E.B. Passos, and T.S. Stanchi. Using remote memory to stabilise data efficiently on an EXT2 linux file system. In *Proc. of the 20th Brazilian Symposium on Computer Networks*, May 2002. 2

[13] F.M. Cuenca-Acuna and T.D. Nguyen. Cooperative caching middleware for cluster-based servers. In *Proc. of 10th IEEE Intl. Symp. on High Performance Distributed Computing (HPDC-10)*, Aug 2001. 2

[14] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. Request for Comments - RFC 3530, April 2003. 3.1.1, 3.3.1

[15] Corbet J Rubini A. *Linux device drivers*. O'Reilly & Associates, Inc., 2nd edition, 2001. 4.5

[16] Silicon Graphics Inc. *STL Quicksort*. 5.1.1

[17] POV-Ray. The persistence of vision raytracer, 2005. 5.1.1

[18] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285. IEEE Computer Society, 1999. 6

[19] S. Bansal and D. Modha. Car: Clock with adaptive replacement, 2004. 6

[20] B. Callaghan. Nfs over rdma. In *Work in progress presented at 1st USENIX FAST Conference*, Monterey, CA, January 2002. 6

BIOGRAPHICAL SKETCH

Mark Lewandowski

Mark Lewandowski was born January 21, 1980 in Evanston, IL. At the age of 8 his family moved to the San Francisco Bay Area, where he lived until 1998 when he came to Florida State University as a college freshman.

Mark earned his BS in Computer Science on May 3, 2003, and quickly enrolled in the Master's program the following Fall semester. Mark plans on continuing school, and has enrolled in FSU's PhD program.