THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

ANEMONE: AN ADAPTIVE NETWORK MEMORY ENGINE

By

MICHAEL R. HINES

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, 2005

The members of the Committee approve the thesis of Michael R. Hines defended on April 7th, 2005.

Kartik Gopalan
Professor Directing Thesis

Zhenhai Duan
Committee Member

An-I Andy Wang
Committee Member

To my Dad, for giving me anything I ever needed growing up, even if he didn't agree with my ideas. His faith in me did not go unused…

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Kartik Gopalan for inviting me to come work with him and for his support in knowledge and financially as a research assistant. I've learned a lot and hope to go even further. I would also like to acknowledge Mark Lewandowski for his future efforts on implementing caching, Dr. Piyush Kumar's suggestion using quicksort and for ideas on future tests of cache oblivious algorithms and Hemant Banavar's suggestion of investigating the use of NBD as an option in Anemone. Finally, I would like to acknowledge Dr. Wang and Dr. Duan for being on my committee and for their important suggestions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Memory hungry applications consistently keep their memory requirement curves ahead of the growth of DRAM capacity in modern computer systems. Such applications quickly start paging to swap space on the local disk, which brings down their performance, an old and ongoing battle between the disk and RAM in the memory hierarchy. This thesis presents a practical low-cost solution to this important performance problem. We give the design, implementation and evaluation of Anemone - an Adaptive NEtwork MemOry engiNE. Anemone pools together the memory resources of many machines in a clustered network of computers. It then presents an interface to client machines in order to use the collective memory pool in a virtualized manner, providing potentially unlimited amounts of memory to memory-hungry high-performance applications.

Using real applications like the ns-2 simulator, the ray-tracing program POV-ray, and quicksort, disk-based page-fault latencies average 6.5 milliseconds whereas Anemone provides an average of latency of 700.2 microseconds, 9.2 times faster than using the disk. In contrast to the disk-based paging, our results indicate that Anemone reduces the execution time of single memory-bound processes by half. Additionally, Anemone reduces the execution times of multiple, concurrent memory-bound processes by a factor of 10 on the average. Another key advantage of Anemone is that this performance improvement is achieved with no modifications to the client's operating system nor the memory-bound applications due to the use of a novel NFS-based low-latency remote paging mechanism.

CHAPTER 1

INTRODUCTION

1.1 Motivation

In a machine, the bottleneck on processing is usually I/O (Input/Output) to the other devices in the system, one of the most slowest of which is the disk. The measured latency in servicing a disk request (including seek, rotational, and transfer delays) typically ranges from 4 to 11 ms on the Enhanced IDE S.M.A.R.T. II ATA/100 disks used in our testbed. In processing terms, this is an eternity and is the most expensive component in a program's execution time. Main memory and commodity machines themselves continue to get cheaper and accessing the remote memory of another machine is a very viable alternative.

Prior efforts to make use of unused memory of remote clients either required extensive changes to the client machines that wanted to use remote memory or were designed as combined address-space mechanisms intended for access to non-isolated memory objects. In this work, we design, implement and evaluate a high-performance, transparent, and virtualized means of aggregating the remote memory in a cluster, requiring no modifications to the client system nor the memory-bound applications running on them.

1.2 Resource Virtualization

Clusters of machines are very popular platforms for high-performance processing, including the execution of memory-hungry applications. Typically, while one machine in a cluster might be pressed for memory resources, most other machines in the same cluster might have plenty of unused memory. Hence, instead of paging to local disk, such memory-hungry clients could page over the high-speed LAN to the unused memory of other machines. Cluster-style machines as well as off-the-shelf networking hardware is often readily available and cheap.

Anemone strives to aggregate the collective resources of those machines for use by clients and then provide access to that combined remote memory through a method called virtualization.

Virtualization refers to the process of transparently dividing a pool of physical resources into a set of virtual resource shares among multiple users. One might consider dividing all the hard-disk space or processing capabilities in a cluster and offering those resources in a unified way to all the processes and users in the cluster, for instance. Just as the operating system virtualizes the different components in a system to it's users and processes, we want to virtualize remote memory in the cluster and provide it to the rest of the cluster's machines in a transparent manner.

Among other design issues described later, an underlying level-2 switched gigabit network helps support remote memory. Gigabit Ethernet technology speeds reach as high as 10-Gigabits per second and also come at very cheap prices, delivering very high bandwidth and very low-latency. For example, transmission of a 4k-packet takes as little as 31.25 microseconds using 1-gigabit NIC (1000 Mbit * 8 * 1024 KB/sec / 4KB).

1.3 Contribution

Anemone provides three novel contributions. First it is a distributed system that collects all the remote memory resources that any particular machine in the cluster is contributing and provides a unified virtual interface to access this low-latency and high-bandwidth memory space for any client machine. Anemone provides such an interface through the use of NFS (Network File System), allowing the client to transparently interact with the Anemone system without modifications to the client. Disk-based page-fault latencies range between 4 and 11 milliseconds whereas Anemone provides an average 700.2 microsecond, and often as low as 599 microseconds. In contrast to the disk, single memory-bound processes can expect a factor 2 improvement in execution time, and many concurrent, memory-bound processes can expect a standard improvement of factor 10.

Second, the client sees a highly scalable, almost unlimited, fault-tolerant pool of memory to which it can use as the needs of the cluster grow or the state of the cluster changes. This remote memory can be viewed as another level in the standard memory hierarchy of today's systems, sitting between the disk and RAM. Figure 1.1 shows a representation of the memory hierarchy before Anemone is used. Figure 1.2 shows the new representation.

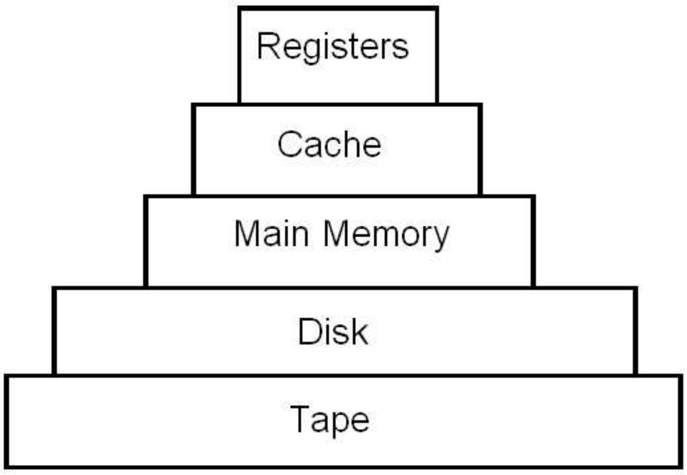

Figure 1.1. A representation of the memory hierarchy within a machine.

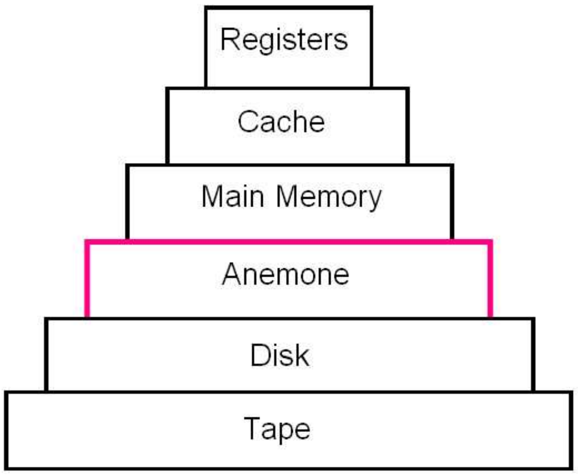

Figure 1.2. A modified memory hierarchy that contains a new layer, Anemone.

Third, Anemone provides foundations for virtualizing other distributed resource types such as storage and computation.

The rest of this thesis is organized as follows. Chapter 2 reviews the related research in the area of remote memory access. Chapter 3 explains the design of the system and Chapter 4 presents its implementation. Chapter 5 provides a performance evaluation of the system followed by a discussion on current and future work in Chapter 6.

CHAPTER 2

RELATED WORK

2.1 Early 90's

In the early 1990's the first two remote paging mechanisms were proposed [1, 2], both of which employed a customized remote paging protocol between clients and memory servers and incorporated extensive OS changes. In contrast, Anemone can operate with any client supporting the ability to swap to NFS-mounted files. Global Memory System (GMS) [3] provides network-wide memory management support for paging, memory mapped files and file caching. This system is also closely built into the end-host operating system. The Network of Workstations (NOW) project [4] performs cooperative caching via a global file cache, but does not provide a complete, network-distributed, transparent remote memory system like Anemone. [5] discusses remote paging which is controlled at the application programming level. Similarly, the Dodo project [6] provides a user-level library based interface that a programmer uses to coordinate all data transfer to and from a remote memory cache. Legacy applications must be modified to use this library, leading to a lack of application transparency. In contrast, Anemone provides a completely transparent and automated remote memory paging mechanism.

2.2 Late 90's

Work in [7] implements a remote memory system in the DEC OSF/1 operating system, but requires a customized device driver in the client-side kernel whereas our system can support any client using the standard NFS protocol. A remote paging mechanism [8] specific to the Nemesis [9] operating system permits application-specific remote memory access and paging. The Network RamDisk [10] offers remote paging with data replication and adaptive parity caching. Network RamDisk is implemented as a device driver inside the OS, again increasing the installation and management complexity at each node in the LAN. Other remote memory efforts include software distributed shared memory (DSM) systems [11]. DSM systems let programmers specify objects to be shared within the same network-distributed memory space (much like shared memory among different processes on a single machine), allowing distributed computing among many, many processors to scale. This is much different from the Anemone system which allows pre-compiled high-performance applications to run unmodified and use large amounts of remote memory provided by the cluster. Furthermore, Anemone only allows clients to swap to pre-sized remote memory regions pulled from the Anemone system - for each client in the system, these regions are all disjoint, and no two clients using the Anemone system ever share any part of their memory space with each other.

2.3 21st-Century

Remote memory based caching and replacement/replication strategies have been proposed in [12, 13], but these do not deal directly with clustered remote memory as Anemone does. [14] reports simulation studies of load sharing schemes that combine job migrations with the use of network RAM. They allow for process execution to be moved to different machines in the cluster in addition to the use of aggregated network memory. This is quite feasible but would again involve adding extra policy decisions and modifications to the kernel. Additionally, this approach does not support any client operating system capable of swapping over NFS. Samson [15] is a dedicated memory server that actively attempts to predict client page requirements and delivers the pages just-in-time to hide the paging latencies. The drivers and OS in both the memory server and clients are also extensively modified.

CHAPTER 3

DESIGN

In this chapter we discuss the architecture of Anemone. First we show a basic remote memory access setup in Section 1. We then extend that to Anemone in section 2. Finally, Section 3 discusses the design decisions involved in the system.

3.1 Simple Remote Memory Access

First we describe a very basic remote memory setup involving the Linux Ramdisk and NFS. A Ramdisk is a pure in-memory storage device. Like the hard disk, the device can be formatted to hold a certain filesystem type and then be accessed through the VFS (Virtual Filesystem) layer. In a simple remote memory setup, we have 2 machines: one which acts as a client low on memory and the other which acts as a server that hosts a Ramdisk. First the server puts a filesystem on the Ramdisk, such as ext2, and creates a large file on it that is big enough to act as swap space. The server then starts up an NFS server and exports the Ramdisk for mounting by the client. The client then mounts this export over NFS to a local mount point and configures the file inside it to contain a "swap filesystem" of sorts, whose format is understood by the client's swap daemon. The command to perform this on the file is "mkswap" in Linux, "mdconfig" in FreeBSD, and "mkfile" followed by "share" in Solaris. Finally, using the "swapon" (or just "swap" in Solaris) command, one can activate this file as swap-space and alternatively turn off the disk-based swap that is probably being used. Future attempts by the client's swap-daemon are re-directed over the network through NFS to be stored on the Ramdisk in the server. This static, one-client/one-server approach does not scale of course, nor does it achieve a minimum amount of latency as page-data must go through the socket and networking APIs of both machines, causing significant delay.

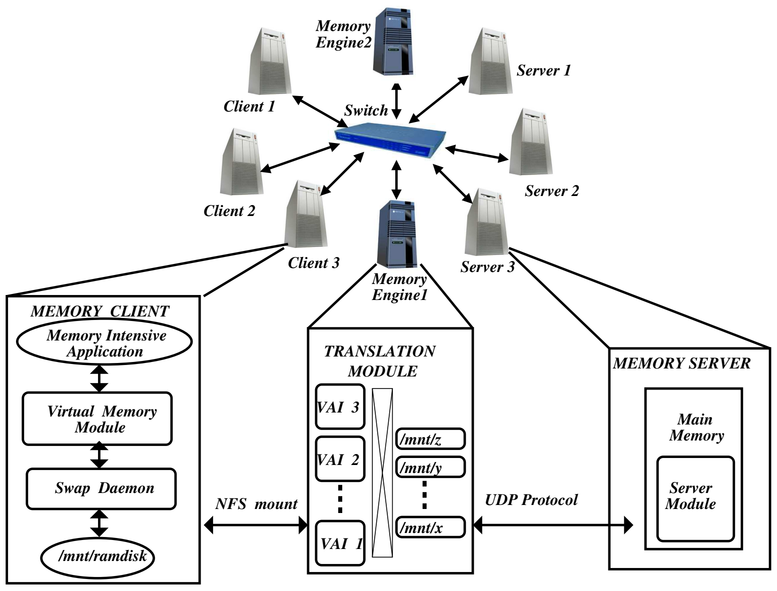

Figure 3.1. Architecture of Anemone: Low-memory Clients on the left send their requests over NFS to the Memory Engine in the center, which in turn get multiplexed on the Memory Servers on the right.

3.2 Anemone Architecture

In Anemone, the client's function stays the same. The server holding the Ramdisk is replaced by what we call a "memory engine". The engine is a Linux machine that serves two purposes: 1. It acts as a pseudo NFS server and accepts NFS requests. 2. It takes these requests and re-transmits them to be stored on a "memory server", another Linux machine offering remote memory space. Figure 3.1 shows a diagram of Anemone.

3.2.1 The Memory Engine

Again, from the point of view of the client, the memory engine is an NFS server. This NFS server is a fully-functional NFS version 3 server that implements only a subset of the necessary requests of both the NFS and MOUNT protocols. Internally, the NFS server operates completely in memory. When a client mounts a file from the engine, the engine allocates what we call a Virtual Access Interface, or VAI. The VAI is Anemone's version of a real file that would normally exist on the filesystem of a disk. The VAI has all the metadata associated with it that a real file on disk would normally have, including size, modification time, etc. This metadata is kept in memory and a file handle corresponding to the VAI is returned to the client to be used throughout the lifetime of the client's interaction with that particular VAI.

3.2.2 The Memory Server

The memory server stores data in its RAM. It offers fixed amounts of its memory to the Anemone system and may or may not be a general purpose machine. A memory server connects to the engine using a lightweight, UDP-based, reliable, stop-and-wait style protocol. Upon connecting, the server informs the engine of the amount of bytes of memory it is offering to the entire Anemone memory space and that it is ready for storage.

3.2.3 Putting it all together

The engine may serve any number of clients in the cluster. At the same time it may also allow any number of servers to connect to it and offer memory space to the Anemone system. When a client has mounted a VAI, configured a swap-file, and activated it, one of two things can happen: a page-out (corresponding to an NFS Write), or a page-in (corresponding to an NFS Read) when page-faults occur during runtime. On a write, the request and it's page-data is sent to the engine and the page is scheduled to be stored on any of the connected memory servers. Any page, including pages from the same source, can possibly be scheduled on any of the connected memory servers, allowing for extremely flexible scheduling algorithms of different types. Then the page is transmitted again to a particular server and stored in memory. On a read, the request is sent to the engine and the engine internally looks up the location of the server at which the page was previously stored. The engine then issues a request to the server to retrieve the page. When the page is retrieved from the server the engine adds an NFS reply header to it and transmits the page to the client(s), fulfilling the page-fault. There are no disks involved in the entire process and the engine and server(s) both operate completely in memory.

3.3 Design Decisions

3.3.1 Virtualization, Centralization, and Scalability

The engine is the central core of Anemone. It multiplexes the memory resources that each server contributes to the system. The use of NFS allows us to employ the technique of Resource Virtualization, the ability to sub-divide a system's physical resources into shares of multiple virtual resources. Since NFS is a standard part of today's operating systems, Anemone can be used transparently to virtualize the memory system and create another level of the memory hierarchy that sits between RAM and the hard-disk. This also has the side-effect of allowing Anemone to do fine-tuned remote memory management, including the placement of pages on memory servers, load-distribution, and scaling. At anytime, clients can use Anemone in a "plug-and-play" manner by NFS-mounting/swapping to VAIs when needed and un-mounting/shutting down when they are finished. Similarly, we have designed Anemone to allow use of memory servers in the following manner: if a memory server wants to reclaim its memory space, it can disconnect from the engine and allow the engine to move the pages contained on it elsewhere, making the system highly adaptable to change. Anemone can then scale by growing and shrinking the number of memory servers and the size of it's aggregated memory space dynamically. This indeed presents the engine as single point of failure. As a solution, future work can employ the use of multiple memory engines, as shown in Figure 3.1. More on this is described in Chapter 6 on Future Work.

3.3.2 Integrated Layer Processing

The translation module, shown in Figure 3.1, handles the engine's NFS and page-scheduling operations. It must be fast so that page-fault latencies are short. Since the engine is a dedicated machine that should only do the specific job of handling memory requests, we decided that the transport layer was too expensive and should be avoided - it incurs an extra processing layer in the networking stack and Anemone should usually be the only UDP-based thread of execution running on the engine. Similarly, Anemone operates within a cluster of machines and there is no need for routing to external networks for packets leaving the engine. Thus Anemone can avoid the extensive IP layer processing as well.

What about the other IP and transport-level abilities? Within the cluster, reliability is sufficiently handled by the retransmission and flow control functions that the NFS specifications require from implementations of NFS clients. Between the engine and the server, Anemone employs a customized light-weight stop-and-wait protocol for reliability. The need for IP fragmentation is eliminated because gigabit networking technology provides support for the use of "Jumbo" frames, allowing for MTU (Maximum Transmission Unit) sizes larger than 1500 bytes and as high as 8 to sometimes 16 kilobytes. Thus, an entire 4 KB or 8 KB memory page, including headers, can fit inside a single packet.

This technique of bypassing the different layers is called Integrated Layer Processing (ILP) [16]. ILP is a design technique used when only part (or none) of the full networking stack is needed. Anemone "integrates" the processing of the NFS, RPC (Remote Procedure Call, upon which NFS operates), UDP, and IP headers into one efficient layer by placing the translation module and memory server module directly into the OS kernel. Client machines do in fact use the entire networking stack, but once the Anemone system has control of the request ILP is used exclusively. Thus, Anemone uses a somewhat modified ILP strategy - allowing the clients to act normally and the engine/server machines to avoid the extra data copy and context switching overheads associated with a user-space level translation module. To accomplish this, the translation module operates in the interrupt context of the device driver for the network card. Immediately upon reception of the packet, the translation module interprets all headers, makes a decision, does a transmission if necessary and returns control. Future network interrupts repeat the cycle throughout the lifetime of the system.

3.3.3 The Network Block Device

We also explored Linux's Network Block Device (NBD) as an option instead of NFS in the Anemone system. NBD is a pseudo block device that accepts I/O from one machine but actually stores the data on a remote machine running an NBD server. The NBD server on the remote machine can accept the I/O and store the data in any way it sees fit, such as on a real device like the disk or a file, or in memory using the Ramdisk described earlier. Just as NFS is used as a transparent front-end for clients to use, NBD would act the same way and the engine would perform the same header and scheduling routines that it does with NFS, storing pages on memory servers. Because of Anemone's use of ILP, NFS allows for faster page-fault latencies than NBD does. This is because NBD operates over TCP. Unlike UDP, TCP maintains too much state for Anemone to keep track of on its own in a reasonable amount of time nor is that state needed, as described earlier. In our own tests, we created the following simple scenario: Using two machines, one acts as a TCP-based echo server and one as an echo client. This is done using kernel modules where each module creates an in-kernel socket. The time taken for one machine to issue a 4 kilobyte echo-request and to receive a reply from the second machine over TCP takes 480 microseconds. This is how much latency would be involved in the first half of a page-fault, i.e. the communication time involved in first transmitting the page to the engine and the time involved in transmitting the NBD reply back to the client. This includes TCP processing, two network transmissions, and copying overhead. The time involved in communication with the memory server (with the engine) always remains constant, as that part of Anemone is of no concern to the client. Alternatively, the same NFS based latency involved in communication between the client and the engine during a page-fault in Anemone only takes 352 microseconds on average. This includes two transmissions, no copying and UDP-based communication using NFS. This is a large difference in such a latency-critical environment. Thus, we decided to use NFS.

3.4 Summary

We first described the basic remote memory paging architecture that can be statically setup between two machines. To improve on that Anemone does the dual job of connecting to any number of server machines in the cluster, becoming aware of the memory resources that they are offering and presenting that through a virtualized interface to any number of client machines by acting as an NFS server. Anemone operates over a switched gigabit network and can support any client machine whose operating system can swap to a file located on a Network File System. Additionally, with the use of ILP, Anemone centrally accepts page-requests from a client and schedules them to be stored on a particular memory server. Future page-faults directed to Anemone cause that page to be retrieved from the memory server and sent back to the client.

CHAPTER 4

IMPLEMENTATION

The Anemone system was first designed as a prototype in user-space to determine the feasibility of the system. In the user-space implementation, both the engine's translation module and the server's storage module were written as C++ programs communicating over the Linux socket API, including all the kernel-to-user copying and context switching overhead. Afterward, both the engine and server implementations were ported to kernel-space by way of Linux loadable kernel modules. First, we present the fundamental parts of the implementation and and hardware used. Then we proceed to go into the specific user and kernel space implementation details.

4.1 Technology

4.1.1 Operating Systems

Our prototype system involves 3 machines: one client, one engine, and one server. The engine and the server must run either Linux 2.4 or 2.6, whereas the client can run any operating system supporting the ability to swap to a file located on an NFS mount, including BSD-variants like FreeBSD or even Solaris. (It may even be possible to use Windows, as they do have their own NFS client implementation in the popular package "Windows Services for Unix" that appears to follow the NFS standard well, but this has not been tested.) Surprisingly, a plain vanilla Linux kernel does not support swapping over NFS. Our performance testing was done using Linux by using a kernel patch provided in [17]. This patch only supports Linux 2.4, therefore clients that wish to run Linux must use version 2.4. We may update the patch to work with 2.6 in the future for users who need Linux clients.

4.1.2 Hardware

The client, server, and engine machines all run identical 2.6 Ghz Pentium processors. The client machine has 256 MB of memory, the server has 1.25 GB of memory and the engine machine has 650 MB of memory. As stated earlier, the system runs on a gigabit network. Each of the three machines use Intel Pro/1000 MT Desktop Adapter cards. The switch, upon which all 3 machines lay, is an 8-port SMC Networks gigabit switch supporting Jumbo Frames. The machines and network card settings have been precisely tuned to operate at gigabit speeds and we have observed a peak UDP-packet stream between any two machines of 976 Mbits/second, using interrupt coalescing and jumbo frames. Interrupt coalescing is a network card feature that allows the card to deliver interrupts for packets to the kernel only after a specified interval of time instead of immediately upon arrival of the packet. At gigabit speeds, the extreme amount of interrupts can severely hinder the performance of the CPU. However, even without interrupt coalescing, we get speeds just above 800 Mbits/second. If we were to use interrupt coalescing, the delay imposed on a page-fault would be lower-bounded by the interval of time specified to the network card between interrupts, which is not desirable. Thus, Anemone operates without this hardware feature.

4.2 Engine Design

4.2.1 Mapping Pages to Servers

The memory engine's most basic function is to map client pages to the server that the page is stored on. We use a hashtable to do this. The hashtable is keyed by the identity of a particular client page and the data attached to that key indicates the particular server that holds that page. A key (a page's identity) is composed of the triplet:

{Client IP address, Offset, VAI id}

The VAI id within the triplet corresponds to a particular VAI or Virtual Access Interface, the engine's name for an exported directory available for mounting by a client in NFS. Inside this VAI appears a file from the client's point of view. Recall that the engine may offer many such VAIs. A VAI may only be used by at most one client, whereas a particular client may use (NFS mount) multiple VAIs. The VAI is not a real file, but simply an in-memory structure containing file-like metadata. Requests to this VAI are scheduled to any particular memory server that the engine sees fit. Thus, to identify the page, the VAI that the page came in on must be identified as well, in addition to the source address of the client.

The offset contained within the triplet is an NFS file offset. When the client activates and issues NFS READ and WRITE requests to the virtualized file contained within the VAI, those requests include the offset within that file that is being addressed. As the swap daemon fulfills page-faults or kicks out dirty virtual memory pages to the VAI, the pages get stored or retrieved from different offsets within the file. To complete the identification of a page, that offset is added to the triplet.

Finally, when indexing into the hashtable, a hash-function passes over each byte within this triplet, creating a hash-value that is unique enough to maximize usage of all the buckets within the hashtable and prevent collisions. At the moment, our system prototype only uses one memory server and one client for performance testing, but this basic framework will allow us to easily implement scheduling algorithms as well as tailored hash-functions that minimize insertion time and lookup time for the hashtable and bucket usage while decreasing collisions. Thus, we only use a simple modulus based hash-function until the rest of the system is developed later. Pages that get kicked out for the first time are entered into the hashtable. Future page-faults to the same page cause the engine to re-construct a key, look into the hashtable, get the page's location, contact the server, retrieve the page, and complete the NFS/memory request.

4.2.2 NFS Implementation

The current version of the Anemone engine follows the NFS version 3 RFC specifications [18]. The engine implements only a subset of the operations in the specifications needed by NFS clients that do swapping. These requests include: READ, WRITE, ACCESS (file permissions), GETATTR & SETATTR (file attributes), FSTAT & FSINFO (filesystem statistical information), LOOKUP (find inodes for files or directories), and the NULL operation. Operations like CREATE and REMOVE (a file or directory) are irrelevant since a client cannot modify the existence of a VAI - they are predetermined upon configuration of the Anemone system. Similar reasons apply towards the other available NFS operations.

4.2.3 Pre-fetching

Many different levels of the virtual memory process do pre-fetching of data, (also called "read-ahead"), where an additional number of blocks of data are fetched during an I/O request in anticipation that the next few blocks will also be read soon. The VFS (Virtual File System), the swap daemon, and NFS all do their own types of pre-fetching. As a result, NFS requests to the engine tend to have lots of requests grouped together which are spatially local to each other. To handle this (as any NFS server probably does), the engine creates a request queue for each client that connects to the engine. This request queue is a FIFO queue. Requests go in the front, and when interaction with a particular memory server is completed, they get popped off the back and an NFS reply is sent back to the client to complete the request. The queue is of a fixed size and will silently drop packets when the queue gets full. NFS clients respond to this flow control problem by setting a timer to detect the dropped request and retrying later.

4.2.4 Engine to Server Communication

The engine and server speak to each other using a lightweight, UDP-based, stop-and-wait messaging protocol. A message consists of the type of operation (Page-IN or Page-OUT), the pre-constructed key that identifies the page, and the page itself. Upon the server's connection to the engine, the server continuously blocks for page requests from the engine. A side-effect of NFS client flow control is that flow control between the engine and the server is also taken care of. The protocol needs only handle retransmissions. This is done by timing out requests contained within the aforementioned request queue. After a maximum amount of retransmissions to the memory server to store or retrieve a page, the request is dropped and the client must retransmit the request from scratch.

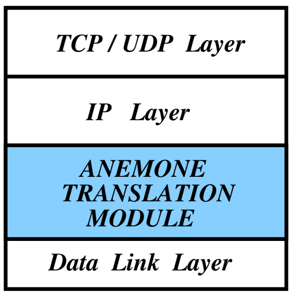

Figure 4.1. Placement of Anemone in the networking stack: Packets destined for Anemone get handled immediately after reception from the network card and never reach the network layer. All other packets continue up the stack.

4.3 Server Design

Just as the engine uses a hashtable, keyed by the identity of a page, the server does as well. The difference is that the data attached to a unique key in the engine is the location of a page, whereas the data attached to that same key in the server is the page itself. This is why the key is transmitted to the server through the messaging protocol on all requests. Additionally, the server does not maintain a request queue like the engine does. Future work that implements a system using multiple memory engines may require the servers to have a queue. This is the extent of the server's design - it is very simple and operates only in memory.

4.4 Kernel-Space Specifics

Implementation of a user-space and kernel-space version of the engine and server are naturally quite different. The most notable differences are how packet reception is done using the ILP approach and the change in memory management in the Linux kernel. The books in [19] and [20] provide a lot of the nitty-gritty details described here that are used in Anemone's kernel code.

4.4.1 Packet Reception

NFS requests in the engine's user-space implementation are handled with the select() system call in the socket API, allowing the program to distinguish between client traffic and server traffic. In the kernel however, both the engine and server implementations sit directly on top of the network card - between the link layer and network layer, as shown in Figure 4.1. In Linux, when an the network card sends the OS an interrupt after the arrival of a packet, a second software-based interrupt called a SOFTIRQ (or bottom-half) is scheduled to run a short time later. This softirq, which runs in "interrupt mode", is what actually takes the packet and hands it to the network layer for IP processing. What the engine does is insert a hook into this part of the kernel's networking code belonging to the softirq. This hook does a quick check to determine whether or not the new packet's UDP header indicates that the packet is destined for NFS (i.e. Anemone), based on the port. If so, the packet is removed from the networking layer's input packet queue and delivered to Anemone instead. At this point we are still in interrupt mode. From here, Anemone interprets the request, schedules communication with the server and returns control to the kernel all in one shot. On the server-side, the exact same procedure happens upon reception of a message to store or retrieve a page.

4.4.2 Memory Management

In order for the user-space implementations of the engine and the server to keep their stack and heap data in memory, we execute the "mlockall" system call. This alerts the kernel to prevent all pages associated with the running process that made the call from being swapped out. The kernel-space implementation does not have this problem since all memory management is done atomically.

However, memory management itself is quite another issue. The hash-tables used in the engine and server consume very little memory. The number of buckets in the hashtable remains static upon startup and is allocated using the get_free_pages() call. Linked-lists contained within each bucket hold 64-byte entry structures that are managed using the Linux slab allocator. The slab allocator does fine-grained management of small, same-sized objects of memory by packing them into pages of memory side by side until a page is full.

When full, a new page is allocated by the slab allocator and the process repeats as more entries are added to the table. kmalloc() is never used as it is not necessary and causes fragmentation. All memory is allocated with the GFP_ATOMIC flag, a requirement of code that operates in interrupt mode. The slab allocator is also very appropriate for the other memory needs for the various structures and objects used in the kernel module.

4.4.3 High-Memory Issues

Linux, which is the only operating system that the prototype implementation of Anemone's engine and server is currently written for, has a specific way of addressing memory. Linux partitions memory into 3 zones: DMA (Direct Memory Access), Normal, and High Memory. The first 16 MB of memory is reserved for DMA, used for direct device-to-memory transfers. After that, the Normal zone occupies memory between the range of 16 MB and 896 MB. The kernel and all memory usage starts here. Everything after the 896 MB range is called High Memory. As of now, the kernel PT (page table) is not designed to address pages in High Memory - it does not automatically reserve PTEs (page table entries) for those high addresses. To use high memory, pages must be allocated with alloc_pages() and then mapped into the PT manually with certain PT mapping functions. The kernel handles this on its own when scheduling processes that use large amounts of memory in that zone. It does this by creating mappings in the PT that are temporary and short-lived so that future processes can re-use those PTEs later. For kernel programmers that need to statically and permanently allocate high-memory pages, this must be done manually. At the time of this writing, we have not properly determined how to programmatically do this in the kernel, which limits the amount of memory that our prototype server can offer in Anemone.

As a result, our memory server will only store about 400 MB of data. This happens as a result of a usage of about 250 megs for the kernel and a base number of processes, another 100 MB that we presume the kernel plays around inside of when doing scheduling, processing, and context switching, and another 150 MB use for the Anemone memory server's hashtable storage and other internal metadata. This totals to about 900 MB of Normal-zone memory without using high-memory. Fortunately, this does not adversely affect our ability to properly measure the latency of a page-fault, but it does limit the amount of memory used by our performance tests. We are currently investigating possible solutions to this problem.

4.5 Summary

Client machines may run any operating system supporting swap over NFS, and machines running Anemone must be Linux machines, all of which support gigabit connectivity. Clients then mount a Virtual Access Interface (VAI) over NFS containing an interface to Anemone and create swapspace out of it. Upon activating that swapspace, pages get sent to the engine, identified with a key, inserted into a hashtable, and scheduled on a memory server. Upon page-faults, clients again contact the engine through the VAI, the engine looks up the requested page's location, retrieves the page from the server, and returns it to the client with an NFS reply. Anemone processes packets in kernel space through Integrated Layer Processing by interpreting all the necessary headers at once without actually going through the TCP/IP stack, while avoiding IP fragmentation needs through the use of Ethernet jumbo frames and using UDP instead of TCP. The hashtable structure used for mapping/scheduling process consumes very little memory and the entire (kernel-based) prototype operates during interrupt mode for every client request. Thus, having large amounts of hashtable storage for many client pages and low interrupt-time computation allows even a single engine to scale well for many clients and many page-requests.

CHAPTER 5

PERFORMANCE

5.1 Setup

We implemented our prototype of the engine and the server first as user-space programs and then again as kernel modules. First we describe the latencies obtained for individual page-faults using Anemone versus using the disk in both user-space and kernel space. Then we show the kernel-based performance improvement gained from Anemone with 2 different kinds of application scenarios: 1. A single memory-bound process, and 2. Multiple, simultaneously running copies of the same memory-bound process, causing competition for the same limited RAM space.

Due to the high-memory limitation described in Section 4.4.3, all of the following tests were performed on the client machine containing 256 MB of main memory and 350 (less than 400) MB of swap space. The same amount of swapspace is used in both scenarios where a test is run against disk-based swap as well as Anemone-based swap. On the average, the kernel as well as other base processes required to run the client's operating system used about constant 60% of of the 256 MB of main memory. We observed that remaining 100 to 110 MB combined with the 350 MB of swap space is used for execution of our tests. Regardless of the high memory problem, Anemone performance results still show significant improvement over disk-based paging. The 350 MB of memory is equivalent to about 90,000 4k pages. The internal hash-tables in the engine and server are configured to have 131072 (2^{17}) buckets. Since we're only dealing with one client at the moment, our hash-function simply performs a modulus of the NFS offset of the page and the number of buckets in the hashtable. Thus, every page gets its own bucket and no collisions happen during page-to-server mappings. This hash function will be enhanced in future versions to handle multiple clients.

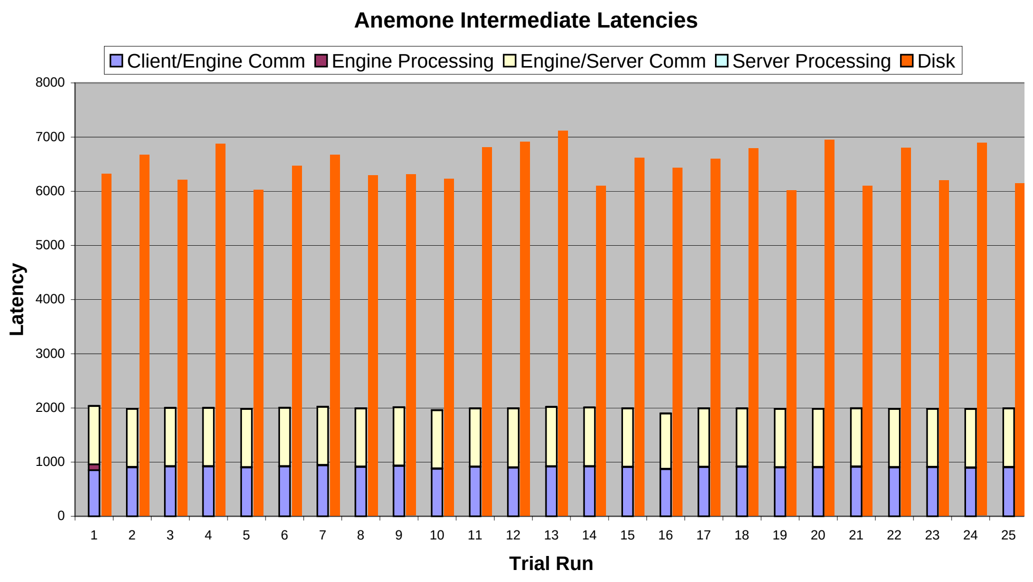

Figure 5.1. User-space Intermediate Latencies: The time it takes a client to request a block and the amount of latency involved in each step in Anemone.

5.2 Per-Block Latencies

5.2.1 User-space

When benchmarking Anemone, it is important to know what is going on at each component of the Anemone system. In our user-space implementation, a request for a page involves 4 steps:

1. An NFS request is transmitted to the engine
2. The engine requests the block from a particular server
3. The block is returned to the engine
4. The block is returned back to the client over NFS

This includes 4 network transmissions. It turns out that the actual computation done at the engine and server is negligibly small: within a few microseconds. The majority of the latency involved is spent in communication time when traversing the TCP/IP stack, traversal of the user to kernel boundary when data is copied from user-space buffers, and in context switching. Figure 5.1 shows the amount of time taken at each of these 4 stages.

The experiment was done in 25 trials. For each trial, we took the average of 100 randomized, non-sequential requests for a block within a 400 Megabyte space of data, composed of 4 Kilobyte blocks. Then we did the same test for 25 trials using Anemone. The graph pairs off trials from both the disk and Anemone, but the individual results in each half of a pair are in no way related to each other and are simply set side-by-side for observational convenience. However, the random numbers generated for each trial for testing are always the same between the disk and Anemone, which is done by using the same random seed with random number generator.

The purple bars at the bottom represent the client to engine communication time for the NFS request, calculated by subtracting the Anemone measurements from the total roundtrip time. This latency includes the time spent in sending the block over NFS from user-space through a socket interface as well as the reverse sequence of events for the process-based engine that received the request. This latency includes the NFS reply as well that occurs using the aforementioned sequence of events involved in completing the request. This total time approaches one millisecond. The yellow bars similarly quantify the end-to-end communication time spent through a socket between the engine and server when requesting a block from the server's storage. This time is also around one millisecond.

The turquoise-colored latencies located at the top of each of the bars describes the amount of computation time spent on the server and the maroon colored latencies sandwiched in between the two communication-time quantities is the computation time spent inside the engine. Since this time is so small, they can barely be seen on the graph at all. This tells us that computation is certainly not a problem, as expected. This also tells us that the half-duplex communication time is around 500 microseconds. The time spent in completing a ping on our network between two machines takes an average of 150 microseconds - a half-duplex time (ICMP request only) of about 75 microseconds. The kernel implementation will help bring the 500 microsecond number closer to that 75 microsecond lower-bound. Conclusively, the total time spent communicating in user-space is about 2 milliseconds versus the average 6.5 milliseconds to complete a request for a block using the disk.

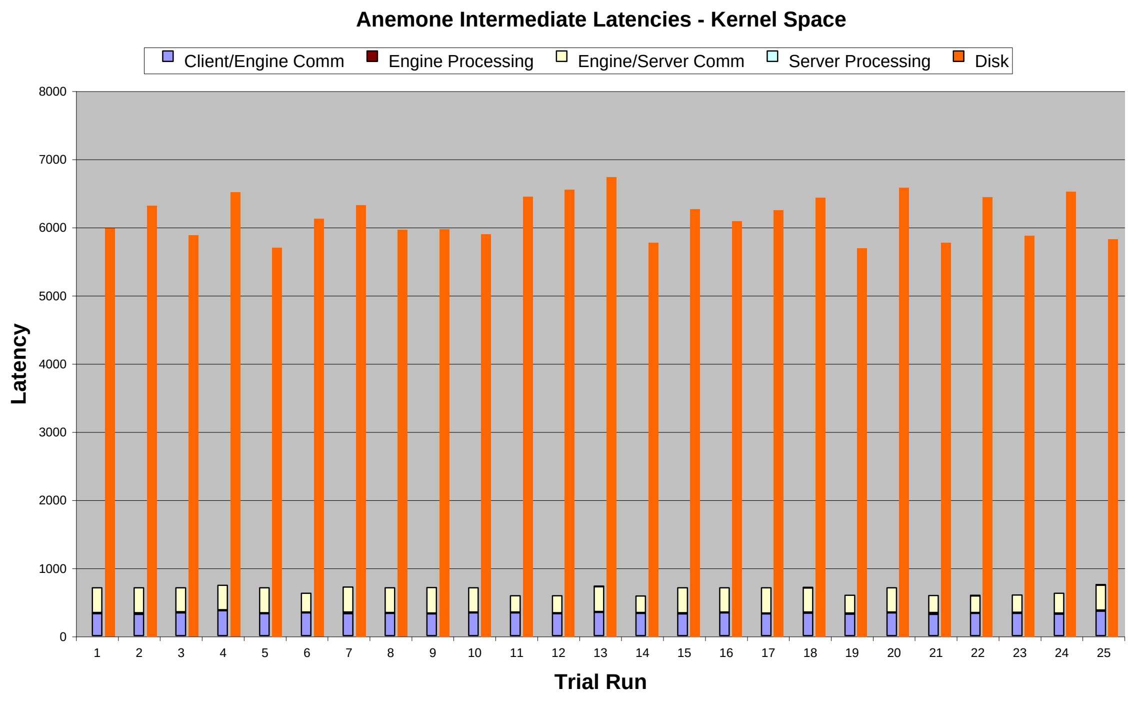

Figure 5.2. Kernel-Space Intermediate Latencies: The time it takes a client to request a block and the amount of latency involved in each step in the Anemone system. Disk latencies are the same as in Figure 5.1.

5.2.2 Kernel-Space

When Anemone's memory engine and memory server were implemented as kernel modules, we again did the same performance tests for individual block requests as we did above in user space. Figure 5.2 shows those results. The amount of time used internally for processing inside the engine and server is again negligible (within a few microseconds) and can barely be seen in the figure. Notice how the communication latency has been dramatically reduced and that the total roundtrip latency has gone down from 2 milliseconds to an average 700.2 microseconds. During testing, we observed that roundtrip latency reached as low as 599 microseconds and as high as 980 microseconds. (Note that the graph is reporting averages.) Additionally, the results in the figure show that client to engine communication averaged at 343.24 microseconds and engine to server communication averaged at 341.8 microseconds.

As expected these two numbers were constant, meaning the half-duplex speed between two machines is around 170 microseconds, only about 95 microseconds more than a half-duplex ping request - which is very low overhead. Furthermore, Recall that a network transmission takes 31.25 microseconds. This means that the device driver incurs about 170 - 31 = 139 microseconds, which is about 80% of the total network communication overhead. In the future, we may look into the possibility of modifying the device driver for optimization on the side of the engine and memory server only. This may tie us to just one NIC, but can shave 10s of microseconds of the total communication latency. Conclusively, the kernel-based version of Anemone is 9.2 times faster than disk-based block requests and 2.7 times faster than user-space based Anemone.

5.3 Single Memory-Bound Processes

The first single application we tested was a graphics ray-tracing program called POV-Ray [21]. Then we tested sorting, based on a quicksort, using an STL-based implementation by SGI [22]. Finally, we tested a network simulation using the popular simulator called ns2 [23].

5.3.1 Graphics

The graphics ray-tracing program POV-Ray was used with a scene rendered within a 310x310 unit grid, holding spheres of radius 1/4 units, all touching each other side-by-side, and one light-source - a total of about 380,000 spheres. Table 5.1 shows system-statistics resulting from the rendered scene. We measured these numbers using /usr/bin/time, as opposed to the built-in shell command. Memory usage for the entire system on all of our tests never exceeded 43%, as reported by top. The high (millisecond-level) page-fault service times in the figure include both the average 700.2 microsecond retrieval latency that was described earlier as well as the more substantial work involved in processing the page once it is received, copying the page to user-space, and performing a context switch to let the application continue running. It is calculated by taking the wall-clock time, subtracting off the wall-clock time obtained when the “No Swap” case is performed and then dividing by the number of page-faults. Notice that the values for System time using Anemone are much larger than the disk - this is naturally because Anemone services page-faults much

Table 5.1. Runtime memory statistics for a ray-traced scene in POV-Ray, averaged over 5 runs.

Statistic	No Swap	Disk Swap	Anemone
Max Used POV-ray Scene Memory	293 MB	293 MB	293 MB
System RAM	1266 MB	256 MB	256 MB
System swapspace used	0 MB	350 MB	350 MB
Major (I/O bound) Page-Faults	0	103661	102262
Minor (Reclaimed) Page-Faults	92185	564702	564244
User Time	14.25 sec	15.09 sec	14.9 sec
System Time	0.68 sec	371.8 sec	11.4 sec
CPU Usage	99%	27%	3.2%
Page-Fault Service Time	N/A	12.9 ms	6.5 ms
Wall-Clock Rendering Time	14.93 sec	1355 sec	677 sec
Anemone improvement factor for page-fault service time:			1.98
Anemone improvement factor execution time:			2.00

faster and incurs much less in-kernel work than the disk does. Anemone was able to improve execution time of a application of a single, dominant, memory-bound process by a factor of approximately 2 and a similar factor 1.98 improvement in page-fault service time. This improvement is restricted by the fact that the scene is rendered by one single-processing POV-ray process, who's memory utilization is restricted to only that one process.

5.3.2 Quicksort

Using the STL version of quicksort, which has a complexity of O(N log(N)) comparisons (both average and worst-case), we implemented a program which does a sort of a randomly populated space of integers that fills a 400 MB array in memory. Table 5.2 shows the statistics for this experiment. The table is in the same layout as the graphics test and shows that quicksort gained a factor 1.63 improvement over disk in page-fault service time and a factor 1.49 improvement in execution time. Later, we will use this same program while sorting different integer-array sizes to test Anemone under severe context switching among multiple, simultaneously running processes.

Table 5.2. Runtime memory statistics for the STL Quicksort, averaged over 5 runs.

Statistic	No Swap	Disk Swap	Anemone
Quicksort Array Size	409 MB	409 MB	409 MB
System RAM	1266 MB	256 MB	256 MB
System swapspace used	0 MB	350 MB	350 MB
Major (I/O bound) Page-Faults	0	7363	7857
Minor (Reclaimed) Page-Faults	0	108945	115351
Wall-Clock Time	6.59 sec	143.28 sec	96.08 sec
User Time	6.63 sec	7.19 sec	7.16 sec
System Time	0 sec	54.84 sec	0.9 sec
CPU Usage	99%	41%	7%
Page-Fault Service Time	N/A	18.56 ms	11.38 ms
Anemone improvement factor for page-fault service time:			1.63
Anemone improvement factor execution time:			1.49

Table 5.3. Runtime memory statistics for the ns2 network simulator, using 1500 wireless AODV-routed nodes in a 1500x300 area, 7 UDP connections, 64-byte packets, sending at 100 packets per second, running for 100 simulated seconds, averaged over 5 runs.

Statistic	No Swap	Disk Swap	Anemone
Max Observed Memory Usage	384 MB	384 MB	384 MB
System RAM	1266 MB	256 MB	256 MB
System swapspace used	0 MB	350 MB	350 MB
Major (I/O bound) Page-Faults	0	401342	360760
Minor (Reclaimed) Page-Faults	105773.8	509224	466785
User Time	1314.12	1137.81 sec	1139.0 sec
System Time	1.32 sec	1369.23 sec	28.99 sec
CPU Usage	99%	31.66%	31%
Page-Fault Service Time	N/A	16.403 ms	6.537 ms
Wall-Clock Simulation Time	1316.86 sec	7900.33 sec	3675.25 sec
Anemone improvement factor for page-fault service time:			2.509
Anemone improvement factor execution time:			2.149

5.3.3 Network Simulator

Finally, we tested the popular network simulator ns2 on Anemone. We simulated a scenario of 1500 wireless nodes in a 1500x300 area, 7 UDP connections sending 64-byte packets at a rate of 100 packets per second, running for 100 simulated seconds. Figure 5.3 shows the results of this experiment. The routing protocol used in the wireless scenario was AODV, the Ad Hoc On-demand Distance Vector protocol. The maximum observed memory used by this 1500-node scenario was 384 megabytes, just within range of our client's memory operating capacity, as was done with the other tests in order to stress the system. The improvement gained by using Anemone was the greatest in this experiment - a factor 2.5 improvement in page-fault service time and a factor 2.15 improvement in execution time. When doing very large and long-running network simulations, a factor 2.15 improvement can save valuable time for the user who needs experimental results. Consider a simulation that is expected to run for 2 days be cut in half to less than one day!

5.4 Multiple Processes

Single processes do not get much of a speedup because their computation and memory access is restricted to a single process, but when the system is under stress by multiple memory bound processes who are all competing for execution time and RAM, Anemone allows for much more efficient concurrent execution.

5.4.1 Stressing the system

We decided to use the quicksort algorithm from the previous section to test a stressed client system by running many, many processes until the entire available virtual memory space is saturated. Recall that our client machine is configured to use 350 MB of either disk-based swap or Anemone-based swap. Additionally, at any given moment there is about 100 to 110 MB of consistently free RAM when the system is not doing very much and is sitting idle. Thus, there is a total of 100 + 350 = 450 MB of memory for processes to use, and any number of simultaneously processes must compete for processor time, context switch, run, and do as many memory accesses as possible in the epoch alloted to it before the process gets paged out.

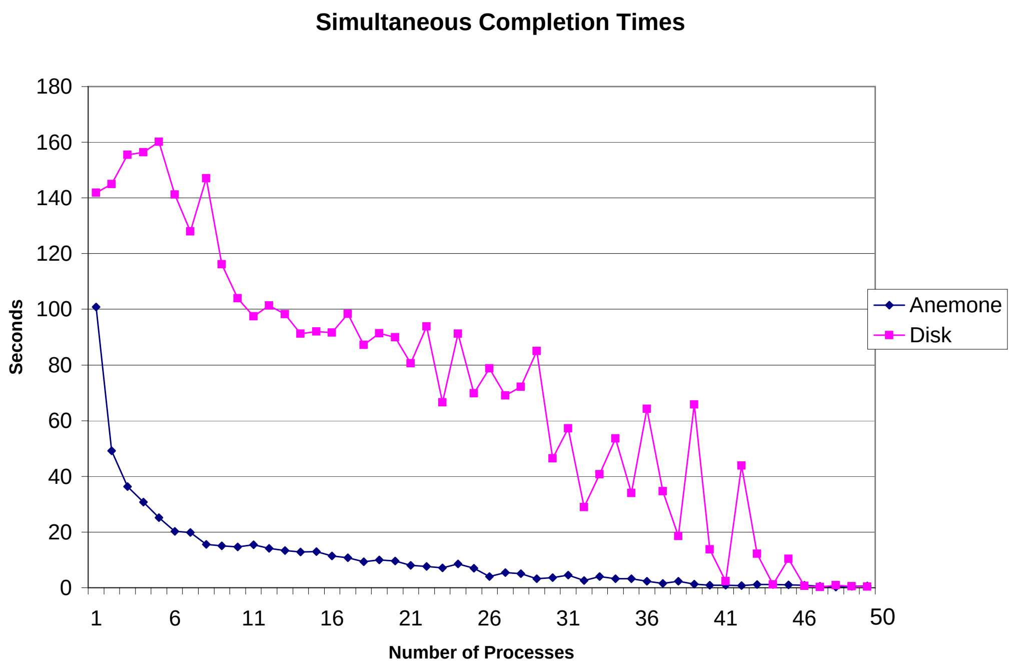

Figure 5.3. We run an increasing number of simultaneously running quicksort processes and time their execution. In each set of processes, each individual process sorts an array who's size is equal to total available memory (RAM and swap combined) divided by the number of simultaneously running processes for that particular run. Each of the 50 runs is tested using both Anemone-based and disk-based swap. Each individual run was performed 5 times and averaged.

5.4.2 Setup

To stress the system, we keep a consistent 400 MB of memory in an actively used state among both RAM and swap space out of the total 450 MB. We do this by first running one quicksort on an array of 400 MB. We then run 2 sorts in individual processes that sort smaller arrays, each of size 200 MB, running simultaneously until completion. We then execute 3 processes simultaneously, and 4 and so, dividing the 400 MB of active memory up by larger and larger numbers of simultaneous processes, all the way up to 50 runs (i.e. 50 processes). Finally, we record the time at which the very last process completed during a particular run. For instance, during run 20, we run 20 different quicksorts simultaneously, each sorting individual arrays of size 400/20 = 20 MB. We then record the time at which process #20

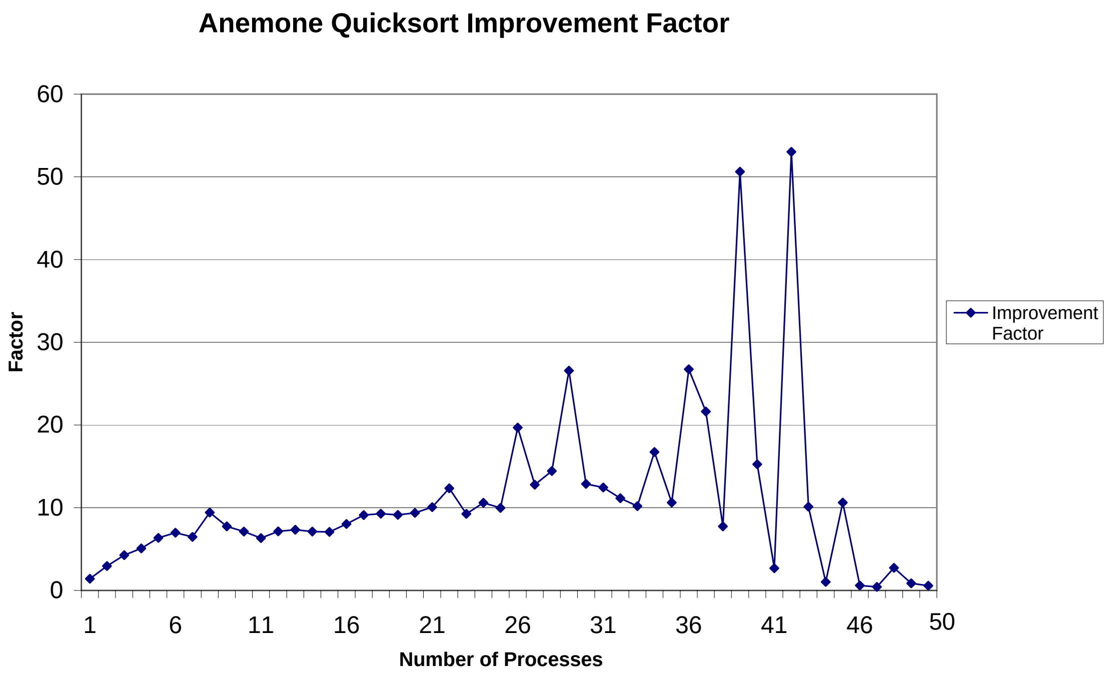

Figure 5.4. Using the values in Figure 5.3, we graph the ratio of the Anemone-based swap execution times for quicksort versus the disk-based swap.

completed. Each run is performed 5 times and the value of all 5 tests is averaged for that run. Figure 5.3 shows the results of this experiment.

5.4.3 Speedup

Anemone performs very gracefully. As the number of processes increases during each run, the amount of time for all the processes in the run to complete quickly decreases. For the same experiment using disk-based swap, execution times do decrease, but start much higher, decrease slower and become more erratic as there are more and more simultaneously running processes. The reason for the disk's increase in execution time is that as more and more processes are running, each has a memory footprint that cannot fit in memory, even as the size of each process begins to shrink. Eventually, the footprint of the sorted array for each process will become small enough such that quicksort takes shorter amounts of CPU time to sort the array. When a point is reached where the array is sufficiently small, quicksort will finish sorting the array so fast that a difference in execution between using Anemone and the disk becomes nonexistent. This happens after about 45 processes. Anemone never increases its execution time though - this is a natural reaction to the fact that the latency involved in Anemone-based paging is so low. A uniform performance increase happens just as it would if the client machine had a sufficiently large amount of RAM.

Figure 5.4 shows the ratio of the execution times of using Anemone versus the disk for the values show in 5.3. In the beginning of the experiment, the speedup quickly moves to a factor 10 around run 8. It then steadily increases and starts to fluctuate a lot between runs 26 and 45 and then it finally dies down. These drastic fluctuations, again, are the result of the smaller and smaller amounts of per-process computation that needs to be done as the array-sizes get smaller. The execution times of these larger amounts of processes depend on the performance of the scheduler, the epoch time and priority alloted to a processes, the number of page-faults generated by each process and exactly *how simultaneous* the processes are exec()'d together - i.e., how small is the amount of time between exec() calls for each process. In short, context switching and computation becomes the bottleneck instead of memory with so many processes running. Nevertheless, those may still get factor 10 performance improvements as long as they do not run an extreme number of simultaneous memory-bound process. For those that wish to use our system, keeping that number under 26 is not an unreasonable expectation.

5.5 Future Experiments & Summary

In light of the previous results, there are a number of other performance evaluations to perform:

1. We will more properly characterize the system under more dynamically changing system scenarios. Examples include the previously discussed increase in the number of processes running, different types of I/O from other sources within the client's system, and applications with other types of memory access patterns.

2. We will do performance tests involving background traffic. Client machines should not be expected to dedicate their network bandwidth to remote paging, so it would be

valuable to know how extra non-paging traffic affects the latency and execution-time improvements of remote paging with Anemone.

3. We also plan to run experiments involving cache-oblivious algorithms [24], which strive to maintain optimal use of cache without knowing the parameters of the system's cache configuration.

When running single, memory-bound applications that use as much of the system swap as possible, we observed that an improvement factor about 2 can be expected, and we suspect this to remain true for larger amounts of remote memory. The most noticeable improvement for single-process applications was the ns2 network simulator which showed a factor 2.5 improvement in execution time. Multiple, simultaneously executing applications demonstrate more of Anemone's power as different processes compete for CPU time and memory resources, which in turn causes lots of swapping. Using the quicksort algorithm, we demonstrated that a factor 10 improvement should be expected when the system is stressed with memory-bound processes.

CHAPTER 6

FUTURE WORK

Anemone can do a lot more. Here we discuss some of the ideas for improving and optimizing the already well-performing prototype. In Section 6.1 we discuss work that is currently being done to enhance Anemone. Section 6.2 talks about ways to further reduce the average 700.2 microsecond roundtrip latency involved in a page-fault. Finally, section 6.3 deals with issues on scalability and reliability of the system.

6.1 Work In Progress

6.1.1 Scheduling

Anemone is still a prototype and we are currently working on implementing the engine's ability to work with multiple backend servers. Our plan is to finish utilization-based scheduling method within the engine. The engine keeps track of what servers are offering remote memory, the amount they are offering, and exactly how much of that is used at any point in time. The basic method of taking the ratio of used memory to available memory for each server and choosing the lowest ratio for future pages will fairly store new pages. This will allow for the memory pages from an individual client to be distributed across all the contributing servers in the cluster so as to not stress any one server too much. Other metrics for utilization include the amount of traffic being sent to the servers and the amount of computation performed.

Additionally, the size of a VAI, the interface through which a client does its paging to Anemone must have memory statically reserved for it before a client can mount and use it. There are many VAIs offered by the engine. As a result, the amount of memory multiplexed to all the VAIs must not exceed the total memory capacity of the system provided by the servers. Other issues arise in VAI-to-server reservation when considering issues of replication and reliability. We expect to have a basic multi-server engine implementation before the beginning of the summer of 2005.

6.1.2 Caching

Another important in-progress component of Anemone is the engine's ability to cache pages. A colleague of mine, Mark Lewandowsky (lewandow@cs.fsu.edu) is working on an LRU-based caching mechanism. Caching will run side-by-side with the engine's normal functionality. When a page-out occurs, it gets stored in the cache first instead of being immediately transmitted to a particular memory server. If the cache is full, the least-recently-used page gets kicked out to a server and the new page is stored in the cache. This can reduce communication time for spatially/temporally located memory accesses by half, and hence significantly reduce the overall latency. Similarly for a page-fault, if a page is located in cache, the engine can immediately send it back to the client rather then spend time waiting for the page to be retrieved from a backend server.

6.2 Latency Optimization

Aside from doing integrated layer processing, future work with Anemone will implement a few more mechanisms to further reduce the significant communication latency involved in transferring pages through Anemone.

6.2.1 Split Page-In

Going back to the basics of the network card itself, how much of the communication latency is actually involved in the transmission of a 4 kilobyte page? Since the network cards used in our prototype are gigabit cards, they transmit at a rate of 1000 Mbits/second = 128,000 Kilobytes/second. A 4 kilobyte page would thus take 31.25 microseconds to be transmitted. One-way transmission of 4k pages for Anemone takes 170 microseconds. This means that there is about 140 microseconds being spent in the kernel between the time a packet is sent to Linux's "dev_queue_xmit()" function (for both receive and send) and the time it takes to actually transmit a packet onto the wire.

Recall that there are 4 network transmissions that happen during a page-fault:

1. An NFS request is transmitted to the engine
2. The engine transmits a request for the block to a particular server
3. The block is transmitted to the engine
4. The block is transmitted back to the client over NFS

If the server could transmit the memory page, including an NFS reply, directly to the client instead of going back through the engine, we could shave 170 microseconds off the roundtrip latency involved in a page-fault, bringing the latency down to an average 530 microseconds. We call this technique ”Split Page-IN”, splitting the responsibility for page-delivery between both the engine and the server. In addition to a reduction in latency, this can also help alleviate the burden of transmission and processing on the engine and allow the engine to scale and handle larger cluster sizes and traffic patterns better.

6.2.2 Compression

Based on the intermediate timing results presented earlier, The amount of processing time incurred at the engine for any request is extremely low and averages at 14.6 microseconds. At the server it's even lower than that - an average of 0.94 microseconds. If we could compress a memory page at the engine and only transmit fewer bytes to be stored on the server, it can significantly lower the amount of communication time involved in a page-fault by lowering two of the four total network transmissions. Although different compression algorithms are yet to be investigated, the additional computational overhead involved in compressing and de-compressing a 4-kilobyte page would probably still remain very negligible in comparison to communication overhead. Finally, not only do we get this benefit, but smaller pages in storage allow for servers to store even more data - a sort of “have your cake and eat it too” situation.

6.3 Scalability and Fault Tolerance

Remote memory availability is critical to the operation of the client because a process cannot continue without its pages. Two situations can potentially cause danger which Anemone must address: a server can go down or the engine itself can go down.

6.3.1 Replication

To combat the failure of a memory server, we plan to introduce the replication of pages to Anemone. Upon the scheduling of a new page on a particular server, the page is also scheduled and transmitted to a second backend server. If one server goes down, the page can be retrieved from the backup server. Any particular server will store both primary and replicated pages belonging to any client. Challenges to this idea include the amount of space necessary for replication: for full replication, at least half of the memory pool must be reserved just to do this. Compression has a third added advantage of helping offset this by reducing the space required to store a page in the first place. A second approach to this is to apply priorities to an individual VAI itself. For instance, a VAI might be in one of two classes: critical or non-critical. Pages belonging to clients who mount VAIs from the non-critical class can afford to give up the benefit of replication, depending on the type of application or processing the client is doing. Most clients will choose the former class. Nevertheless, such a cluster-based highly-managed system like Anemone is expected to have very good base reliability and can make the non-critical VAI a viable option in some situations.

6.3.2 Process-Pair

An equally important issue is the failure of the memory engine (a single point of failure, or SPOF) and/or when the engine degrades in performance as the amount of swapping by the clients in the cluster increases. One way to address this problem is to add multiple memory engines to Anemone. An individual client can be assigned to an engine through the use of dynamic DNS, for instance. Two different DNS requests can rotate name-to-IP address mappings upon connection to an engine, where all the engine's share the same DNS name. Multiple engines can serve two purposes, both of which must be intelligently and carefully managed. First, the engines can be designed to communicate with each other. This approach is traditionally called the "Process-Pair" technique, where engines provide meta-data with each other about their internal state, the clients they manage and the servers that they are connected to. Upon a page fault, the request can be "paired" or mirrored to the second engine but actually fulfilled by the primary engine. This open issue involves (1) allowing the backup engine to recover and intercept NFS requests if the primary engine fails, (2) synchronizing the two engines, and (3) determining the mechanism of failure detection, such as using a keep-alive style message. Again, the engine is expected to be a very powerful & reliable machine in the cluster but we intend to investigate these important ideas in the future.

6.4 Summary

We have described many methods for improving and strengthening the Anemone system. We plan to fully implement and finish a utilization-based scheduling mechanism to support multiple backend servers. Completion of this will allow for more work in important latency optimizations like split page-in, and compression & caching to improve the communication bottleneck involved in network transmissions. Finally, we will investigate better methods for handling scalability and fault tolerance, including multiple memory engines for load distribution & SPOF (Single Point of Failure) recovery and the use of page replication for higher availability and bandwidth.

CHAPTER 7

CONCLUSIONS

This paper presents the Anemone project - an Adaptive Network Memory Engine. Anemone pools together the memory resources of many machines in a clustered network of computers. It then provides an interface to client machines in order to use that collected memory resource in a virtualized manner, providing a potentially unlimited amount of memory to client machines that need it.

Although single memory-bound processes improve their execution time by a factor of 2, Anemone performs much better when the system is under stress from multiple competing processes. Not only does the system administrator get the benefit of a minimally configured client system and the latency/bandwidth benefits of Anemone, they also get the confidence of a system that performs much better under stress, resulting from many users or multiprocessing. Anemone can potentially be expanded into other areas of resource virtualization. Aggregated and transparent virtualization of processing or storage across many machines are examples of this. Future work in Anemone strives to combine the caching and compression of pages, more latency-optimizing techniques for faster page-faults and replication/pairing strategies that allow for higher scalability and reliability.

REFERENCES

[1] 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] E. Felten and J. Zahorjan. Issues in the implementation of a remote paging system. Technical Report TR 91-03-09, Computer Science Department, University of Washington, 1991.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] I. McDonald. Remote paging in a single address space operating system supporting quality of service. Tech. Report, Dept. of Computing Science, University of Glasgow, Scotland, UK, 1999.

[9] 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.

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

[11] 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.

[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.

[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.

[14] L. Xiao, X. Zhang, and S.A. Kubricht. Incorporating job migration and network RAM to share cluster memory resources. In *Proc. of the 9th IEEE Intl. Symposium on High Performance Distributed Computing (HPDC-9)*, pages 71–78, August 2000.

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

[16] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM '90: Proceedings of the ACM symposium on Communications architectures & protocols*, pages 200–208. ACM Press, 1990.

[17] Linux 2.4 Patch for swapping over NFS. *http://nfs-swap.dot-heine.de/.* 2003.

[18] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS version 3: Design and implementation. In *USENIX Summer*, pages 137–152, 1994.

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

[20] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel.*

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

[22] Inc. Silicon Graphics. *STL Quicksort.*

[23] Steven McCanne and Sally Floyd. ns2: Network simulator.

[24] 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.

BIOGRAPHICAL SKETCH

Michael R. Hines

Michael R. Hines was born and raised in Dallas, Texas in 1983. He spent his childhood playing classical piano. In 1999, he began college in a program called the Texas Academy of Math and Science at the University of North Texas. 2 years later he transferred to Johns Hopkins University in Baltimore, Maryland and received his Bachelor of Science degree in Computer Science in May 2003. Immediately after that he entered Florida State University to complete an Information Security certification in May 2004 and a Masters degree in Computer Science in May 2005. He will begin work on his Ph.D in Computer Science at FSU in the Fall of 2005.

Michael is a recipient of both the Jackie Robinson undergraduate scholarship and an AT&T labs research fellowship for his Ph.D. program beginning in 2005. He is a member of the academic honor societies Alpha Lambda Delta, Phi Eta Sigma and the Computer Science honor society Upsilon Pi Epsilon. His hobbies include billiards, skateboarding, and yo-yos.