



University  
of Glasgow

McDonald, Ian Lindsay (2001) *Memory management in a distributed system of single address space operating systems supporting quality of service*. PhD thesis.

<http://theses.gla.ac.uk/5427/>

Copyright and moral rights for this thesis are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the Author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the Author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

# Memory Management in a Distributed System of Single Address Space Operating Systems supporting Quality of Service

Ian Lindsay McDonald

Department of Computing Science  
University of Glasgow



A dissertation submitted for the  
degree of Doctor of Philosophy

November, 2001

Copyright © 2001 Ian McDonald. All rights reserved.

# Summary

The choices provided by an operating system to the application developer for managing memory come in two forms: no choice at all, with the operating system making all decisions about managing memory; or the choice to implement virtual memory management specific to the individual application. The second of these choices is, for all intents and purposes, the same as the first: no choice at all. For many application developers, the cost of implementing a customised virtual memory management system is just too high. The result is that, regardless of the level of flexibility available, the developer ends up using the system-provided default. Further exacerbating the problem is the tendency for operating system developers to be extremely unimaginative when providing that same default.

Advancements in virtual memory techniques such as prefetching, remote paging, compressed caching, and user-level page replacement coupled with the provision of user-level virtual memory management should have heralded a new era of choice and an application-centric approach to memory management. Unfortunately, this has failed to materialise.

This dissertation describes the design and implementation of the *Heracles* virtual memory management system. The Heracles approach is one of inclusion rather than exclusion. The main goal of Heracles is to provide an extensible environment that is configurable to the extent of providing application-centric memory management without the need for the application developer to implement their own. However, should the application developer wish to provide a more specialised implementation for all or any part of Heracles, the system is constructed around well-defined interfaces that allow new implementations to be “plugged in” where required.

The result is a virtual memory management hierarchy that is highly configurable, highly flexible, and can be adapted at run-time to meet new phases in the application’s behaviour. Furthermore, different parts of an application’s address space can have different hierarchies associated with managing its memory.

# Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of my dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed 50,000 words.

All trademarks used in this dissertation are hereby acknowledged.



# Acknowledgements

I am grateful to the following people for their endless support, constant encouragement, and constructive contributions throughout my period at Glasgow University.

- ◇ *Peter Dickman*, my supervisor, for his guidance throughout my PhD and much of my undergraduate days.
- ◇ *Richard Black*, for undertaking the role of second supervisor and for adding balance to the first.
- ◇ *Huw Evans*, for his seemingly inexhaustible consumption of drafts of various chapters and his pint-balancing trick.
- ◇ *Rolf Neugebauer*, for much of the Nemesis maintenance at Glasgow and for accompanying me on frequent visits to the snooker hall.
- ◇ The members of Systems Research Group here at Glasgow for the endless trivia that I hope I will never need. Two members in particular deserve a mention: *Michael* for cutting his hair; and *Matt* for his apparently unbounded gullibility – Matt, did you know the word “gullible” is not in the dictionary?
- ◇ My office mate *John Hagemeister*, a model PhD student who myself and Rolf have tried to corrupt as much as possible without much success.
- ◇ My viva panel for their constructive feedback and lively debate.
- ◇ My friends and family for their tolerance, support and gentle ego massaging.
- ◇ *Gillian*, for everything.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Thesis Statement . . . . .	3
1.2	Overview . . . . .	3
<b>I</b>	<b>Background</b>	<b>4</b>
<b>2</b>	<b>Approaches to Virtual Memory Management</b>	<b>5</b>
2.1	Multics . . . . .	6
2.1.1	The Segment Fault Handler . . . . .	7
2.1.2	The Page Fault Handler . . . . .	7
2.2	4.3 BSD UNIX . . . . .	8
2.2.1	Page Faults . . . . .	9
2.3	Mach . . . . .	10
2.3.1	Page Faults . . . . .	11
2.4	L4 . . . . .	12
2.5	Exokernel . . . . .	14
2.6	SPIN . . . . .	15
2.7	The Nemesis Single Address Space Operating System . . . . .	17
2.8	Discussion . . . . .	18
<b>3</b>	<b>Paging to Local Disk</b>	<b>20</b>
3.1	The Hard Disk . . . . .	21
3.1.1	Internal Performance Factors . . . . .	21
3.1.2	External Performance Factors . . . . .	23
3.1.3	The Effect of Performance Factors on Paging . . . . .	24
3.2	Beating the Bottleneck . . . . .	24

3.2.1	Paging Daemons . . . . .	25
3.2.2	Page Prefetching . . . . .	25
3.3	Discussion . . . . .	27
<b>4</b>	<b>Remote Paging</b>	<b>29</b>
4.1	Background . . . . .	30
4.2	Issues in Remote Paging . . . . .	36
4.2.1	Network Bandwidth . . . . .	37
4.2.2	Availability of Physical Memory . . . . .	39
4.2.3	Dynamic Versus Static Configuration . . . . .	40
4.2.4	Fault Tolerance . . . . .	41
4.2.5	Sub-Page Fetches . . . . .	45
4.2.6	Acquiring Physical Memory . . . . .	46
4.2.7	Revocation . . . . .	47
4.2.8	Quality of Service Guarantees . . . . .	51
4.3	Discussion . . . . .	52
<b>5</b>	<b>Compressed Caching</b>	<b>54</b>
5.1	Background . . . . .	56
5.2	Issues in Compressed Caching . . . . .	58
5.2.1	Compression Speed . . . . .	59
5.2.2	Compression Ratio . . . . .	61
5.2.3	Global Management . . . . .	61
5.2.4	Page Access Patterns . . . . .	62
5.2.5	Cache Size . . . . .	63
5.2.6	Cache Arrangement . . . . .	64
5.2.7	Responsibility . . . . .	66
5.3	Discussion . . . . .	68
<b>II</b>	<b>Towards a Better Environment</b>	<b>71</b>
<b>6</b>	<b>Discussion</b>	<b>72</b>
<b>7</b>	<b>Overview of the Nemesis Operating System</b>	<b>75</b>
7.1	Structure . . . . .	76

7.2	Quality of Service . . . . .	76
7.3	Virtual Memory Management . . . . .	77
7.4	Devices in Nemesis . . . . .	78
7.5	Sharing in Nemesis . . . . .	79
7.6	Nemesis IO System . . . . .	80
7.6.1	The Nemesis QoSEntry . . . . .	81
7.7	Discussion . . . . .	81

### **III The Heracles VMM System 83**

#### **8 Heracles System Design Overview 84**

8.1	Local Disk Manager . . . . .	85
8.2	Remote Paging . . . . .	86
8.3	Backing Store Manager . . . . .	87
8.4	Cache Manager . . . . .	87
8.5	Page Replacement . . . . .	88
8.6	Stretch Driver . . . . .	88
8.7	Dynamic Behaviour Modification . . . . .	89
8.8	Application Startup . . . . .	89
8.9	Application Control . . . . .	90
8.10	Extending Heracles . . . . .	90

#### **9 Local Disk Manager 92**

9.1	Basic Disk Manager . . . . .	92
9.2	Asynchronous Disk Manager . . . . .	93
9.3	Prefetching Disk Manager . . . . .	93
9.3.1	The Friends Algorithm . . . . .	94
9.4	Summary . . . . .	95

#### **10 Remote Paging in Heracles 96**

10.1	Remote Paging Protocol . . . . .	96
10.1.1	RPP Operations . . . . .	99
10.2	Remote Paging Trader . . . . .	99
10.2.1	Locating the Trader . . . . .	100
10.3	Remote Paging Server . . . . .	101

10.3.1	Server Control Path . . . . .	101
10.3.2	Server Data Path . . . . .	102
10.3.3	Advertising Threshold . . . . .	102
10.4	Remote Paging Client . . . . .	106
10.4.1	Page Compression . . . . .	107
10.5	Resource Revocation . . . . .	107
10.6	Quality of Service . . . . .	111
10.7	Handling Server Crashes . . . . .	111
<b>11</b>	<b>Cache Manager</b>	<b>113</b>
11.1	Basic Cache Manager . . . . .	114
11.2	Compressed Cache Manager . . . . .	115
11.2.1	Servicing a Cache Read . . . . .	115
11.2.2	Servicing a Cache Write . . . . .	115
11.2.3	Eviction Policy . . . . .	116
11.2.4	Managing the free space . . . . .	117
11.3	Extended Compressed Cache Manager . . . . .	117
11.3.1	Servicing a Cache Read . . . . .	118
11.3.2	Servicing a Cache Write . . . . .	118
11.3.3	Fetch Area Adaption . . . . .	119
11.4	Revocation of Physical Memory . . . . .	119
11.5	Compression Algorithm . . . . .	119
<b>12</b>	<b>Heracles Evaluation</b>	<b>120</b>
12.1	Experimental Results . . . . .	122
12.2	Experimental Variation . . . . .	129
12.3	The Cost of the Heracles VMM System . . . . .	130
12.4	Using the Heracles VMM System . . . . .	130
12.4.1	Application Startup . . . . .	131
12.4.2	Accessing Components from Within an Application . . . . .	132
12.5	Future Work . . . . .	134
<b>13</b>	<b>Conclusions</b>	<b>136</b>
	<b>Bibliography</b>	<b>139</b>

<b>IV</b>	<b>Appendices</b>	<b>148</b>
<b>A</b>	<b>Read/Write Interface</b>	<b>149</b>
<b>B</b>	<b>Disk Manager Interface</b>	<b>150</b>
<b>C</b>	<b>Remote Paging Server Interface</b>	<b>152</b>
<b>D</b>	<b>Remote Paging Client Interface</b>	<b>156</b>
<b>E</b>	<b>Cache Manager Interface</b>	<b>158</b>
<b>F</b>	<b>QoSEntry Interface</b>	<b>161</b>
<b>G</b>	<b>Test Applications</b>	<b>164</b>
G.1	Matrix Multiplication . . . . .	164
G.2	Image Filter . . . . .	165
G.3	SPEC Int 95 Compress . . . . .	166

# List of Figures

3.1	Parse Tree Constructed by the Character-Based Encoder $\mathcal{E}$ . . . . .	26
4.1	Remote versus Local Paging. . . . .	38
4.2	Effect of Limiting Server CPU Bandwidth and Network Access. . . . .	39
4.3	Free Memory on i386 Linux Machines . . . . .	40
4.4	Memory Consumption on a PC running Linux . . . . .	41
4.5	Mirroring Pages to Disk on Page Out . . . . .	43
4.6	Transparent Revocation. . . . .	49
4.7	Active Participation Revocation. . . . .	50
4.8	Unequivocal Revocation. . . . .	50
5.1	Compressed Memory Architecture. . . . .	55
5.2	Paging to/from a Compressed Cache . . . . .	56
5.3	In-Memory Compression Vs Reading and Writing from/to Disk. . . . .	59
5.4	In-Memory Compression and Copying for Different Processor Speeds. . . . .	60
5.5	Effect of Cache Size as a Percentage of Memory on Performance . . . . .	63
5.6	Effect of Adding Prefetching and Group Writes to a Compressed Cache. . . . .	64
5.7	Effect of Sub-Page size on Cache Performance . . . . .	65
5.8	Matrix Multiplication Without Competition for CPU and Disk. . . . .	67
5.9	Effect of CPU Competition and Having to Perform One's Own De/Compression. . . . .	68
5.10	Effect of CPU and Disk Bandwidth Competition on Compressed Caching Performance. . . . .	69
7.1	Nemesis Structure. . . . .	76
7.2	Nemesis VM System Architecture . . . . .	77
7.3	Self Paging versus External Paging . . . . .	78
7.4	Nemesis Device Driver Architecture . . . . .	79

7.5	Module Interface Data Structures in Nemesis . . . . .	80
7.6	High Volume I/O using Rbufs . . . . .	80
8.1	System Design . . . . .	85
8.2	VM Hierarchy Combinations. . . . .	86
9.1	A Group Split. . . . .	94
10.1	Successful Page Write. . . . .	98
10.2	Dropped Packet on Page Write. . . . .	98
10.3	Setting up a Server Link . . . . .	100
10.4	Algorithm for Monitoring Local Memory Commitments. . . . .	103
11.1	Paging to/from a Compressed Cache . . . . .	114
11.2	Cache Arrangement. . . . .	114
11.3	Cache Arrangement for Prefetching. . . . .	118
12.1	Demand Paging in Heracles. . . . .	122
12.2	Remote Paging in Heracles. . . . .	123
12.3	Remote Mirroring in Heracles. . . . .	124
12.4	Local Paging using Friends. . . . .	124
12.5	Effect of compressed cache on Compress. . . . .	125
12.6	Effect of compressed cache on Matrix Multiplication. . . . .	126
12.7	Effect of compressed cache on Filter. . . . .	127
12.8	Effect of combining a compressed cache with the friends algorithm on Filter. . .	128
12.9	Configuring the Nemesis Namespace to use Heracles. . . . .	131
12.10	Using Heracles Within an Application. . . . .	133
G.1	Locality of Reference in Matrix Multiplication $C = A * B$ . . . . .	165
G.2	Page Eviction Pattern in Data Compression Program . . . . .	166



# List of Tables

- 4.1 Performance trends in Disk, Memory and CPU. Note that the SPECInt performance is highly architecture-dependent and has since been superseded by SPECInt95 and SPECInt2000. . . . . 30
- 10.1 Memory startup costs of ten typical UNIX programs. . . . . 104
- 10.2 Possible MAX\_THRESHOLD values as a percentage of physical memory. . . . 105
- 12.1 Mean value and range of results for different paging configurations. . . . . 129
- G.1 Matrix dimensions and their associated memory costs. . . . . 164

# Chapter 1

## Introduction

*There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success than to take the lead in the introduction of a new order of things*

**Niccolo Machiavelli, *The Prince* (1532).**

A great deal of today's operating system (OS) literature focuses on extensibility. Be it user-level policies outside of the kernel, as in the case of  $\mu$ -kernels or exokernels, or downloading code into the kernel, as in so-called extensible operating systems, the trend appears to be moving towards application-specific environments. An important aspect of this new application environment is virtual memory management (VMM). The ability for application developers to provide their own virtual memory management has been seen as a significant step towards an application-specific environment. However, while application-specific virtual memory is a welcome sign, modern operating system developers have assumed too much. The cost of implementing an application-specific virtual memory system is simply too high for most application developers. Consequently, the additional flexibility offered by operating systems supporting this new feature goes widely unused. The result of which is that most developers end up using the system-provided default.

The tragedy of providing a powerful and flexible operating system, only to have application developers use system-provided defaults, is further exacerbated by the fact that operating system developers are extremely unimaginative when it comes to the provision of such defaults. There are many techniques available to system developers for improving application performance when virtual memory usage exceeds available physical memory. Techniques such as

compressed caching [Dou93, WKS99] can improve application performance considerably. Similarly, work in the area of remote paging [SD91, FMP<sup>+</sup>95, MD96] has shown some promising results. Although these techniques can greatly benefit applications, their performance relies on a great many factors. Compressed caching is significantly affected by a program's locality and can severely debilitate performance under some circumstances. Remote paging is affected by network bandwidth and congestion and can, under some circumstances, be outperformed by a local disk and a reasonably intelligent paging strategy.

This dissertation describes the *Heracles* virtual memory management system. Heracles provides a virtual memory framework that can be customised by the application developer to meet the individual demands of their application without the need to implement their own strategy. The system combines disk managers that support asynchronous writes, prefetching and contiguous writes; a compressed cache manager that compresses pages in memory in order to “extend” the amount of physical memory available; and a remote paging subsystem that provides reliable delivery of pages and supports fault tolerance. Heracles is further augmented by the support for user-provided page replacement policies. The application developer can select which components make up the virtual memory hierarchy for their application. They can further parameterise the individual components that make up that hierarchy. The result is an extremely flexible and highly configurable virtual memory management system that provides application-specific memory management without the need for application developers to implement their own strategy.

Although Heracles is extremely flexible and highly configurable, the need for more specific policies for individual components should not be ignored. Consequently, the interface to each component in the Heracles system is specified in an interface definition language. Should a more specific implementation of a component be required, it can easily be “plugged in” to the existing framework without the need to recompile any of the other components.

Heracles is implemented in the Nemesis single address space operating system. Nemesis forces all applications to perform their own virtual memory management and supports quality of service (QoS) for resources. Applications can acquire guarantees for resources such as the CPU, the local disk and access to the network interface. These guarantees come in the form of a slice  $s$  per period  $p$  with an additional flag informing the system if the application will accept additional slack time.

## 1.1 Thesis Statement

While it is generally regarded within the OS community that application-centric virtual memory management offers great potential for improving application performance, the move towards it has been hampered by the assumption that developers would implement their own VMM system. While this may be acceptable for a small minority, it effectively places this opportunity out of reach for the vast majority of developers; for these people the cost of implementing a customised VMM system is simply too high.

I assert that the power and flexibility offered by application-centric VMM can be made available to developers without the need for the implementation of a customised VMM system. By providing a highly flexible and customisable VMM system that can be tailored to the individual application, or indeed to part of an application's virtual memory, it is possible to improve application performance at very low cost. To take full advantage of this, the developer need not understand the complexities of virtual memory management, although they may have to understand the behaviour of their application. Furthermore, this flexibility can be extended, should it be desirable, to the application user who is in a position to determine system resources when the application is run.

## 1.2 Overview

The rest of this dissertation is structured as follows:

- Part I (chapters 2 to 5) provides the background to Heracles, looking at virtual memory management in operating systems as well as the issues involved in local paging, remote paging and compressed caching.
- Part II (chapters 6 and 7) sums up the conclusions drawn from part I and describes the platform on which Heracles is implemented.
- Part III (chapters 8 to 13) describes and evaluates the Heracles system, as well as drawing some conclusions and suggesting future work.

# **Part I**

## **Background**

## Chapter 2

# Approaches to Virtual Memory Management

*Memory is like an orgasm. It's a lot better if you don't have to fake it.*

**Seymore Cray, on virtual memory.**

The advent of virtual memory heralded a new era in computing: not only could applications be larger than the physical memory of the machine, but there was also no need to recompile applications whenever the amount of physical memory changed. It also, to a large extent, freed programmers from the constraints of managing their own memory and allowed them to concentrate on the functionality of their programs. However, with this new programmer freedom came the added complexity of providing virtual memory management. Virtual memory brought new challenges to the operating system (OS) developers and, over the years, there have been many approaches to virtualising physical memory.

Discourse on the subject of virtual memory has been extensive and there have also been many reviews of aspects of virtual memory in many different systems: how different operating systems provide different page replacement strategies [Nel86]; the effect of page table structure [Elp93, Han99] on performance; the relative performance of paging to the local disk versus other methods, such as compressed caching [Dou93, WKS99] and remote paging [FMP<sup>+</sup>95, MD96]. The purpose of this chapter is not to provide an exhaustive study of virtual memory management (VMM). Instead, we focus on the choices afforded the application developer by different operating systems when it comes to managing the application's memory.

In this chapter, we concentrate on a collection of representative operating systems. More specifically, we focus on how different operating systems manage memory: how they divide physical memory between different processes, handle page faults, and select victim pages for replacement. It is shown that although there are many different operating systems supporting many different features, there are basically only two approaches to memory management: either the operating system handles all aspects of memory management or it allows the developer to provide their own implementation of particular aspects, e.g. page fault handling. It is argued that the choice between system provided and infinite possible developer implementations is in fact no choice at all. For most application developers, the cost of implementing a customised VMM system is just too high. This results in applications resorting to the system-provided implementation, regardless of what operating system the application is running on.

We begin our discussion of memory management in sections 2.1 and 2.2 with Multics and 4.3 BSD UNIX, representing the monolithic operating systems. The VMM in Mach, the embodiment of first generation  $\mu$ -kernels, is described in section 2.3 and L4, representing second generation  $\mu$ -kernels, is described in section 2.4. The library operating systems approach is represented in section 2.5 by the exokernel model. SPIN represents the extensible OS approach in section 2.6. Finally the single address space operating system approach, as represented by the Nemesis OS, is presented in section 2.7, before drawing some conclusions in section 2.8.

## 2.1 Multics

The Multics [DD68, Org72, BCD72] operating system was designed with the aim of facilitating the sharing of data in memory. Through segmentation, Multics provides direct hardware addressing by user and system programs of all information, regardless of its physical location. There is a one-to-one correspondence between processes and address spaces in Multics. Each segment can be mapped into a process' address space, providing that process possesses the appropriate access rights. This method of sharing presented a move away from the traditional file-level sharing in previous systems. A key factor of the Multics segmented approach is that segments are paged. This means that not all of a segment need be resident in physical memory at once.

In Multics, memory is split into fixed-sized segments which, in turn, are split into fixed-sized pages. Addresses consist of a pair  $[s, i]$  (where  $s$  is the segment number and  $i$  is the index into  $s$ ), where each is in the range of  $0..(2^{18} - 1)$ . Each segment is referenced via a page table (PT). The

PT for a segment is an array of physically contiguous words in core memory. Each element in the PT is called a page table word (PTW). All segments are paged (page size = 1024 words) and  $i$  refers to the  $w^{th}$  word of the  $p^{th}$  page, where:  $w = i \bmod 1024$ , and  $p = \frac{(i-w)}{1024}$

### 2.1.1 The Segment Fault Handler

When a missing segment fault is generated, the segment fault handler (SFH) takes control and stores the proper segment attributes in the segment descriptor word (SDW) and resets the missing segment switch. These attributes consist of page table address, segment length, and access rights. If the page table does not exist, one is created for it. All page tables are the same size and a portion of core memory is set aside for them, the size of which, and subsequently the number of page tables, is assigned at system initialisation time.

A portion of memory is permanently reserved for recording attributes needed by the page fault handler, e.g., the segment map and segment length. This is referred to as the active segment table (AST) and contains one entry per page table. A page table is always associated with an ASTE (active segment table entry), the address of one implying the address of the other - referred to as the (PT, ASTE) of a segment. A segment with a (PT, ASTE) is said to be *active*.

When the active switch of a segment is ON, both the segment map and segment length are no longer stored at the branch but are in the segment's (PT, ASTE), whose address is recorded in the branch on activation.

Once a segment is made active, the corresponding SDW must be *connected* to the segment. This involves storing the absolute page table address, the segment length, and the segment access rights and then turning off the missing segment flag.

### 2.1.2 The Page Fault Handler

On a page fault, the page fault handler is given control with the PT address and the page number of the faulting page. To bring the page in it requires the address of a free frame and the address of the page in secondary storage. A free frame is found by a look-up of the *core map*.<sup>1</sup> The core map is divided into 2 lists: a free list and a used list. If a free frame is available the address of

<sup>1</sup> An array of core map entries (CME) where the  $n^{th}$  entry contains information about the  $n^{th}$  frame.



the page in secondary storage is found via the ASTE. If the address is “null” a page is zeroed in to the empty frame; otherwise a request is issued to the I/O system for the page and the process blocks awaiting completion of the request. The request is issued via a call to the *traffic controller* which is responsible for processor multiplexing. Next, the core address is stored in the PTW, the fault is removed and the core map entry is placed in the used list.

To ensure that in most cases there is always a free frame, on removing a free frame from the core map the length of the free list is examined. If the length is below a certain threshold, pages are expelled until the length is once again above the threshold. The selection algorithm for page expulsion is a variant of global least recently used (LRU). Every time a page is referenced, the hardware sets a *used* flag in the corresponding PTW. Each entry in the used list must also have a pointer to the PTW.<sup>2</sup>

Once a victim page has been selected, its PTW is set to missing. If no secondary storage has been allocated, space is assigned for it and the address is stored in the segment map. A call is then made to the I/O system and the page is transferred to backing store.

Although address translation consists of five levels of indirection, making translation more expensive than its contemporaries, the Multics system offers a higher degree of flexibility than some more modern operating systems. It provides support for sharing virtual memory, as well as segmented address spaces and paging. Multics is generally considered a milestone in virtual memory management and some of its features have found themselves incorporated into operating systems throughout the years. However, although Multics is perhaps more appreciated nowadays than it was at the time, it offers little choice to the application developer in terms of managing their own memory. The operating system takes all the decisions regarding the eviction of pages and the handling of page faults.

## 2.2 4.3 BSD UNIX

The UNIX operating system represents the standard against which all other operating systems are judged. Indeed, it seems nowadays that operating systems must not only outperform UNIX but must also be able to support it and its applications. Although we concentrate on 4.3 BSD

---

<sup>2</sup>This is set by the page fault handler when a page is moved into the used list.

[LMKQ89] in this section, the general principles discussed are the same for most flavours of UNIX.

The UNIX operating system provides a one-to-one mapping from process to virtual address space (VAS) and this forms the basis of its protection model. Address translation is carried out within the scope of a process' own address space and one process cannot access the address space of another. Unlike the Multics system, UNIX was designed for isolation rather than sharing. A consequence of this is that sharing in modern unices has a very "tacked-on" feel.

The *core map* is the central data structure used to manage physical memory. It consists of an array of structures, one entry per cluster of memory frames, excluding those allocated to the kernel. A cluster typically consists of a pair of frames. Because there is a one-to-one correspondence between the core map array and the page clusters, it is simple to locate one given the other.

Unlike the Multics system, user page tables are located in virtual, rather than physical, memory. The hardware locates these page tables in kernel virtual memory and not all of the page table need be resident in physical memory. The virtual address translation is a 2-step translation using a 2-level page table.

### 2.2.1 Page Faults

When the memory hardware encounters a valid bit that is 'off', the pending instruction is, typically, backed up to the beginning, the page is fetched from disk, or a new one is allocated, and the system returns from the trap. The return to user mode allows the CPU to re-attempt execution of the original instruction.

If there is no free frame to page into, then the servicing of a page fault is delayed until a page is evicted to make room for the faulting page. To prevent this from happening, 4.3BSD uses a page daemon in an attempt to ensure there is always a free frame. The page daemon uses a second-chance FIFO algorithm, implemented using a two-handed clock, for selecting suitable 'victims'. Each hand of the clock points to an entry in the core map a fixed distance apart. Every 250ms, the system checks the availability of physical memory. If physical memory needs to be freed, the page daemon iterates over the core map and frees enough clusters to maintain a system-specified threshold.

The key factor to note about the replacement algorithm is that it is global: a victim cluster is selected purely on the basis of whether it was referenced between the front and the back hand passing over it in the core map. This means that a process can find a cluster of its pages evicted to make way for the pages of another process.

Although 4.3 BSD may not be considered the modern epitome of UNIX, the core ideas are still the same. The UNIX-like model of memory management is extremely restrictive and offers the developer little choice in how the memory for their application is managed.

## 2.3 Mach

The Mach kernel [RTY<sup>+</sup>88] significantly extends the UNIX notion of virtual memory management. The aim of Mach was to move some of the decision making process out of the kernel and into user space, providing a more flexible environment. Mach supports:

- large, sparse address spaces
- copy-on-write virtual copy operations
- copy-on-write and read-write memory sharing between tasks<sup>3</sup>
- memory mapped files and
- user-provided backing store objects and pagers

There are four basic memory management data structures used in Mach:

- *the resident page table*  
tracks machine independent pages
- *the address map*  
maintains mappings between ranges of addresses and memory objects
- *the memory object*  
the unit of backing store managed by the kernel or a user task

---

<sup>3</sup>A task is equivalent to a UNIX process

- *the pmap*

machine dependent memory mapping data structure (i.e. hardware defined physical address map)

A key Mach abstraction is the memory object: a collection of data provided and managed by a server which can be mapped into the address space of a task. A managing task (sometimes called a pager) is associated with each memory object. The kernel manages pages in physical memory, while those not in physical memory are stored and fetched by a pager via messages from the kernel. The pager may be internal to the kernel or an *external* user-task; the default is internal and pages to files.

The provision of support for *external memory management* is not without problems, many of which were highlighted by the designers themselves, however the Mach VM system is extremely flexible and forms an important break with the UNIX notion of memory management. For example, the authors of [YTR<sup>+</sup>87] describe the implementation of distributed shared memory (DSM) on Mach, using a memory object supporting this feature, without the need to alter the kernel in any way.

### 2.3.1 Page Faults

The kernel maintains two internal queues: one holds information on active pages in LRU order; the other holds information on pages waiting to be paged out. Like UNIX, the way in which the physical memory is shared between tasks is managed by the operating system. In Mach, the kernel ‘caches’ pages belonging to tasks; pages held in the cache are in physical memory. On a page fault, the kernel sends a message to the appropriate memory object requesting the faulting page. The memory object returns that page, and may also return others. If there is no room in the cache, the kernel first selects a victim page and sends a message to the appropriate memory object informing it to write the page and subsequently free the space. When the message is sent, the page is added to the “awaiting paged-out” queue; the kernel does not wait on the pager performing the write and freeing the space. The developers of Mach admitted that there was a potential problem if the external pager did not free the page in the “awaiting paged-out” queue — the kernel did not reclaim this space until it was freed.

Although the flagship of first generation  $\mu$ -kernels, Mach fundamentally represents an attempt to reduce UNIX to a  $\mu$ -kernel. Consequently, there are several UNIX hangovers: the kernel

manages the physical memory as a global cache on behalf of all tasks, and it is the kernel which selects the page to be expelled from the cache to make way for another.

The main goal of Mach was to move away from the primitive virtual memory management offered by traditional UNIX systems, showing that much of the memory management can be moved out of the kernel without affecting performance. However, Chen and Bershad [CB93] showed that Mach did not perform as well as a monolithic kernel (Ultron) and highlighted the ramifications for caching and TLB performance in providing external memory management. Despite this, the VM system used in 4.4 BSD [MBKQ96] is largely based on the Mach VM system.

Other  $\mu$ -kernel systems have aimed at improving some of the perceived problems with Mach. The V++ Cache Kernel [CD94] represented an attempt to reduce the size of the  $\mu$ -kernel, pushing more of the decision making into application space. Although the authors report comparable performance with monolithic systems while offering application-level control of system resources, V++ provides the application developer with the same memory management choices as Mach.

## 2.4 L4

A well known problem with  $\mu$ -kernels is their poor performance relative to monolithic kernels. The reason for the relatively poorer performance has often been attributed to the  $\mu$ -kernel model. In particular, the costs of inter-process communication (IPC) and context switching have been highlighted as the main problems and are often seen as inherent inefficiencies with the  $\mu$ -kernel model. Liedtke [Lie95] argues that the inefficiencies attributed to the  $\mu$ -kernel model are a result of poor implementation of  $\mu$ -kernels and are not inherent in the model itself. Indeed, in the construction of the L4  $\mu$ -kernel, he shows that both IPC and address space switches can be extremely fast.

The L4  $\mu$ -kernel [Lie96a] in itself does not represent an operating system, but instead provides a minimal framework for the construction of arbitrary operating systems on top of it. L4 defines only three operating system concepts supported by the  $\mu$ -kernel itself: address spaces, threads with IPC, and unique identifiers.

Address spaces are described as mappings. The initial address space is represented by  $\sigma_0 : V \rightarrow R \cup \{\phi\}$ , where  $V$  is the set of virtual pages,  $R$  is the set of available physical pages (real) and

$\phi$  is the nilpage which cannot be accessed. Further address spaces are defined recursively as mappings  $\sigma : V \rightarrow (\Sigma \times V) \cup \{\phi\}$ , where  $\Sigma$  is the set of address spaces. The  $\mu$ -kernel provides three operations for building address spaces on top of  $\sigma_0$ :

- **Grant** : the owner of an address space can grant any of its pages to another space. The granted page is removed from the granter's address space and included in the recipient's address space.
- **Map** : the owner of an address space can map any of its pages into another address space. Afterwards, the page is accessible in both address spaces.
- **Flush** : the owner of an address space can flush any of its pages, leaving the pages accessible in the flusher but removing them from all other address spaces which received them, directly or indirectly, from the flusher.

In the case of grant and map, the recipient must agree to the operation. There is no agreement required for a flush as the recipient of a page has to accept that a flush may be invoked on any pages it receives by mapping or granting.

A memory management server  $M_0$  manages the initial address space  $\sigma_0$ . It can map or grant physical memory to  $\sigma_1$ , managed by  $M_1$ ,  $\sigma_2$ , managed by  $M_2$ , and so on. By integrating the memory manager with a pager, it is possible for a server to closely control the memory management of an application.

This address space model leaves memory management and paging outside of the kernel. Moving memory management, and not just paging, into user-space is a key difference between L4 and other  $\mu$ -kernels and gives it a much greater degree of flexibility.

As mentioned previously, L4 is not a complete operating system in its own right. However, by providing only a minimal framework, other operating systems, with vastly different models, can be built on top of it. For instance, the Mungi single address space operating system [HEV<sup>+</sup>97] was ported to run on top of L4, as was the Linux operating system [HHL<sup>+</sup>97], supporting the claim that second generation  $\mu$ -kernels may be able to support a variety of operating systems [Lie96b].

Although both efficient and flexible, the L4  $\mu$ -kernel shares some of the problems of other  $\mu$ -kernels. For an application to fully take advantage of the memory management features, the

application developer must build a memory manager and a pager. Furthermore, the external server approach means that a heavily faulting application can affect the performance of other applications by over-utilising the kernel.

## 2.5 Exokernel

The exokernel [EKO95, Eng98] is an architecture for the construction of library operating systems (libOS) on top of a minimal kernel. The motivation is to provide a minimal abstraction over physical resources, allowing specialised library operating systems to take advantage of this low-level exposure in order to improve performance. The authors argue that traditional operating systems are too general and that  $\mu$ -kernels must place their trust in a shared server to execute privileged code. Library operating systems can be simpler and more specialised and incur fewer context switches as most of the OS is executed in user-mode. Furthermore, by employing object-oriented techniques, they are easy to extend and parts can be easily re-used.

There are four key design principles behind the exokernel architecture:

- **Securely expose hardware.** The hardware should be accessed as directly as possible including: physical memory, CPU, disk, and translation lookaside buffers (TLB). This principle is also extended to interrupts, exceptions and cross-domain calls.
- **Expose allocation.** The libOS should be able to request specific resources. For instance, allowing a libOS to select specific physical pages provides a means for reducing cache conflicts among pages in its working set.
- **Expose names.** The exokernel should expose physical, rather than symbolic, names. In addition, book-keeping information, such as cached TLB entries and disk arm position, should be exposed.
- **Expose revocation.** The libOS should be aware that resources are being revoked and be allowed to participate in selecting suitable instances.

Due to the nature of the exokernel environment, there must be a means of arbitrating between competing libOSes. The exokernel must decide which resource requests to grant and which ones to refuse. These policies are dependent on the particular implementation.

A key feature of the exokernel architecture is protection via secure bindings. The kernel authorises bindings to resources at bind time only, separating the protection from the management. Secure bindings to physical memory are implemented using a combination of capabilities and the address translation hardware. The implementation of the capabilities depends on the particular exokernel. For example, the Aegis exokernel [EKO95] uses self-authenticating capabilities and the Xok exokernel [KEG<sup>+</sup>97] uses hierarchically named capabilities. On access to a physical page, the libOS must provide the appropriate capability. In keeping with the central tenet of exposing as much as possible, the page table is accessible, read-only, by the libOSes.

A consequence of the libOS approach is that virtual memory management occurs at user-level [EGK95]. Should an application developer desire application-specific virtual memory management, he would have to provide his own implementation. The alternative is to use a system provided default or an implementation aimed at another application which may be more suitable than the default.

## 2.6 SPIN

Another method of providing application-specific virtual memory management is to allow application extensions to the kernel. SPIN [Ber94] does this via events and event handlers. A kernel extension installs a handler on an event via a central dispatcher that routes events to handlers. SPIN relies on four techniques implemented at the language level or its run-time:

- **Co-location:** OS extensions are dynamically linked into the kernel virtual address space.
- **Enforced Modularity:** extensions are written in Modula-3 and the compiler enforces interface boundaries between modules.
- **Logical Protection Domains:** extensions exist within logical protection domains, which are kernel namespaces containing code and exported interfaces. An in-kernel dynamic linker resolves code in separate logical protection domains at run-time, enabling cross-domain communication to occur which has the same overhead as a procedure call.
- **Dynamic Call Binding:** extensions execute in response to system events, such as page-faults or thread scheduling.



Object files are deemed safe if they have been signed by the SPIN Modula-3 compiler or asserted safe by the kernel. This latter method was a convenient way to use existing device driver implementations. All kernel resources are referenced by capabilities: an unforgeable reference to a resource, which can be a system object, an interface or a collection of interfaces.

The SPIN memory services consist of three basic components: physical storage, naming and translation. Application-specific services interact with the SPIN services to define higher level abstractions such as address spaces. Clients can request physical memory and receive a capability for the memory they get. The virtual address service allocates capabilities for virtual addresses and the translation service interprets virtual and physical references, and is responsible for constructing mappings between the two.

The translation service raises a set of events corresponding to certain memory management unit (MMU) conditions, such as a page fault. Implementors of higher level memory management abstractions can install handlers to define services such as demand paging, distributed shared memory (DSM) and concurrent garbage collection.

The developers of SPIN decided on kernel extensions as a way of providing improved performance over  $\mu$ -kernel approaches. It was developed under the assumptions that context switching and IPC are inherently expensive. However, section 2.4 would suggest that these assumptions are incorrect. Downloading code into the kernel still has the advantage of not requiring cache and TLB flushes, but limiting the extensions to the system's modula-3 compiler reduces the flexibility somewhat. The developers were forced to relax this rule in order to utilise existing device driver implementations.

Memory management choices in SPIN are essentially the same as those for Mach and exokernel: use the default or implement your own. Although this provides the appearance of empowering the application developer, it offers very little option in real terms.

The extensible approach is also advocated by the authors in [SSS95] and their approach features in the Vino operating system [SS94]. The Vino system performs a great deal of information gathering in an attempt to flag key areas affecting system performance [SS97]. Like SPIN, Vino requires code from the application developer in order to take advantage of its extensibility.

## 2.7 The Nemesis Single Address Space Operating System

With the advent of 64-bit processors such as Sun's UltraSparc-III [SUN98] and Digital's Alpha architectures [Sit92], OS developers are being confronted with new challenges. "Unlike the move from 16- to 32-bit addressing, a 64-bit address space could be revolutionary instead of evolutionary with respect to the way operating systems and applications use virtual memory" [CLBHL92]. This is highlighted by the rather amusing, though slightly misleading, claim that a "full 64-bit address space, consumed at the rate of 100Mb per second, will last for nearly 5000 years" [KCE92]. This has led to a revival in the interest in single address space operating systems (SASOS). Furthermore, it has led many developers to believe that we can extend the address space to map everything on the system - not only data in physical memory, but also data on long-term storage and across the network. Some systems, such as Opal [CLBHL93] and Mungi [HERV93] have seen the SASOS as a catalyst for distributed shared memory where the single address space is distributed over all the nodes in a local area network (LAN).

The Nemesis operating system [Ros95] is similar in concept to the exokernel model. It focuses on providing mechanisms for a rich and highly specialised environment where the NTSC<sup>4</sup> is responsible for the secure multiplexing of resources and the application domain<sup>5</sup> is responsible for everything else. Most code traditionally executed by the operating system on the application's behalf is executed directly by the application in the same protection domain. The result is what is termed a vertically structured operating system [Bar96].

Nemesis splits the address space into sections called stretches [Han97]. A stretch is an abstraction over a contiguous region of the virtual address space where every page has the same access rights. A stretch cannot shrink or grow once created and different stretches cannot overlap. A stretch is only meaningful when bound to a stretch driver. Any attempts to address memory in an unbound stretch will result in an unresolvable page fault.

The stretch driver handles page faults, implements the replacement strategies, and performs virtual to physical mappings. The closest analog to a stretch driver would be a combined memory manager and pager in L4.

On a page fault, the NTSC sends an event to the appropriate domain. When that domain is reactivated it should resolve the fault. This may involve replacing a page currently in physical

---

<sup>4</sup>Nemesis Trusted Supervisor Code: equivalent to an extremely small kernel.

<sup>5</sup>A domain is analogous to a process in UNIX or a task in Mach.

memory – the method used to do this is entirely the responsibility of the stretch driver for a particular domain.

Due to the fact that Nemesis is a SASOS, it does not suffer from the need to flush the cache and TLB (though it must flush protection information) on a context switch (providing the cache is virtually addressed). Therefore, there is not the same penalty for performing the memory management at the user level as there is for  $\mu$ -kernel operating systems. However, Nemesis does rely on the application developer providing their own virtual memory management.

The Nemesis operating system is described in more detail in chapter 7.

## 2.8 Discussion

In this chapter, we have looked at seven different operating systems and examined the choices available to the application developer with respect to managing their own virtual memory. Although each operating system was fundamentally different, we have encountered only two approaches to virtual memory management: either the OS provides a generalised “best guess” approach, or the application developer provides their own specialised implementation. While this last approach is generally considered to offer more potential for better performance, it has the considerable drawback of placing large demands on the application developer. Consequently, the operating systems offering the “do-it-yourself” approach to VMM tend to provide a default memory manager that most applications end up using whether they would benefit from a more specialised solution or not. For many applications, the cost of constructing a customised implementation is too high. Thus, these systems often end up being simply slower versions of the monolithic systems.<sup>6</sup>

That user level virtual memory management can offer a much more flexible and specialised environment is undoubted. However, the idea that application developers will readily provide their own VMM implementations seems somewhat misplaced. For the most part, the application developer is unaware of the mechanisms involved in managing their memory. They are aware of higher level notions relating to their program: locality of reference, compressibility of data, what

---

<sup>6</sup>It is widely regarded that it is the flexibility aspect of user level virtual memory management that offers the potential for performance gains; if this flexibility goes unused, i.e., applications use the default implementation, then the result is equivalent to a monolithic kernel with more indirection and poorer cache and TLB performance.

pieces of information it is important to have in memory. These are notions that can be derived from the program itself. What cannot be immediately derived from the program is its paging behaviour, where its data will be in memory and what pieces of data will be next to others.

Of course there are certain types of program where information that can be exploited by user level decisions is well known. In particular, it is well known that an LRU page eviction scheme performs extremely poorly for certain applications, such as multimedia and database programs. Lee et.al. [LCC94] implemented user-level page replacement on the Mach 3.0 kernel to allow such applications to provide their own page replacement implementation via the HiPEC command set. These commands are stored in user-space and the kernel accesses them via an object known to it. Krueger et.al. [KLVA93] also looked at allowing applications which perform poorly with LRU to provide their own page replacement policy. Unfortunately, the number of context switches incurred in their implementation is extremely high.

Moving page replacement decisions into the application domain is reasonable for instances where it is known that a particular policy works best. Under these circumstances, the developer need not know about the underlying VM mechanisms in order to improve their application's performance; they only need to be able to implement a page replacement policy. Unfortunately, such techniques are only suitable for a small subset of applications and do not provide any benefits to the vast majority. However, their potential benefit to some should be borne in mind when considering the design of a VMM sub-system.

There are other VM techniques available to the OS developer that can be of benefit to a wide range of applications. Techniques such as compressed caching [Dou93, WKS99], remote paging [FMP<sup>+</sup>95, MD96] and prefetching [CKV93, PGG<sup>+</sup>95] have been shown to greatly improve application performance for various types of application. However, in some cases these methods can severely penalise applications. For instance compressed caching relies on many factors in order to offer any performance gain (see chapter 5). Ideally, all of these techniques would be made available to the application developer and the choice of which to utilise would be left in their hands.

The next few chapters will examine some of these VMM techniques. Chapter 3 looks at paging to a local disk and how developers have attempted to optimise the use of the disk in light of the performance penalty normally associated with it. Chapter 4 discusses the issues in paging across the network to memory on a remote machine. We conclude the first part of this dissertation in chapter 5 by looking at the possible benefits of compressed caching.

# Chapter 3

## Paging to Local Disk

*The forms of things unknown, the poet's pen  
Turns them to shapes, and gives to airy nothing  
A local habitation and a name.*

**William Shakespeare. A Midsummer Night's Dream.**

Paging to local backing store has been around since the Atlas [Fot61] system was built and still remains the most common means of virtualising physical memory. The most prevalent means of providing that backing store today is via a hard disk. However, a disk is composed of several moving parts that are operated mechanically. The time taken to move the read/write heads or spin a disk until the correct sector is underneath the appropriate head is measured in milli-seconds. In todays computing environment, a milli-second is an extremely long time: memory access is measured in nano-seconds (one million times smaller); and a 500MHz CPU can, theoretically, execute 500,000 instructions in a milli-second. Thus, servicing a page fault from a local disk can be an extremely costly operation. Despite this, every commercial operating system uses paging to the local disk in order to support virtual memory. Although researchers have come up with a range of techniques in an attempt to reduce the effect of having to fetch a page from disk, the disk is still a system bottleneck.

This chapter outlines the important characteristics of a modern disk and looks at some of the methods employed to attempt to improve its performance in a virtual memory setting.

## 3.1 The Hard Disk

There are many factors that affect the performance of a hard disk. This chapter outlines some of the most important performance factors, differentiating between those internal to the drive and those external to it. Before this, it may be useful to outline some of the physical characteristics of a disk.

A disk contains one or more platters consisting of a hard substrate coated in a magnetic material. Each platter is split into tracks, which are tightly packed concentric circles, which are further split into sectors. Sectors are usually the smallest individually addressable units. The platters are mounted on a spindle connected to a motor that spins the platters as one unit. The more tightly packed each platter, the more information that can be stored in the same space. This packing is referred to as areal density and can be increased by more tightly packing the tracks and the sectors within each track. All other things being equal, a higher areal density means more storage and faster drives.

To maximise the space on a disk platter, modern disks employ a technique called zoned bit recording (ZBR). Each track is located within a zone dependent on its physical location on the disk. Each track in a zone has the same number of sectors which increases as you move from the inner to the outer part of the disk. This allows more efficient use of the larger outer tracks. A consequence of this is that raw data transfer rates are higher when reading from the outside cylinder than from inner cylinders.<sup>1</sup> This is an important factor when considering the reported internal transfer rates of a disk as the figure given often represents reading a relatively small amount of data from the outer cylinder.

### 3.1.1 Internal Performance Factors

The internal performance factors of a modern disk refer to those operations handled solely within the drive itself. This section considers the effect of some of the more important factors on disk performance.

---

<sup>1</sup>The Quantum Fireball TM hard disk has a total of 14 zones, each consisting of 454 tracks (with zone 0 being the outermost). The data transfer rate of zone 0 is almost double that of zone 14 (92.9Mbits/s as opposed to 49.5Mbits/s).

### 3.1.1.1 Seek Time

Seek time refers to the amount of time required for the read/write heads to move between tracks. It is usually supplied as the average seek time and is generally in the range of 8 to 12ms. This average represents the amount of time to travel half the radius of the disk. In addition to seek time, figures for track-to-track time and full stroke time are sometimes supplied. The former refers to the amount of time to seek between adjacent tracks and is usually 2-4ms, but it can be as low as 1ms. The latter represents the amount of time to seek the entire width of the disk and is usually around 20ms.

### 3.1.1.2 Latency

Sometimes referred to as rotational latency, this refers to the time it takes for the platter to spin, placing the correct sector under the read/write head. As with seek time, this is usually given as an average time, i.e. the time it takes for one half revolution of the platter. Because the spin speed of the disk is not synchronised to the process of moving the heads, access time is usually used to refer to the combination of seek time plus latency. Typical modern disks rotate at around 7200 rpm, giving an average latency of 4.2ms.

### 3.1.1.3 Track Switch Time

Also called cylinder switch time, this is the time taken to move the head from one track to an adjacent track. This is an important metric as switching between adjacent tracks is a very common operation. An average cylinder on a modern disk contains less than 1MB of data which means multi-megabyte reads involve many cylinder switches.

### 3.1.1.4 Internal Data Transfer Rate

The internal data transfer rate refers to the rate at which the disk can read from the platter and transfer it to the internal drive cache ready for sending. This is different from the speed at which the data can be sent from the buffer over the interface to the system — the external data transfer rate. Although the external rate is usually higher, it is limited by the size of the buffer which, in

most modern disks, is less than 1MB. No disk can maintain its internal rate for long as this rate refers to “ideal” conditions, i.e. reading a small number of consecutive sectors from the fastest part of the disk.

The internal transfer rate can be calculated as:  $\frac{(\frac{\text{Spindle Speed}}{60} \times \text{Sectors Per Track} \times \text{Sector Size} \times 8)}{1024 \times 1024}$  However, as mentioned previously, there are less sectors in inner tracks than there are in outer ones. Thus, the transfer rate varies greatly from one zone to the other. Furthermore, sectors are actually 540 bytes in size, with 512 of these reserved for data. Manufacturers sometimes use 540 instead of 512 to calculate transfer rate, giving higher numbers — this does not refer to the transfer of “useful” data.

The internal performance factors of a disk can have an important impact on the performance of local paging. Disks rely on moving parts that must be placed in the correct position before any data transfer can take place. Once the disk heads are in place, internal data transfer is reasonably fast. Factors such as seek time and latency are more significant when transferring small amounts of data. Indeed, the time taken to transfer a page of 4KB from the disk is entirely dominated by access time. Such small transfers are not as affected by track switch time as this is less likely to occur during small transfers.

### 3.1.2 External Performance Factors

External performance factors relate to how the hard drive interacts with the rest of the system. A key factor when attempting to obtain the most from a hard disk is the external transfer rate. The external rate is an electronic operation as opposed to a mechanical one and, as such, is usually much faster than the internal rate. The external transfer rate is dictated primarily by the interface used and the mode the interface operates in. Most modern disks used in PCs are typically enhanced integrated drive electronics (EIDE) or advanced technology attachment-2 (ATA-2) drives running with programmed input/output (PIO) mode 4 or multiword DMA mode 2, giving a theoretical maximum of 16.6MB/s. The newest drives support Ultra ATA, giving an external transfer rate of up to 33.3MB/s.

The external data transfer rate can be a somewhat confusing figure. The external rate relies on data being in the drive’s buffer which means that it can only possibly apply for the time taken to empty the buffer and does not give a real indication of overall performance.



Disk caching allows limited separation of the slower internal transfer rate from the external transfer rate. A larger cache will improve performance by reducing the number of physical seeks and transfers from the platters themselves. Clever caching algorithms can also increase this performance. As well as the drive's internal cache, some operating systems reserve an area of physical memory to act as a first level cache before that of the drive itself.

Other system factors include the CPU, the motherboard, I/O bus and chipset. Typically, a faster CPU will run disk benchmarks quicker than a slower one, especially if using an interface that relies on CPU intervention, such as PIO. Similarly, the faster the system bus, the faster data can be transferred from the drive to memory.

The external performance factors do not have a significant impact on paging to/from the local disk. Writing to the disk's cache is extremely fast for small amounts of data. Similarly, when reading from the disk, the significant factor is not transferring the data over the I/O bus or from the disk's cache, but waiting for the disk head to get into the correct position to begin the transfer.

### 3.1.3 The Effect of Performance Factors on Paging

Regardless of how the operating system supports paging to a local disk, the key factor about paging is that the units sent to the disk are very small. A system page for the Intel ix86 architecture is only 4KB in size — only eight disk sectors. This means that the data transfers are relatively small, even on operating systems that perform writes of many (usually up to 16) pages at once, and the access time is the dominating factor. If the disk is hardly used and the disk arm can be kept within the swap range on the disk, the most significant costs are likely to be rotational latency and, possibly, track switch time. These costs can in turn be affected by where the swap range is positioned on the disk — there are more sectors per track on the outer cylinders.

## 3.2 Beating the Bottleneck

Improving disk performance tends to come in two forms: changes in disk architecture, such as RAID [PGK88] and disk-caching disk (DCD) [HY96], and changes in how the disk is used, such as the log structured file system (LFS) [RO92] and data prefetching [HV84]. The effects of the disk architecture are not considered here as the virtual memory system should be largely

independent of this. Instead, this section focuses on techniques that attempt to use the disk in such a way as to minimise the effect of the latency, in particular, memory “cleaning” and page prefetching.

### 3.2.1 Paging Daemons

One way of reducing the time required to service a page fault is to always have a frame available to map the faulting page. A paging daemon is a special OS process that is activated periodically, or on the occurrence of a special condition, and cleans modified pages in memory by flushing them to disk. This increases the likelihood that a faulting page can be mapped without the need to write a page to disk. This process is sometimes referred to as cleaning. The aim is to try and perform the disk writes during “quiet” time, thus reducing the chance of having to perform a write when faulting in a page.

Although the paging daemon can increase the overall number of I/O operations — pages can be cleaned several times before being finally evicted from memory — it can still greatly improve system performance.

The Linux operating system [BBD<sup>+</sup>98] has a thread *kswapd* which monitors the availability of free pages in the Linux Kernel. If the number of free pages drops below a certain threshold, the thread attempts to free up memory via calls to `try_to_free_pages()`.

Most modern operating systems seldom write single pages to the local disk. Instead, they tend to buffer a group of pages and write them all at once. This reduces the impact of the access time.

### 3.2.2 Page Prefetching

Instead of performing page-in operations as they are required (known as demand paging) it would benefit the system greatly if it could fetch pages that are likely to be required in the very near future so that they are already resident in memory when they are accessed. The trick of this is knowing which pages will be required when. Fetching pages that are expelled before being used takes up I/O bandwidth and memory that could be used more efficaciously.

There are many techniques available which attempt to guess which pages will be used again in the near future. For instance, one can use lessons learned from data compression techniques

which, in essence, attempt to do the same thing as prefetching: they attempt to predict the future. Vitter and Krishnan [VK96] describe a prefetching algorithm based on a character-based version  $\mathcal{E}$  of the Ziv-Lempel algorithm for data compression. Pages are encoded in the same manner as substrings and a parse tree is constructed. For example, for an alphabet  $\{a,b\}$ , a page request sequence “aaaababaabbbabaa · · ·” results in the substrings  $(a)(aa)(ab)(aba)(abb)(b)(abaa) \cdot \cdot \cdot$ .

In the character-based version  $\mathcal{E}$  of the Ziv-Lempel encoder, a probabilistic model (or parse tree) is constructed for each substring when the previous substring ends. Figure 3.1 shows the parse tree at the start of the seventh substring. There are five previous substrings beginning with an “a” and one beginning with a “b”. The page “a” is therefore assigned a probability  $5/6$  at the root and the page “b”  $1/6$  at the root. Of the five substrings beginning with an “a”, one begins “aa” and three begin with “ab”, resulting in the respective probabilities  $1/5$  and  $3/5$ , and so on.

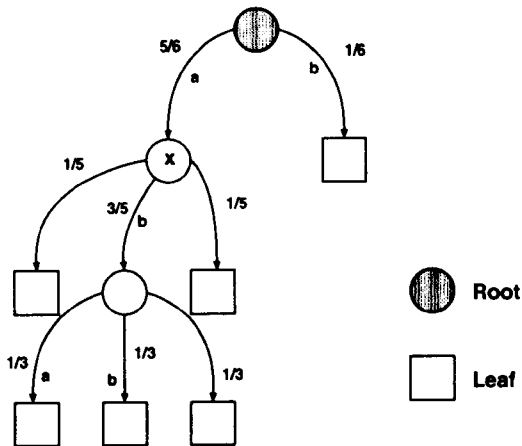


Figure 3.1: Parse Tree Constructed by the Character-Based Encoder  $\mathcal{E}$ .

Before each page request, the prefetcher  $\mathcal{P}$  prefetches the pages with the top  $k$  estimated probabilities as specified by the transitions out of its current node. On the fetching of the actual page requested,  $\mathcal{P}$  resets its current node by walking down the transition labelled by that page and gets ready to prefetch again. When  $\mathcal{P}$  reaches a leaf, it fetches  $k$  pages at random.

Curewitz et. al. [CKV93] compared Vitter and Krishnan’s prefetcher with two others, reasoning that although ZL was theoretically optimal, convergence on optimality was slow. This led to them adapting the prediction-by-partial-match (PPM) data compressors. They found that all three prefetchers improved page fault rate and that the relative performance of each algorithm paralleled their relative performance for data compression.

Patterson et. al. [PGG<sup>+</sup>95, TPG97] show that allowing applications to provide hints to the system about what data is likely to be accessed in the near future can speed up application performance by up to 85%. Their system, TIP (and later TIPTOE), replaced the unified buffer cache (UBC) of a Digital 3000/500 running OSF/1 with an array of ten disks and showed that hints always increased throughput, regardless of how many applications were running.

The notion of allowing the application to provide prefetching information was extended by Cox and Ellsworth [CE97] to allow applications to control segmentation and demand paging. This work focused on applications that could not fit in main memory, and maybe not on the local disk, in particular, the visualisation of computational fluid dynamics (CFD) where input data sets can exceed 100 Gigabytes. The authors attempted to move as much of the decision making process into the application space as possible and found significant improvements in system performance, despite being limited by the platform, an R10000 running IRIX. They showed that the operating system did not have enough information to make the correct management decisions on the application's behalf.

### 3.3 Discussion

Paging to the local disk is still the preferred method of providing backing store for virtual memory. Although pure demand paging is extremely costly in terms of missed CPU cycles, researchers have come up with many techniques to reduce the impact the disk has on application performance. Unfortunately, the disk is being somewhat left behind in terms of latency and bandwidth. Modern networks have very low latency and high bandwidth and these are improving at a greater rate than their disk counterparts. Improvements in disk "speed" are also lagging far behind improvements in memory and CPU speed. Couple this with the fact that disk bandwidth is improving at a greater rate than disk latency and the impact of latency continues to grow. The effect is that, as the disparity between latency and bandwidth continues, bigger and bigger transfers are required in order to reduce the impact of latency on disk performance.

The effect of current trends in relative performance of CPU, memory, networks and disks should not entirely rule out the use of a local disk. The local disk is still used for paging because it has something to offer. There is no need to worry about fault tolerance when paging; an issue which is important if paging across the network. Pages can be buffered to perform writes in larger blocks and can be prefetched to reduce the likelihood of a page not being in memory when

it is required. Although these techniques can greatly improve performance, their improvement relative to other techniques is dependent on each application's memory access patterns and the nature of their in-memory data.

We have described the internal structure of a hard disk and explained why it is not ideal for handling the small reads and writes normally associated with paging. We have expanded on this by showing some ways in which the disk is used in an attempt to reduce the effect of its inherent inefficiencies. The next chapter provides an alternative to paging to the local disk. Remote paging involves using the memory of a remote host as backing store and, like local paging, there are many issues that can have an impact on its performance.

# Chapter 4

## Remote Paging

*Be neither too remote nor too familiar.*

**Charles, Prince of Wales.**

Remote paging involves using free memory on remote hosts to service page faults on behalf of local clients. There are two main flavours: the first involves a central server, or servers, with large (in the range of a few gigabytes) amounts of physical memory that all participants use to hold swapped-out pages; the second involves the use of free memory on workstations within a distributed system.

Table 4.1 (baseline and improvement rate taken from [Dah96]) shows the relative changes in the performance of disk, network, memory and CPU, analysed over a fifteen year period from 1980 to 1995. Using these improvement rates, we have projected the figures to the years 2000 and 2005. The projections for the year 2000 are found to be in accordance with the actual technology levels observed to be available, suggesting that the trend is continuing as expected.

A key factor with regard to paging across the network versus paging to the local disk is the difference between network latency and disk latency. The time taken to fetch 8KB of data from the disk is dominated by the latency, whereas the preponderant factor for fetching the same amount of data across the network is the bandwidth. Also, because the disk bandwidth increases faster than the disk latency, increasingly large transfers are required in order to maintain a certain level of efficiency. If we fetch a page from memory on a remote machine, the fault can be resolved long before a local disk head is in the proper position for reading.

Hardware	1995 Baseline	Yearly Improvement Rate	2000 Projected	2005 Projected
Disk Latency	12ms	10%	7ms	4.2ms
Disk Bandwidth	6-9MB/s	20%	15-22MB/s	37-57MB/s
Network Latency	1ms	20%	0.33ms	0.1ms
Network Bandwidth	20MB/s	45%	128MB/s	821MB/s
Processor Performance	100 SPECInt92	55%	N/A	N/A
Memory Bandwidth	30-70MB/s	40%	161-448MB/s	867-2024MB/s

Table 4.1: Performance trends in Disk, Memory and CPU. Note that the SPECInt performance is highly architecture-dependent and has since been superseded by SPECInt95 and SPECInt2000.

Recent years have seen a large increase in the amount of physical memory available in desktop computers. The fall in memory prices was drastic in the period 1989 to 1991, dropping by over 50% per year. However, the decline in cost has flattened since around 1992. Thus, the idea of simply continually expanding the amount of physical memory to cope with larger and larger applications will eventually run into problems. Furthermore, memory prices are cheap only in commodity sizes (32/64/128 MB). For instance a 512 MB DIMM costs over six times as much as four 128 MB DIMMs [AS99]. Consequently, virtual memory is expected to be as important a component of future operating systems as it is for today's. However, an important aspect of increasing memory sizes that is extremely interesting from a virtual memory standpoint is the availability of free memory on workstations within a local area network. Workstations running large memory consuming applications could benefit significantly from unused memory on workstations where the memory consumption is much more moderate. This availability of memory is discussed in detail in section 4.2.2. Before that, some background on how other researchers have attempted to utilise this free memory to speed up application performance is presented.

4.1 Background

Researchers have taken different approaches to the use of memory on remote hosts. Some have seen it as a way for improving local performance by extending the amount of physical memory available to processes, i.e using it in the same way as local memory as opposed to backing

store, while others have considered it the solution to the disk latency problem. Some of these approaches are now discussed with a view to highlighting the positive aspects and the possible limitations.

**Remote Paging Protocol:** Comer and Griffioen [CG90] describe a remote memory model that uses dedicated memory servers on which clients can allocate memory. The remote memory servers could extend their capacity by utilising a local disk. This would be transparent to clients who would not know where their data was coming from. The remote memory model was designed as a framework primarily for distributed shared virtual memory (DSVM). When client machines exhaust their physical memory, they can move parts of their address space to the remote server and retrieve pieces as needed. It was never intended to solve, or even reduce, the effects of the disk latency problem. However, the authors did suggest a machine-independent communication protocol that could be utilised by a remote paging system.

**Dodo:** The Dodo system [KAS98] provides a method of using remote memory as an intermediate cache between memory and the local disk.<sup>1</sup> The Dodo system is provided via application level libraries and does not change any of the OS code. This makes the system more portable but also limits the implementation options. The Dodo system requires applications to use it explicitly but does provide a region-management library to simplify this process. However, the region-management library is only useful for applications with well-defined memory access patterns, others still have to use the Dodo system explicitly. This means that they have to manage the tracking of objects in remote caches themselves.

When applications wish to allocate a remote object, they contact a central memory manager with the request. The manager finds a suitable host and contacts it with the request. If successful, the host and a memory region identifier are returned to the client. A request for data from the remote cache can fail under two circumstances: the remote server crashes; or the remote server decides to return memory to the local machine (this occurs when the machine ceases to be idle<sup>2</sup>).

There are several problems with the Dodo system. Firstly, the central memory server not only represents a single point of failure, it also represents a system-wide bottleneck — all remote

---

<sup>1</sup> Note this is not a remote paging system as such but a way of speeding up the performance of applications where the memory access patterns are well understood.

<sup>2</sup> The authors define being idle as a combination of no keyboard or mouse events and a CPU load of less than 0.3 over the last five minutes.



allocations go through a single server. Secondly, the system offers no deterministic level of performance. When a remote host ceases to be idle, it evicts all memory held on behalf of remote clients to the local OS without informing them. Thirdly, because the authors chose portability over performance, their system does not have enough control over its environment to provide real flexibility. If the designers were to alter the VMM system, they could provide a much more flexible system where local and remote commitments could be more easily balanced. For instance, it appears that the system relies on the fact that the remote machine is idle in order to prevent the cache from being paged to that host's local disk. By altering the VM system, it would be possible to balance memory for local processes with memory for remote caching.

Although Dodo is not a remote paging system, the idea of allowing regions to be cached in the memory of remote hosts has some merit. Unfortunately, the system is not flexible enough to take full advantage of its potential.

**Sprite:** Nelson ([Nel86]) proposed having a central server for paging in the Sprite operating system. In Sprite, paging occurs over the network to files and there is no notion of a swap partition. Although this system performed remote paging, it was not a remote paging system *per se*. The main motivation behind this method was not to increase performance, indeed pages could find themselves on the server's disk causing a swap to be more expensive than using a local disk, but to eliminate the need for a dedicated local swap partition.

**Remote Paging on Mobile Computers:** Schilit and Duchamp [SD91] looked at remote paging for mobile computers and concluded, somewhat ironically, that "portable computers need neither a hard disk nor an excessive amount of RAM, provided that they will operate in environments in which remote storage is plentiful". This is somewhat contrary to the primary advantage of mobile computers which is that they can operate for large periods of time independently of a network.

Schilit and Duchamp implemented their adaptive paging scheme on the Mach  $\mu$ -kernel and they define three software components: a *paging server* runs on each machine and is responsible for the storage of pages on behalf of remote clients; one *broker* runs on the network and is used to put clients in touch with suitable servers; and a *service organiser* runs on each host and is used to liaise with the broker and the server for remote resources. Although clients can use several

servers at once, preference is given to utilising only one server to reduce the chance of a crashed host holding a client's pages.

Schilit and Duchamp provide a framework for a flexible and efficient remote paging system. Unfortunately, a combination of environmental factors and poor design decisions have managed to rob it of much of its promise. Due to Mach 2.5 not allowing user-level control of physical memory management, pages sent to a remote server can end up on that machine's local disk. Also, the authors decision to use RPC as the protocol for requesting pages from remote servers has incurred a large overhead, most of which can be attributed to the underlying use of the transfer control protocol (TCP). Consequently, it takes 45ms to perform the reading of a random page from the memory of a remote host.<sup>3</sup>

By using RPC, and its dependence on TCP, Schilit and Duchamp do not need to worry about reliable and out-of-order delivery. However, the problem of a server crashing is not addressed, though a re-election scheme for the broker is described, and so applications do not have the ability to make progress in light of their paging server crashing or becoming unreachable.

**Markatos and Dramatinos** [MD96] describe a remote paging system which utilises free memory on workstations within a distributed environment. The system is implemented as an OSF/1 device driver that replaces the local disk device driver. This allows the system to be implemented without making changes to the operating system itself. While providing the remote pager in this way means that it can be incorporated into OSF/1 machines easily, it also limits the flexibility of the system and deprives it of access to the virtual memory management (VMM) implementation that could enhance its performance. A consequence of this is that when a page is transferred to a remote host, it becomes a participant in the paging scheme employed by the operating system on that host. This means that an evicted page could find itself on the disk of a remote machine, thereby increasing the time it takes to service a page fault.

Markatos and Dramatinos also outline a scheme for providing fault-tolerance, known as parity logging, in a remote paging environment but, as discussed in section 4.2.4.4, their solution has a very high space overhead coupled with an inability to scale. Furthermore, the possibility of pages being distributed across many disks on the network further degrades the potential performance of their scheme. For a full discussion of this see section 4.2.4.

---

<sup>3</sup>Hosts are connected via a 10Mb Ethernet.

**GMS:** Feeley et.al. [FMP<sup>+</sup>95] propose a much more flexible solution but for a much more specialised environment, described as the global memory system (GMS).<sup>4</sup> GMS uses a global LRU replacement algorithm within a tightly coupled cluster of homogeneous workstations, connected to a MYRINET network, in an attempt to use as much physical memory within the cluster as possible. It differentiates between local and global pages: those belonging to local processes and those belonging to processes on remote hosts. GMS is implemented as part of the virtual memory management system within the OSF/1 kernel. Unlike the system described by Markatos and Dramatinos in [MD96], pages never end up on the disk of a remote host. However, as discussed below, the runtime overheads in GMS are high. While these overheads may not be as severe in a small cluster of machines connected via a very fast low latency network, they would have a more pronounced effect in a general LAN environment.

The idea behind GMS is to provide a global LRU replacement strategy that forces local pages on idle machines to gradually drift onto the local disk and global pages to gradually fill the frames left. To manage the global aging of pages, time is split into epochs of a maximum duration  $T$ , where a maximum number of replacements  $M$  are allowed in that epoch (the current implementation has the epoch between 5 and 10 seconds). At the start of an epoch, every node sends a summary of the ages of all its local and global pages to a designated initiator node. The initiator then computes a weight  $w_i$  for each node  $i$  such that, for the  $M$  oldest pages in the network,  $w_i$  reside in node  $i$ 's memory. The initiator also determines the minimum age, *MinAge*, that will be replaced from the cluster in the new epoch. It then sends all the weights and the value of *MinAge* to all the nodes in the cluster and selects the node with the most idle pages to be the initiator for the following epoch. Thus, when a node looks for another node to take a page, it selects node  $i$  where the probability of selecting node  $i$  is dependent on the weight  $w_i$ .

This generates a large amount of traffic quite frequently: every node sends to the epoch server and the epoch server sends to every node at the beginning of each new epoch. This is not a problem on a very high speed, low latency network, where the only traffic is generated by the GMS, but if porting the system to a general purpose LAN this may prove prohibitive. There is also the problem of joining and leaving a cluster at any one time. The scheme relies on a *master* node coordinating the redistribution of data structures to all the nodes in the cluster every time a node joins or leaves the cluster. The authors admit that this, along with the epoch server, represents a problem if failure should occur and recommend a re-election scheme to solve it.

---

<sup>4</sup>It should be noted that this is not a DSVM system, as the name may suggest, but is in fact a remote paging scheme.

In its favour, GMS is adaptive to the load on the network. The values  $T$ ,  $M$  and  $MinAge$  vary according to the behaviour of the system during the last epoch. Thus, if the number of old pages becomes too small,  $MinAge$  is set to zero and all nodes resort to using the local disk. This scheme exploits the dynamic load and ensures that pages never end up on the disk of remote hosts. However, while the average page-fault servicing time should out-perform paging to disk, given a certain load, it is not known where a page will be fetched from on its request. The system contacts the remote host it expects to have the page and, if the page is not there, resorts to reading from the local disk. This also implies that the speed of transferring a page from a local disk does not represent the upper-bound. Indeed, having to resort to the local disk is almost comparable to fetching from the disk on a remote host.

There are some limiting factors to note about both the GMS scheme and the scheme presented by Markatos and Dramatinos. Firstly, neither of them take account of out-of-order delivery or dropped packets. Feeley et.al. state that no dropped packets were ever observed during their tests but do not mention out-of-order delivery; Markatos and Dramatinos presented a scheme for “fault-tolerant” remote paging but do not mention reliable delivery. Secondly, neither scheme supports remote paging in an architecturally heterogeneous environment. Also, neither scheme offers deterministic performance. A page request to a remote host in GMS may fail due to that page being evicted, forcing the request to be forwarded to the local disk; a page request in the scheme proposed by Markatos and Dramatinos could be served by the remote disk.

**Remote Paging in a Heterogeneous Environment:** The work described by Markatos and Dramatinos was extended by Flouris and Markatos [FM98] to include adaptive parity caching, as the replacement to parity logging, to provide fault tolerance. The authors also claimed to support remote paging in a heterogeneous environment. However, after stating that all machines in the network of workstations (NOW) ran the Network RamDisk (NRD) server, the authors then described the operating system running NRD clients and that running servers separately. This suggested that, although it was possible to page from a Linux host to a Solaris host, it was not possible to page from a Solaris to a Linux host. The authors further compound the confusion by stating that NRD clients go through a “configuration phase” on start-up where they determine what servers they will use. How a client decides which servers to use and where it gets the necessary information was not described.

**Sub-Page Paging** Jamrozik et.al. [JFV<sup>+</sup>96] utilised the GMS system to investigate the effect of sub-page size on paging across the network.<sup>5</sup> They suggest that although large pages are well suited to TLBs and disk reads/writes, they are ill-suited to reading/writing across a network. By fetching a portion of the page, it is possible to restart the application and continue to fetch the rest of the page asynchronously instead of waiting until the entire page arrives. This means that sub-page accesses have to be trapped somehow in order to prevent accessing a sub-page yet to be fetched. This is achieved by marking a page as invalid until all sub-pages are fetched; each access to that page is trapped and the appropriate bits are checked to determine whether the sub-page is currently in memory. If the sub-page is in memory then the read or write access is carried out. Although this process is carried out in PAL code it still means that, until the entire page is resident in memory, every access to a sub-page, even a resident sub-page, causes a page fault. Despite this, the authors report that the optimal sub-page size is 1 or 2KB. These sub-page sizes give enough computational overlap to allow the application to make progress while the rest of the page is being fetched and results show a performance improvement of between 8 and 40% for the applications tested.

**PGMS:** Voelker et.al. [VAK<sup>+</sup>98] describe an extension to the GMS scheme that incorporates prefetching and caching (PGMS). While this was reported to improve on GMS, it still shares the network overhead problems of GMS. Indeed, the network traffic costs incurred by GMS are extended by PGMS to incorporate prefetch requests. While this is not a problem in the tightly coupled environment in which the system runs, it does not represent a general purpose solution.

## 4.2 Issues in Remote Paging

In this section we discuss some of the issues in remote paging. We begin in sections 4.2.1 and 4.2.2 by considering the viability of remote paging. We continue in section 4.2.3 by looking at the configuration method that decides which hosts may take part and the role which they wish to fill, i.e. whether they act as a client or a server. We then discuss fault-tolerance and the effect it can have on the remote paging system in section 4.2.4. The possibility of splitting pages into sub-pages is considered in section 4.2.5 and the acquisition of physical memory in

---

<sup>5</sup>Their experiments were carried out on machines running OSF (where a page is 8KB) connected by a 155Mb/sec ATM network.

4.2.6. In section 4.2.7 we discuss the issue of revocation and conclude in section 4.2.8 with quality of service considerations. Some sub-sections here refer to measurements obtained from applications which are more fully described in appendix G. Note that, because these applications were run with specific guarantees from the system, the results varied very little over many runs (typically within 1%). See section 12.2 for a fuller discussion on the variation of results.

### 4.2.1 Network Bandwidth

For paging across the network to be a viable alternative to paging to a local disk, there must be network bandwidth relatively comparable to the external transfer rate of the local disk. The bandwidths need not be directly comparable, as the network latency is not as significant a factor for remote paging as the disk latency is for local paging. Thus, combining lower bandwidth with lower latency can give better performance than higher bandwidth with higher latency.

Another factor affecting the bandwidth is the relative location of the client and the server. If both hosts are connected via the same switch, for example, then the actual bandwidth between the two is very close to the wire speed. However, if the hosts are connected to different switches, the actual attainable bandwidth may be significantly reduced.

A significant difference between how data is placed on the disk and how it is sent across the network is that there is a maximum transfer unit (MTU) associated with the latter. This means that data being sent across the network has to be broken into multiples of the MTU and reassembled at the other end. In addition to this, there is no way of knowing if the data definitely arrived without the recipient acknowledging the receipt of it. This can effectively reduce the data throughput.

One possible way of reducing the network transfer time is to compress the data before it is sent. Figure 4.1 shows the relative times for writing to a local disk and writing across a 100Mb Ethernet with and without compression.<sup>6</sup> Writing across the network also includes the time taken for the client to receive an acknowledgement for the arrival of the page. As can be seen, the disk and 100Mb Ethernet are comparable in performance. However, compressing the pages before transmitting them across the network improves performance significantly (approximately twice as fast, depending on compression ratio). This is because the bandwidth is the limiting factor for the network. The limiting factor for the disk, however, is the latency. Thus, as would be

---

<sup>6</sup>One hundred pages were written consecutively and an average taken.

expected, there is no performance benefit in compressing pages before writing them to the local disk.

Compressing pages before transmitting them to the server has the additional advantage of saving the memory required at the server. However, this must be traded against more complex memory management. For instance, the first time a page is stored at the server it takes up a certain amount of memory. However, the next time it is stored, it may be a different size. This means that the old space must be freed and new space found.

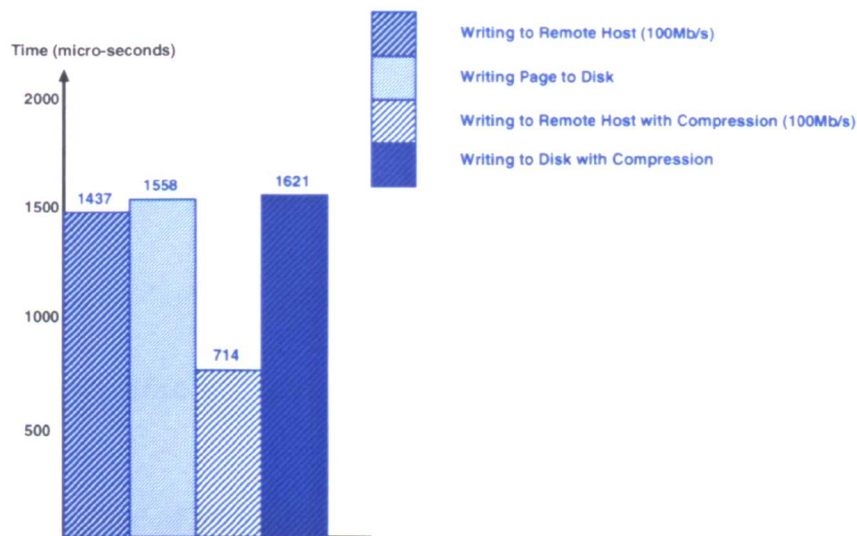


Figure 4.1: Remote versus Local Paging.

An effect associated with network bandwidth is CPU bandwidth at the server. Sending pages across the network to a remote host requires that host to store the page and perform some book-keeping. This requires CPU intervention.

The relative speeds of the processor and the interconnect can have a significant impact on performance. Figure 4.2 shows the results of running the *compress* application with varying restrictions. The machines were connected via a 100Mb switched Ethernet network. Each machine had a Pentium PIII processor running at 450MHz. Each configuration was run with 10% of local CPU bandwidth. Reducing the network access by a factor of 10 results in a fivefold increase in the time taken for the application to complete. Reducing the CPU bandwidth by a similar factor causes only a twofold increase. This is because the CPU can copy the data from the receive

buffer to memory quicker than the network can transport the data. However, as these relative speeds converge, the more delicate the balance between network and CPU bandwidth becomes.

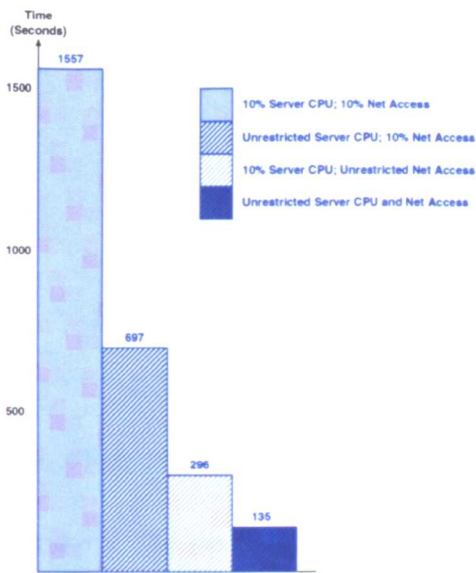


Figure 4.2: Effect of Limiting Server CPU Bandwidth and Network Access.

4.2.2 Availability of Physical Memory

For remote paging to be viable, there must be enough physical memory available on remote hosts that can be utilised by processes running on machines with limited resources. Figure 4.3 shows the availability of free memory on a total of 113 i386 machines running Linux (Redhat 5.2 and Redhat 6.0), over a one month period.<sup>7</sup> The measurements were taken every 15 minutes between the hours of 8.00 a.m and 6.00 p.m on Mondays to Fridays only. The vast majority of these machines reside in labs used by Computing Science students at the University of Glasgow.

As can be seen, there were gigabytes of free memory in total at any given time over that one month period. This is memory that can be utilised to improve the paging performance of applications running on heavily loaded hosts.<sup>8</sup> For instance, figure 4.4 shows free memory and swap space usage over the period of one day, between the hours of 8.00 a.m and 6.00p.m, for one of

<sup>7</sup>Note that this does not include free memory that Linux reserves for buffers or caches. If this were to be taken into account then the figures would be significantly higher.

<sup>8</sup>The vast majority of machines had 256MB of physical memory but some had 64MB or less.



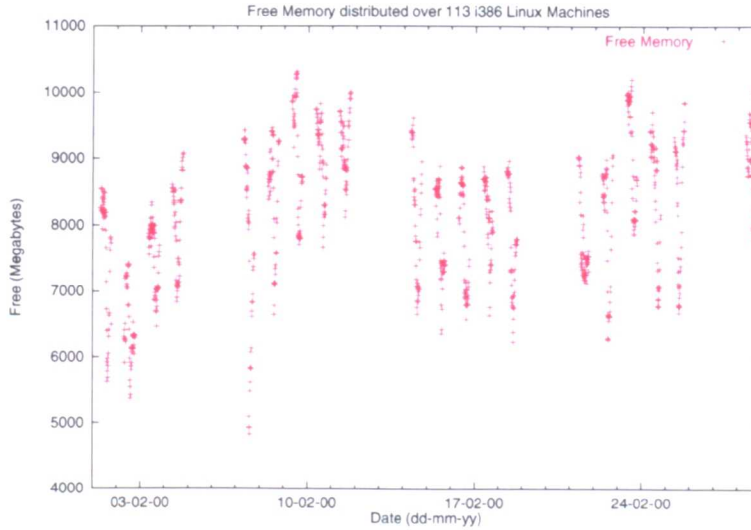


Figure 4.3: Free Memory on i386 Linux Machines

the Linux machines mentioned above. The machine is a P133 with 64MB of RAM and had very little free memory, relying on swap space to help meet the demands placed on it.

Acharya and Setia [AS99] looked at the availability of “idle” memory in workstation clusters with a view to examining the viability of the Dodo system described earlier. They focused on two clusters of machines: cluster A consisted of 29 Sun workstations running Solaris 2.5.1/2.6 and cluster B consisted of 23 Sun workstations running Solaris 2.6. Most machines in cluster A had 128 MB or more of physical memory while most in cluster B had 64 MB or less. The authors concluded that, for machines with 64 MB or more, half of the physical memory could be expected to be free for a period of 12 minutes at a time and a quarter could be expected to be free for up to 30 minutes at a time. This suggests that harvesting free memory on remote hosts is a viable proposition. It also suggests that by harvesting smaller chunks of physical memory, clients can use that memory for longer periods.

### 4.2.3 Dynamic Versus Static Configuration

There are two approaches to configuring a remote paging system. One approach is to allow hosts to move from one state (i.e., client or server) to another as the availability of resources changes. The other involves each node taking on a particular role and maintaining that role for the entire time that it participates in the scheme.

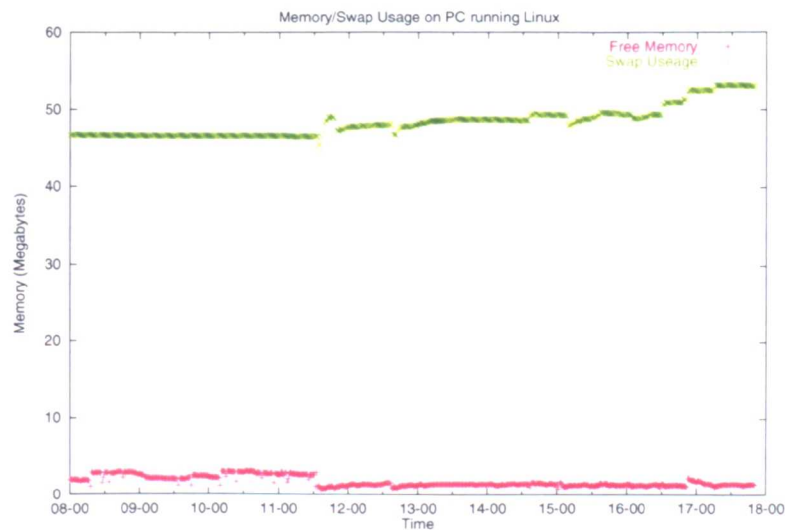


Figure 4.4: Memory Consumption on a PC running Linux

A static system where there are pre-designated servers is very restrictive: it does not take into account the current load on a given host at any given time. For instance, a client may be sitting idle with a large amount of physical memory not being used while the servers have their physical memory fully committed. Under such conditions it would be more beneficial to have that client register itself as a server and make its resources available to other hosts. In favour of the static approach, it is much easier to implement than a dynamic system where hosts evolve over time according to current commitments. For example, Markatos and Dramatinos hold the remote paging servers in a file that clients can read.

The management of a dynamic system is more challenging as the current role of any host depends on its present commitments. Thus, we must find a way of allowing those currently acting as clients to find those currently acting as servers. However, a dynamic system does offer greater flexibility and takes into account the nature of overall load on any given distributed system, i.e., that at any given time, there may be lightly loaded hosts and hosts which are heavily loaded. This dynamism is something we wish to exploit with remote paging.

4.2.4 Fault Tolerance

When considering the use of remote hosts as backing store we introduce added complexity to our system and we also extend our reliance on local resources to include non-local resources. In

doing so, we introduce more potential for error: pages may be lost or corrupted during transit to/from the memory server; servers may crash, causing loss of data held on a client's behalf; and a client may crash and not free the resources held by servers on its behalf. A remote paging system must define the level of tolerance to such faults.

Comer and Griffioen [CG90] describe a protocol for reliable delivery of pages between machines in a heterogeneous environment. The Remote Paging Protocol consists of two layers: the Xinu Paging Protocol (XPP) and the Negative Acknowledgement Fragmentation Protocol (NAFP). The XPP layer is responsible for the reliable delivery of *pages*. On sending a page to a remote host, the XPP layer expects a positive acknowledgement (ACK) of the reliable delivery of that page. If a positive ACK is not received after a designated timeout period, the page is re-sent. If the page happens to be sent twice, the server simply overwrites the first with the second. Because pages may be larger than the maximum transfer unit (MTU) of the network, the NAFP layer fragments the pages into units small enough for transfer and reassembles them at the other end. The NAFP layer does not send positive ACKs; it will send a negative ACK if a portion of a page is not received. Thus, under normal circumstances, with no network errors, NAFP incurs no additional overhead.

Coping with server crashes is a more difficult proposition than reliable delivery of messages. First of all, there must be a way of detecting when a server may have crashed; secondly, invariably, some form of redundancy must be employed to cope with the loss of data. Detecting the possibility of a server crash, or of it becoming unreachable, can be solved by applying some form of upper limit to the resending of data. If a client repeatedly times-out on the sending or requesting of a page, it may decide that a server has become unreachable, or may have crashed. Of course, another way of detecting a server crash is to have the servers send "I'm alive" messages to the clients on behalf of whom it is holding data. The remote paging client can then note the period between such messages and use this data to determine a server becoming unreachable. What the client does on such an occasion depends on the level of fault tolerance being employed (it may be providing no fault tolerance, decide that it cannot make any progress and simply halt).

Some possible solutions to ensure fault tolerance in the case of a single server crashing are discussed in the following sections. Note that although multiple server crashes are possible, the likelihood of such an occurrence is considered extremely small and if such an event must be dealt with then mirroring pages to the local disk, if available, is recommended. Should multiple hosts crash at the same time, the most likely reason would be widespread failure due to a power cut

and the application would most likely fail anyway.

#### 4.2.4.1 Local Mirroring

A simple solution to providing fault tolerance is to mirror dirty pages to the local disk as they are paged out to a remote host. If we use a buffer for page writes, we can perform writes asynchronously (see figure 4.5). This means, conceptually at least, that a disk write is free; reads are from memory on a remote host and the cost consists of a network request plus a network transfer and perhaps a memory copy at both ends. Only when a host is suspected of being unreachable does the VM system have to resort to using the local disk. This of course requires extra physical memory. However, the buffer need not be large. Consider, for simplicity, a single-threaded application performing local mirroring. If, on a page out, the write to “disk” is performed first, the page is copied to the buffer. Then the page is sent to a remote server and the application blocks. This allows the disk thread to empty some of the buffer without seriously affecting the application. Of course in a multi-threaded application the effects may be slightly more significant as the disk thread would then be competing with other application threads.

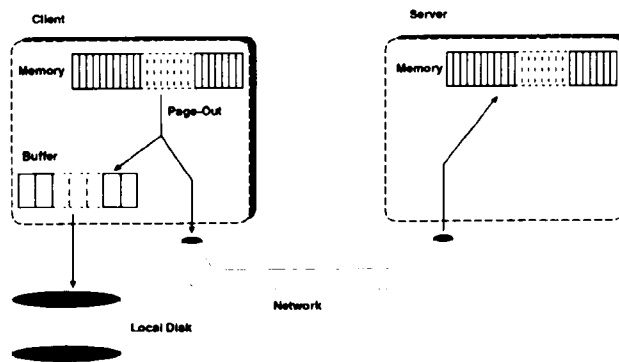


Figure 4.5: Mirroring Pages to Disk on Page Out

#### 4.2.4.2 Remote Mirroring

Remote mirroring involves using two or more hosts and sending each page to both hosts. Thus, if one host should crash, we could simply switch to using the other<sup>9</sup>.

<sup>9</sup>Of course we now have to mirror all pages on the new server to another host to ensure that we continue to have fault tolerance, although we may be able to do this on the fly.

This method incurs a potentially large runtime overhead: every page-out operation requires two page transfers (plus 2 ACKs) and, perhaps most significantly, we require double the amount of memory. The page transfer cost could be resolved by using a multicast scheme (for instance, [KKT96] describe a group based multicast scheme that is both reliable and scalable and offers almost comparable performance to a unicast scheme and [BE98] describe a scalable scheme for on-to-many multicasting using polling). It may also be possible to reduce the memory overhead by compressing the pages at the server, or before transfer. If the pages were compressed at the server and the mirror, then the client would have to pay the decompression costs on each page-in. However, if the pages are only compressed at the mirror, the space overhead could be reduced without, in the normal case, paying any costs: the client does not need to wait until the page is compressed before continuing and we never page-in from the mirror so there is no decompression cost. Only if the server crashes would any decompression cost have to be paid.

#### 4.2.4.3 Parity Scheme

Another method of providing fault tolerance in the event of a server crash is to employ a simple parity scheme based on that used in RAID [PGK88]. Consider a scheme where there are  $S$  servers, each having  $N$  pages. Page  $(i,j)$  refers to page  $j$  on server  $i$ . A parity set consists of the  $j$ th page on each server. Parity page  $j$  is formed by taking the XOR of all the  $j$ th pages in all servers. Thus, to recreate any page, on a server crash, we simply reconstruct it by XORing all the other pages in its parity set. A page swap now consists of two stages: firstly, the client sends the swapped page to the server, which computes the XOR of the old and the new page; secondly, the server sends the resulting page to the parity server which XORs it with the old parity, forming the new parity. Although we have reduced the space overhead to  $\frac{1}{S} * N$ , we still now have two page transfers for every swapped out page. Also, we cannot afford to overwrite a page in memory until we are sure that the parity page has made it to the parity server. In addition to the page transfer overhead, the parity server is now a performance bottleneck with all remote paging servers sending it a copy of each page they receive.

#### 4.2.4.4 Parity Logging

Parity logging is a scheme devised by Markatos and Dramatinos [MD96] to improve the performance of the simple parity scheme described above. This scheme, like the RAID method,

requires  $\frac{1}{S}$  transfers for  $S$  servers. However, instead of the XORing being carried out by a server, the client XORs each page with a page-size buffer (initially of all zeros) and sends the buffer to a server on a round-robin basis. Because the XORing is carried out by the client, it does not have to wait for the page to reach the server before replacing the page in memory. On a server crash, the lost pages can be recreated from the appropriate pages of a parity group.

The main problem with this scheme occurs when a page is re-paged out. When this happens, the page is marked in the old parity group containing it as inactive and the page becomes part of a new parity group (as well as being required by the old one in case of failure). This entails a memory overhead that is difficult to predict. Furthermore, if available physical memory runs low, the authors suggest garbage collecting inactive pages, thus freeing parity sets by combining the active pages with new ones. Distributed garbage collection is by no means a trivial problem in itself and the overheads of such collections and the recalculation of parity groups would be severe should it be required.

#### 4.2.4.5 Adaptive Parity Caching

Adaptive parity caching [FM98] was proposed as a solution to the problem of wasted space due to the retention of old parity groups in the parity logging scheme. The idea behind parity caching is that adjacent disk blocks are usually part of the same file or data stream and are likely to be read or written together. If the rewritten blocks were placed in the same parity group then the space occupied by the old group could be reclaimed easily. To aid this process, the client maintains a number of buffers and attempts to order the blocks before creating parity groups and then sending to the appropriate servers.

#### 4.2.5 Sub-Page Fetches

It may be possible to speed up application execution by splitting pages into sub-pages for transferring across the network. As reported in [JFV<sup>+</sup>96], a sub-page of 2KB (for 8KB pages) could increase application performance by up to 40%. However, the results reported by the authors depended a great deal on computational overlap, i.e., the ability for the application to progress while the rest of a page is being transferred and then being able to interleave application and page processing once it arrives. This may be a lot easier in a UNIX environment in which the

application is the only one running and there is not a lot of competition for the CPU, but in a QoS environment, where CPU guarantees are valid regardless of other running processes, the computational overlap may not be as significant. For instance, if an application has a 1ms slice of the CPU every 10ms (i.e., 10% of CPU bandwidth) then it may not have the CPU for a significant amount of time while the transfer is being carried out. Furthermore, once the data arrives, the processing of it is the responsibility of the application and that time is deducted from its resource guarantees – it is not billed to the device driver.

#### 4.2.6 Acquiring Physical Memory

Acquiring physical memory under the GMS scheme, described in section 4.1, is based upon usage. On any given page-out, the global LRU page is expelled and, if it belonged to a different process, the faulting process' physical allocation is increased by one frame. In an operating system supporting QoS, such as Nemesis [Ros95], each domain has a contract with the frames allocator for a certain number of guaranteed physical frames [Han99]. In the case where the memory is remote, it is the paging server that maintains the contract with the frames allocator on behalf of clients. For a client to initiate this contract, it must first find a server with free physical memory.

There are several ways of tracking which hosts are willing to act as paging servers. Markatos and Dramatinos [MD96] maintained a list of servers in a file that could be accessed by clients. GMS [FMP<sup>+</sup>95] use epoch servers to distribute global LRU information to participants. Neither scheme deals with the possibility of the machine holding this information crashing. Feeley et al. mention the possibility of the epoch server crashing and suggest an election scheme could be implemented to resolve it. However, the GMS scheme does not cope well with server crashes. The nature of the scheme not only requires an election of a new epoch server but also the redistribution of data-structures across the remaining hosts.

Another possible way of tracking hosts with free memory is to have them register how much free memory they have with a trader. Clients could then contact the trader to acquire a server with enough free memory. This means that the clients have to have a way of locating the trader. There are some standard methods of advertising services in a distributed system using, for instance, IP multicast [Edw99]. Depending on the involvement of the trader, some form of replication could be used to minimise the effect of a trader crashing. For example, if the trader only held

information about what hosts were advertising physical memory and was not involved in the actual process of acquiring resources beyond that (as in [SD91]) then replicating the trader at, say, two other locations should be ample to ensure that, in the vast majority of cases, a trader would always be available.

#### 4.2.7 Revocation

In a remote paging environment the aim is to utilise **free** memory on hosts within any given distributed system. Thus, we would like to be careful about denying local domains physical resources, such as memory and network bandwidth, due to the behaviour of domains on remote hosts. Under the GMS system, pages are gradually freed by the global page replacement algorithm, however, an operating system offering resource guarantees requires the explicit return of committed resources. Subsequently, a revocation policy that specifies under which conditions resources can be re-acquired by the system is required. Similarly, a mechanism must be provided that allows domains to know when resources have been revoked in order that they can adjust their behaviour accordingly.

Although acceptable under normal conditions, there are situations where revocation is slightly more complicated. In the case where a server is acting on behalf of a diskless client, that client does not have the facilities to handle its paged-out data. This could be handled in several ways. One way would be to prioritise the memory a server holds for clients such that clients with no local disk are offered the highest priority. Under this scheme, revocation of pages held on behalf of diskless clients would be a last resort.

Another possible way to deal with diskless clients is to negotiate the guarantee for the lifetime of the domain. This is a rather serious commitment and could incur a heavy penalty for local domains. Of course, we could take the opposite standpoint and inform the client that it must make way for the pages being returned or risk losing them. There are ramifications for the return of pages that must also be taken into account, for example, the server must agree a rate of return that allows the client to meet its local guarantees.

Below is a summary of some possible revocation policies.



#### 4.2.7.1 Transparent Revocation

Figure 4.6 shows a revocation scheme where *Server 1* first offloads the pages held on behalf of the client before notifying the client that *Server 2* will now act as the remote paging server. Note that, on the *Forwarded* notification, the client may still be sending pages to *Server 1*. However, when the client responds to the *Forwarded* notification, it must not send any further pages to *Server 1* and must not request any pages from *Server 2* until it has received the *Flushed* notification, informing the client that *Server 2* is completely up to date. This prevents the client from using *Server 2* before it is fully up to date and also prevents it from sending further pages to *Server 1* after it has forwarded all of the client's pages to *Server 2*.

Under this scheme, pages expelled by the client will be sent to *Server 1* in the first instance. *Server 1* will then forward them to *Server 2*. This may cause some pages to be sent across the network twice, but this is unavoidable if the offloading of pages is to be transparent to the client. This transparency does not hide the location of pages from the client — *Server 1* holds all the pages until *Server 2* is fully up to date and the client has been notified of the new location. However, during the transition between servers, the pages exist at more than one location.

The drawback of a transparent scheme is that the server has very little knowledge about the client's running environment. The client may be using several servers for handling different areas of memory and this would not be apparent to any one server. The client is aware of all of the resources it is utilising and is thus in the best position to determine a suitable replacement for any revoked resource. Another consequence of the transparent scheme is that the client cannot perform any paging between times  $t_7$  and  $t_9$ .

#### 4.2.7.2 Active Participation Revocation

Figure 4.7 shows a revocation scheme where the client is responsible for acquiring other resources revoked by *Server 1*. In this case, *Server 1* informs the client that it is revoking its resources and awaits a reply. The client will first attempt to acquire a suitable server and, if successful, responds by telling *Server 1* to hold its pages for the time-being. If a server cannot be attained, then the client can inform *Server 1* that it will take the pages back, or, if the client has the pages mirrored elsewhere, instruct *Server 1* to simply discard the pages. On instructing *Server 1* to hold its pages, the client then contacts *Server 2* and initialises the appropriate resources. It then instructs *Server 1* to populate *Server 2* with the pages held on its behalf. Once

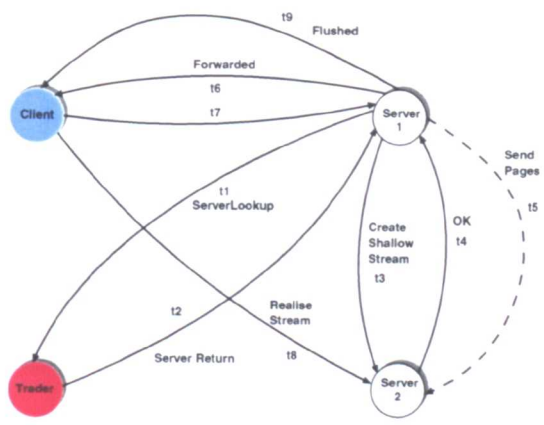


Figure 4.6: Transparent Revocation.

the pages have been offloaded, *Server 1* contacts the client, informing it that *Server 2* is up to date. The client then switches over to using only *Server 2*.

While *Server 1* is populating *Server 2*, all expelled pages at the client are only sent to *Server 2*. If *Server 1* sends a page to *Server 2* that it already has, then *Server 2* simply discards it. Because this process is not transparent to the client, the client can keep track of which pages it has sent to the new server. It does not, of course, know which pages have been sent to *Server 2* by *Server 1*. Consequently, faulting on a page that has not been sent by the client to *Server 2*, causes the page to be faulted-in from *Server 1*.

Like the transparent scheme described above, the client is always aware of the location of each page. However, in contrast to the transparent scheme, the client has complete control over the selection of a suitable replacement.

4.2.7.3 Unequivocal Revocation

Figure 4.8 shows a revocation scheme where the server simply informs the client that its resources have been revoked and the client must take the pages back. Under this scheme, the server must allow the client suitable time to receive the pages before the resources are fully withdrawn.

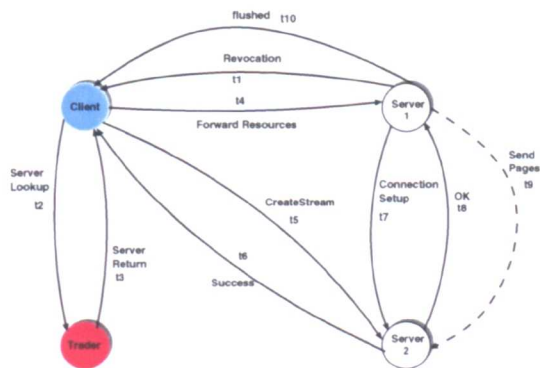


Figure 4.7: Active Participation Revocation.

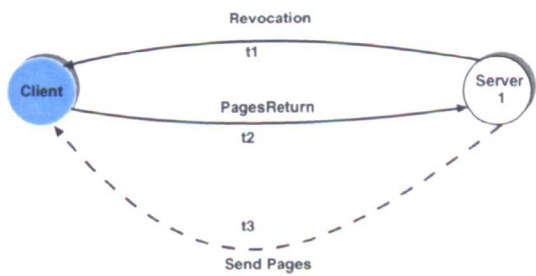


Figure 4.8: Unequivocal Revocation.

### 4.2.8 Quality of Service Guarantees

Quality of service requirements for remote paging incorporate system guarantees on many levels. Some considerations to ensure predictable behaviour are:

- CPU bandwidth guarantees at server and client.
- Network guarantees between server and client.
- Physical memory guarantees at server/s.
- Rate of return — following revocation.

An application must have enough processor bandwidth to make progress at both client and server. The client is “billed” for all local resources and servicing page faults happens within the limits of its local guarantees. The server, on the other hand, is not billed directly to the client: its guarantees are met by the machine it is running on and so we must find a way of tying the client to the server and ensuring that a client is not permitted to “hog” a paging server. This can be done by providing QoS parameters for the amount of time the server spends dealing with requests from a particular client.

To provide true determinism, an application must be able to acquire network bandwidth guarantees and coordinate those guarantees with CPU guarantees at both ends. This allows us to determine the end-to-end performance for any given combination of client, network, and server resources/guarantees.

For remote paging to be viable at all, an application must be able to acquire physical memory guarantees from a remote host. To augment this, a rate of return must be agreed, should this guarantee change and a client find its resources revoked, for pages held on a client’s behalf. The rate of return allows both the client and the server to meet their respective local guarantees.

An interesting difference between paging to a local disk and paging across a network to memory on a remote host is the reliance on CPU bandwidth. When paging to a local disk, a request is issued to the device driver and the requesting thread is blocked pending completion. Once the request has been serviced, the thread is again runnable. In the interim, the thread has used no CPU resources. Consider the case where an application is running with a guaranteed CPU slice of 1 milli-second every 10 milli-second period. If the time taken to service a page fault from

the local disk is 2 milli-seconds, the application can effectively absorb 4 page faults per 10 milli-seconds (provided it has the necessary bandwidth and CPU utilisation is less than 100%) and still receive 1 full milli-second on the CPU. However, a remote paging system will invariably have to copy data, possibly from memory into a transmit buffer and, from a receive buffer to memory at the remote host. This requires CPU intervention. The consequence of this is that even if sending a page across the network is quicker than sending it to the local disk, the number of page faults per given period could influence the overall performance.

The need for CPU bandwidth at the server has ramifications for the server's decision to advertise free memory: even if the server has a great deal of free memory, there is no point advertising the fact unless it also has enough free CPU bandwidth to service requests from clients in a reasonable time. This is discussed further in 10.3.

## 4.3 Discussion

Although aspects of remote paging have been tackled by researchers, no-one has yet tackled the general problem of remote paging in a distributed system of architecturally heterogeneous hosts. GMS, and later PGMS, provided a solution for a tightly coupled cluster of homogeneous hosts, both architecturally and with respect to the operating system, connected via an extremely fast network, and Markatos and Dramatinos provided a solution for architecturally homogeneous hosts all running OSF/1. However, the system described by Flouris and Markatos seemed to offer a limited form of heterogeneity, with the servers running one operating system and the clients another. While each solution may be suitable for its specific environment, neither could be construed as a general purpose solution. Furthermore, neither scheme is sufficiently flexible to allow hosts to dynamically alter their participation in the scheme. More specifically, individual processes cannot choose whether to participate or not.

Another important oversight in the remote paging systems described in this chapter is the effect of CPU usage. In order to satisfy requests for pages from remote clients, the servicing thread, be it the operating system or a daemon of some description, must have CPU resources. Furthermore, the CPU load will have an important effect on overall throughput. It is quite feasible for a machine to have a lot of running domains and still have free memory. These domains, though not affected by memory being given over to storing pages from other hosts, would be affected by CPU resources being given over to the servicing of page faults. Similarly, the servicing of page

faults is affected by the CPU usage of local domains. For a remote paging system to be viable, the availability of CPU resources at the server must be taken into account.

Three key factors can be derived from the requirements of a remote paging system: performance, knowledge of page locations, and flexibility. A remote paging system cannot offer itself as a viable alternative to paging to the local disk if it does not result in an increase in performance for the applications utilising it. Of course, where there is no local disk available, the performance of paging across the network relative to paging to a local disk is not an issue. Linked to the performance of a remote paging system is the issue of knowing where a particular page is. If the location of each page is not known, then the time taken to fetch a particular page increases with the number of locations to which a request is forwarded. Finally, the system must be flexible enough to allow hosts to join and leave the scheme at any time to allow for the dynamism inherent in a distributed system.

We have now covered the issues relating to paging to the local disk and paging across the network. The next chapter looks at how the number of page faults serviced by the backing store can be reduced in order to speed up overall application performance.

# Chapter 5

## Compressed Caching

*He can compress the most words into the smallest ideas of any man I ever met.*

**Abraham Lincoln.**

The memory requirements of average programs on workstations have grown by 50-100% per year over the last decade and this growth is set to continue [KGJ96]. Compare this to the rate of increase in memory cost/capacity (45% per year) and we see that memory is as valuable a commodity as it has always been. This is particularly true for mobile computers, where increases in memory sizes have been much more modest. If we examine this data in the context of increases in CPU speeds and disk latency, we can conclude that paging to a local disk is more expensive, and just as likely to occur, than it ever was. Consequently, interest in storing data in memory in compressed form has risen over the last five years.

Approaches to utilising compression as a way of “extending” physical memory has come in two forms: hardware and software. Kjelsø et. al. [KGJ96] describe a hardware data compressor with a compression throughput of 100MB/second. They show several-fold increases in performance for a range of traditional UNIX applications by eliminating paging to a local disk.

A new architecture for compressed memory has been proposed by Franaszek et. al. [FR98, FHW99, BFR00] which could be viewed as extending the notion presented by Machanick et. al. [MSP98]. All data in main memory (level-4) is in compressed format and is decompressed when faulted into the level-3 cache. Figure 5.1 shows the layout of the proposed memory hierarchy. The level-3 cache is typically quite large, in the region of 16 or 32MB. When a line is evicted

from the level-3 cache, it is compressed before being stored in level-4. If a page is faulted in from backing store, it is faulted in directly to level-3. This system has the advantage that the compression engine operates at a bandwidth close to that of the physical memory system.

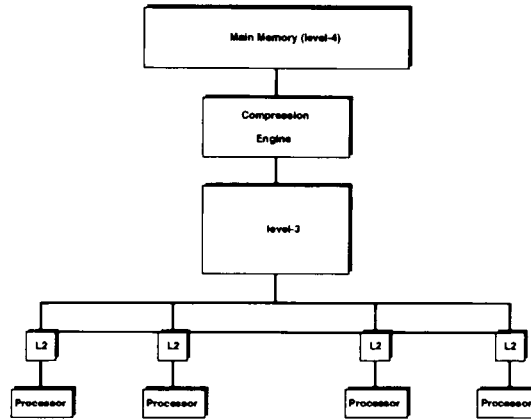


Figure 5.1: Compressed Memory Architecture.

The software approach to using a compressed cache involves splitting physical memory into two chunks: one chunk consists of the active pool of pages; the other is used as a cache into which pages expelled from the active pool are compressed, instead of being sent to backing store (see figure 5.2). Only when pages are expelled from the cache are they written to backing store. The idea here is to obviate the writing of pages that may be required again soon to backing store. This method relies on three inter-related factors: that we can achieve a compression ratio high enough to make the compression worthwhile (even a 2:1 ratio may give us significant gains in performance); that the time taken to compress/decompress a page is significantly less than the time taken to write/read a page to backing store; and that the extra paging overhead incurred from having the active pool reduced in size, in order to make space available for the cache, should still out-perform paging to backing store.

While the hardware approach potentially offers increased throughput, the idea is still in its infancy and one would not expect to see such support for many years. Consequently, this chapter focuses on the software approach and the issues pertinent to implementing such a scheme. Section 5.1 discusses the approaches other researchers have taken to compressed caching. Section 5.2 elaborates on the issues involved in providing such a scheme, before conclusions are drawn in section 5.3.



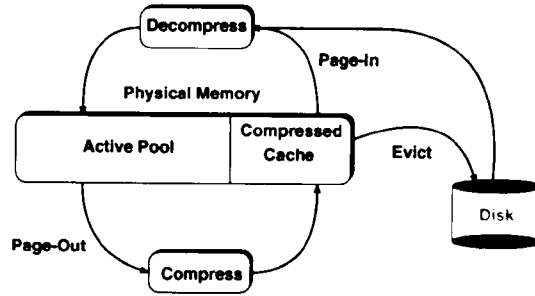


Figure 5.2: Paging to/from a Compressed Cache

## 5.1 Background

The potential benefits of on-line compression have increased dramatically in recent years due to the increasing disparity between CPU, memory and disk speeds. For instance, Burrows et. al. [BJLM92], as far back as 1992, suggested using on-line compression as a space-saving mechanism in Sprite LFS but noticed a degradation of a factor of up to 1.6 for file intensive operations. However, their throughput was only between 0.4 and 1.7MB per second. Contrast this performance with the 20MB per second reported by Wilson et. al. [WKS99] and we can see that, in just a short space of time (some 6 years), the compression throughput has increased by more than an order of magnitude. Should the disparity between CPU, memory and disk speeds continue, as is expected, then we should continue to gain more benefits from on-line compression techniques both as a space saving mechanism and as a method for reducing paging overheads.

Douglis [Dou93] described the use of a compressed cache in the Sprite Operating System that changed size dynamically according to the relative usage of pages in the cache, the compressed cache and file blocks. The cache is implemented as a circular buffer with new entries added to the end and old entries evicted from the front. However, should no clean entries be found at the front, entries could be expelled from the middle. The cache is not divided into sub-pages, instead entries are fitted together as they are entered in the cache. This arrangement means there is no upper bound on the external fragmentation. Furthermore, it is quite possible that a page may be evicted from the cache to make way for a new addition even though there is enough space. The alternative, though a potentially expensive one, is that you compact the cache.

Douglis reported speedups between 0.73 and 2.68. In fact, only three applications of the seven used in the test suite reported gains in performance. Wilson et. al. [WKS99] blamed the adaptive caching strategy employed by Douglis for these relatively unpromising results.

Russinovich and Cogswell [RC96] evaluated the potential benefits of a compressed cache in the Windows95 OS but concluded it would not be beneficial. This seems to be related to the fact that the compression speed they assumed was extremely poor - around 2ms to compress or decompress a page. This speed was used to incorporate both the compression/decompression time and the time taken for Windows to service a page fault. However, the same consideration did not seem to be given to accessing the local disk. Consequently, compressing a page was only five times faster than writing to the local disk. As shown by figure 5.3 (section 5.2.1), even on a relatively slow machine (an Intel P200), compressing a page is an order of magnitude quicker than writing to the local disk.

Wilson et. al. [WKS99] simulated the implementation of a compressed cache under differing memory sizes with different compression algorithms, using their own compression algorithm. The results were an average performance increase of 40%. Two key contributions are made in their paper: the compression algorithm (WKdm), which out-performs even the mature LZO algorithm; and their method for adapting the size of the compressed cache to take into account recent program behaviour. The strategy for adjusting the size of the compressed cache uses a combination of least recently used (LRU) ordering and information on recently evicted pages. From this, it performs a “what if analysis” on the current split of memory and attempts to determine if there may be a better one. The actual split is generalised to a number of target compression sizes: 10, 23, 37 and 50% of memory. The “what if analysis” occurs on every uncompressed cache miss and takes a “few hundred instructions”.

Wilson et. al. [WKS99] traced six UNIX programs on an Intel x86 machine running Linux and captured page image traces. The simulator takes a record of the touched pages with compressibility information and the cost of compression/decompression. For the purposes of the experiments, disk writes were assumed to be free and the disk seek time was assumed to be 5ms. The results showed that for varying memory sizes, the paging costs when using a compressed cache were significantly smaller than for the traditional VM structure due to less disk I/O.

Though more encouraging than the results reported by Douglass, there are a couple of potential problems with the scheme described by Wilson et. al. Because the compression cache was simulated, it was easy to separate the maximum potential benefits for each individual application. However, there are limitations to simulating performance. Firstly, there is no accounting for hidden implementation costs that could affect performance. For instance, there is no information on how items are placed in the cache: Douglass [Dou93] originally intended to use an LRU cache

split into sub-pages, where entries could be broken up and scattered throughout the cache, but found that reclaiming physical memory for use by the active pool was much more expensive. He subsequently resorted to a circular buffer implementation which did not split the cache into sub-pages but attempted a kind of best-fit. Secondly, it is hard to accurately account for costs. If, like Wilson et.al., we assume that disk writes are free and the disk latency is 5ms, then on each disk read we pay the cost of latency plus the time taken to read the page. This would result in a disk read costing around 6-7ms. In theory, this seems quite reasonable but in practice, the latency varies quite dramatically depending on the last position of the disk heads. This means that the latency for the second disk access could be dramatically different from the first. A further problem with simulating the cache performance for individual applications is that you do not take into account the effect of interference from other applications. Although a simulation, Wilson et.al.'s target environment seems to be UNIX-like, i.e. an environment where the OS manages the physical memory and implements the page replacement policy.<sup>1</sup>

Both Douglass and Wilson performed measurements on UNIX-based systems (Sprite is largely compatible with 4.3BSD) and yet provided per-application performance results. Although this provides an upper-bound on performance gains, it does not accurately reflect a running system. In a UNIX system where processes compete for memory, the compression cache would adjust in size dynamically according to overall performance. This means that an application with very poor locality, running at a time when many others with good locality are also running, may be penalised by the reduction of the active pool even though that reduction is not to its benefit. An ideal environment would provide memory management on a per-process basis. This would allow each process to determine if a cache would be beneficial or not. It may also make the need for an adaptive strategy largely redundant.

## 5.2 Issues in Compressed Caching

This section discusses some of the issues pertinent to the employment of a compressed caching scheme. Some sub-sections here refer to measurements obtained from applications which are more fully described in appendix G. Note that, because these applications were run with specific

---

<sup>1</sup>The fact that the cache adapts according to recent performance suggests that the authors assume a competitive environment.

guarantees from the system, the results varied very little over many runs (typically within 1%). See section 12.2 for a fuller discussion on the variation of results.

5.2.1 Compression Speed

As mentioned in section 5.1, the relative increase in processor speeds compared to memory and disk speeds has made on-line compression increasingly attractive. Figure 5.3 shows the relative times taken to compress/decompress<sup>2</sup> a page in memory and read/write to the local disk for the Nemesis operating system. The measurements were taken on an Intel P200 with 64MB RAM. The domain was guaranteed 100% of the disk bandwidth. The drive used was a Seagate ST32132A DMA IDE hard drive with 128KB write cache. One hundred pages were written consecutively followed by one hundred consecutive reads and the average taken.



Figure 5.3: In-Memory Compression Vs Reading and Writing from/to Disk.

As can be seen, even the slower process of compression is an order of magnitude quicker than a disk read or write at full throughput. However, it is not enough for the compression/decompression to be faster than a disk read/write, we must also consider the effects of using less working memory on paging behaviour. A smaller active pool means more page faults. Thus, given a smaller

<sup>2</sup>Using the WKdm compression algorithm, available from <http://www.cs.utexas.edu/users/oops/compressed-caching/index.html>.

active pool, the overall performance of using a compressed cache must be significant enough to out-perform paging using a local disk but with a larger active pool of memory.

Figure 5.4 shows the time it takes to compress, decompress and copy a system page of 4KB in memory for three Intel processors: the Pentium P200, the Celeron 366 and the Pentium PIII 450. The compression/decompression rates on the PIII are almost four times faster than on the P200. This suggests that in-memory compression will become increasingly attractive as processor speeds increase.

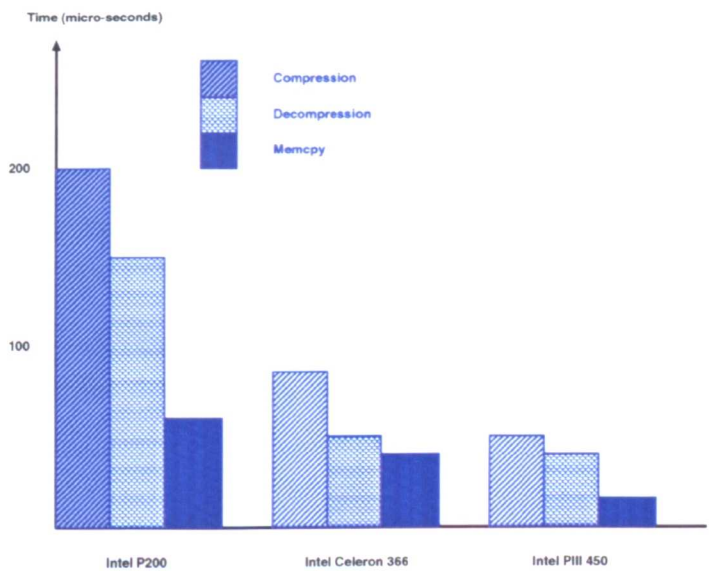


Figure 5.4: In-Memory Compression and Copying for Different Processor Speeds.

If we examine the figures for the PIII 450, we see that compressing a page takes 55 micro-seconds and decompressing a page takes 41 micro-seconds. This gives us a compression throughput of 71MB/second and a decompression throughput of 95MB/second. These figures are not very far away from the 100MB and 140MB per second throughput for Kjelsø et. al.’s hardware implementation. This shows more than a threefold improvement for compression throughput in the two years since Wilson et. al. reported their results. In that same time, the time taken to write a page to disk has reduced from 2.2ms to 1.4ms (from the P200 using a Seagate ST32132A IDE Drive to the PIII 450 using a Fujitsu MPE3102AT ATA IDE Hard Drive). These figures suggest that the viability of compressed caching is improving as CPU and disk speeds diverge further each year.

### 5.2.2 Compression Ratio

The compression ratio must be such that there is a space-saving benefit to compressing pages. If the cache holds little more than it would had the pages not been compressed then the cost of having it may be more than the benefits. Wilson et. al. [WKS99] reported an average compression ratio, for data pages, of 2:1 and an average performance increase of approximately 40%. The performance increase relies on applications performing a significant amount of paging.

Kjelsø et. al. [KGJ98] examined the compressibility of memory-data, which they define as everything in memory (code and data), for a suite of 16 typical UNIX applications plus the operating system itself (SunOS 4.1.3). The total volume of memory-data for these tests exceeded 500MB. The authors found that memory-data contain a large proportion of zeros which often occur in contiguous runs. They also found that low values and ASCII lower case letters have greater than average probabilities. They concluded that memory-data typically compresses by 50%, doubling the amount of data that can be kept in memory in compressed form.

This, taken with the throughput results from section 5.2.1, suggests that compressing data in memory is not only viable due to speed, but also due to the compressible nature of data in memory.

### 5.2.3 Global Management

Once it has been decided to utilise compressed caching, a policy for the management of pages in physical memory must be decided. Two possible schemes for the management of physical memory are redundant caching and exclusive caching.

**Redundant Caching:** A redundant caching scheme allows for the possibility that pages can reside in both the active pool and the compressed cache simultaneously. When a page is faulted in from the cache to the active pool, it is decompressed and the compressed form is maintained in the cache. If that page should be subsequently evicted from the active pool, and has not been written to, there is no need to compress it again — unless it has since been evicted from the cache. This method sacrifices memory in favour of reducing the number of compressions. A consequence of such a scheme is that a similar page replacement policy used in the active pool may be required for the cache as well.

**Exclusive Caching:** An exclusive caching scheme maintains only one copy of a page in memory at any time. A page faulted in from the cache is decompressed to the active pool and then the space in the cache is freed. Thus, if that page should then be evicted from the active pool, it must be compressed once more whether it has been written to or not. This method attempts to keep as many different pages in memory as possible at any one time.

The reasoning behind redundant caching was that compression was expensive and an attempt should be made to reduce the number of times it had to be performed. This is simply no longer true in today's environment.

#### 5.2.4 Page Access Patterns

There is no point in providing a compressed cache if page access patterns dictate that “cache-hits” will be infrequent and most of the pages compressed to the cache end up on backing store before being used again. In such a case, the application would be severely penalised. Firstly, it is penalised by having its active pool reduced, potentially causing more page faults. Secondly, each eviction from the active pool can cause a compression (depending on the overall management) plus a potential write to disk. Thirdly, each page fault may cause a disk read plus a decompression. For instance, the *compress* application, whose memory access pattern is linear, suffered a performance decrease of 25% over paging to a local disk when run with a compressed cache and was nine times slower than with using the local disk and the *friends* prefetching algorithm (described in 9.3.1).

Linked to this is the page replacement policy implemented in the active pool. Kaplan [Kap99] studied memory reference patterns in programs and found that the replacement policy used in the active pool had a marked effect on the compressed cache performance. He found that although LRU was a good general policy, it was very poor for some types of programs, e.g. programs with loop-like access patterns. This led to the development of early eviction least recently used (EELRU). The idea behind EELRU was to use LRU where it performed well and adapt it, evicting pages early, when pages were seen to be being faulted on after being recently evicted.



5.2.5 Cache Size

How one organises the segmentation of the active pool and the compressed cache can have an important impact on application performance. Wilson et. al. [WKS99] blamed the poor results achieved by Douglass [Dou93] on the adaptive caching strategy used in his tests. Figure 5.5 shows the effects of different sizes of cache, as a percentage of total memory, for a matrix multiplication program (this application is fully described in section G.1). The program multiplied two matrices of integers with dimensions 500 by 600, producing a result of 500 by 500. The application was run with two megabytes of physical memory. As can be seen, the difference in the level of performance can be quite significant.

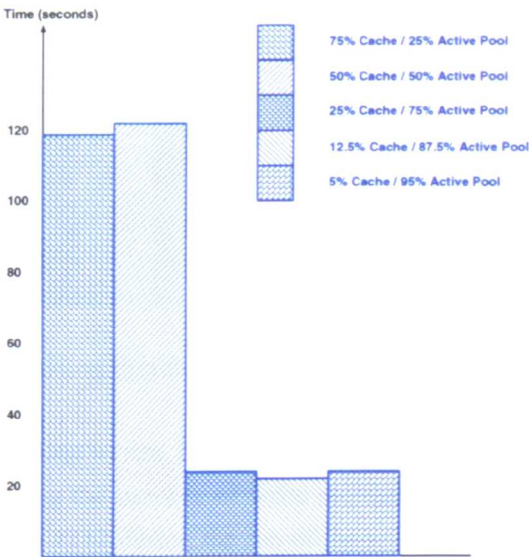


Figure 5.5: Effect of Cache Size as a Percentage of Memory on Performance

When the cache is greater than 25% of the physical memory, the application faults at a significantly higher rate. At this point, the number of compressions and decompressions is sufficiently high to be slower than servicing a much smaller number of page faults from the local disk.

An interesting aspect of the compressed cache is how it interacts with page prefetching and group writes. If we take an application that performs worse with a compressed cache and add prefetching and group writes to the cache management, what effect does this have on performance? Figure 5.6 shows the performance of an application which applies a sharpen filter to a 3210KB bitmap (see appendix G for full description) running with 8MB of physical memory. The cache



was augmented with the *friends* clustering and prefetching algorithm described in chapter 9. As can be seen, although including a 1MB cache causes the performance over local paging to reduce by over a third, adding prefetching and group writes causes more than a three-fold performance increase.

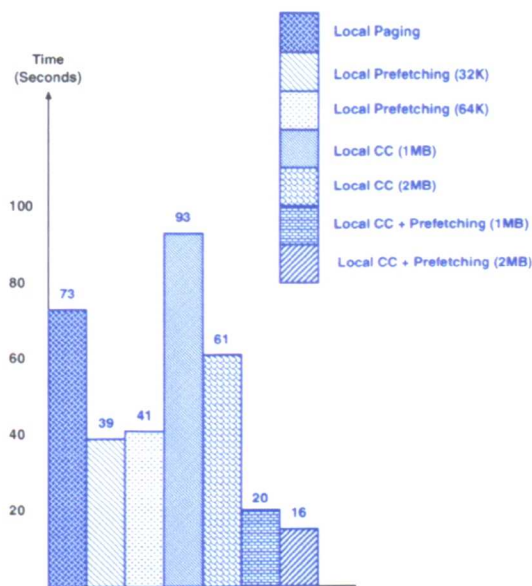


Figure 5.6: Effect of Adding Prefetching and Group Writes to a Compressed Cache.

By adding prefetching and group writes, the percentage of cache hits increase from 14% to 50%. This has the effect of reducing the number of write requests from 5630 to 905 and the number of read requests from 3845 to 1212 during the run of the application. Even in the case where the cache is increased in size to 2MB and the performance improves, adding prefetching and group writes still increases performance over three-fold.

5.2.6 Cache Arrangement

How pages are stored in the cache can have an effect on how the cache performs. For instance, as mentioned earlier, Douglass originally intended to use an LRU cache split into sub-pages, where entries could be broken up and scattered throughout the cache, but found that reclaiming physical memory for use by the active pool was much more expensive.

The decision about whether to split the cache into sub-pages or simply attempt to fit pages in as well as possible affects potential fragmentation. Using sub-pages allows us to determine the average fragmentation per page. It also makes cache management slightly easier due to using fixed size blocks.

If it is decided to segment the cache into sub-units of a system page, then the size of these sub-pages can have an effect on the performance of the cache. Figure 5.7 shows the results of the effect of differing sub-page sizes in the compressed cache. The program multiplied two matrices of integers with dimensions 1000 by 2000, producing a result of 1000 by 1000.<sup>3</sup> The left bars indicate the difference in cache-hit percentage and the right bars show the effect on overall run-time.

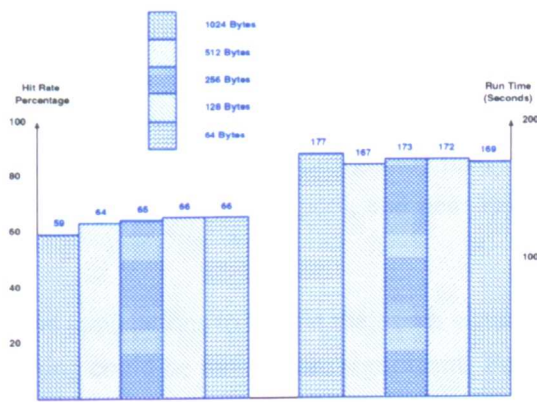


Figure 5.7: Effect of Sub-Page size on Cache Performance

A sub-page size of 128 bytes has one of the largest hit rates and performs best in terms of overall run-time. Although the average size of a compressed page for the above experiment is around 1500 bytes, the smallest compressed page is 28 bytes. Such a page would only waste 100 bytes of cache space. Even though this is more wastage than would be incurred using 64 byte blocks, there are not enough instances of 28 byte pages to increase the cache hit percentage using this size. Thus, the 128 byte sub-pages seem to offer a good compromise between fragmentation and the simplicity of managing the cache.

<sup>3</sup>Note that the size of the matrices were increased only to increase the length of time the application takes to complete and to highlight more clearly the effect of altering the sub-page size.

### 5.2.7 Responsibility

The previous sub-sections have shown that the relative benefits of a compressed cache can depend greatly on the type of applications running at any one time, and yet Douglass' scheme described in [Dou93] was implemented as part of the VM system where the compressed cache was shared by all applications, even if the testing focused on a single running application, and the contents were based on a global paging scheme. The scheme simulated by Wilson et. al. [WKS99], though not implemented within a VM system, was based on simulations of traces of applications executed on machines running UNIX. Subsequently, if it were implemented in such an environment it would be subject to the same limitations as that implemented by Douglass. The problem with the compressed cache being shared by all applications is that they all must pay the cost for the behaviour of others. If a compressed cache greatly benefits one application but the performance and compression ratio of others causes the cache size to be reduced to an insignificant amount then the benefitting application will be penalised. Similarly, if an application receives no benefit from the cache but others in the system are benefitting greatly then more resources may be given over to the cache, reducing the size of the active pool, and penalising the application receiving no benefit.

In an environment that allows applications to handle their own virtual memory management, the benefits of compressed caching could be much more significant. Furthermore, applications that do not benefit from a compressed cache could decide not to use one. By making the compressed cache domain-specific, it is possible to have the cache use much more accurate prefetching techniques to increase performance.

A further consequence of the utilisation of a compressed cache in a UNIX-like environment is that the operating system performs the compression/decompression on behalf of the applications. This, of course, means that the applications not benefitting from the compressed cache, due to poor locality, are further penalised by the operating system spending time compressing and decompressing pages. This also has the effect of masking the cost of a compressed cache by amortising that cost over all running applications. Thus, it is easy to see the benefits of running a single application, which has good locality, with a compressed cache. Firstly, this application will not be penalised by applications reducing the cache size. Secondly, in an environment using round-robin scheduling, the application spends most of its time on the CPU and blocks only for small periods during de/compression. In such an environment, blocking on the disk is exaggerated by the fact that the application is otherwise afforded almost all of the CPU. How

does competition for the CPU affect the performance of compressed caching? What happens if the application had to perform the de/compression itself? Would compressed caching continue to be viable?

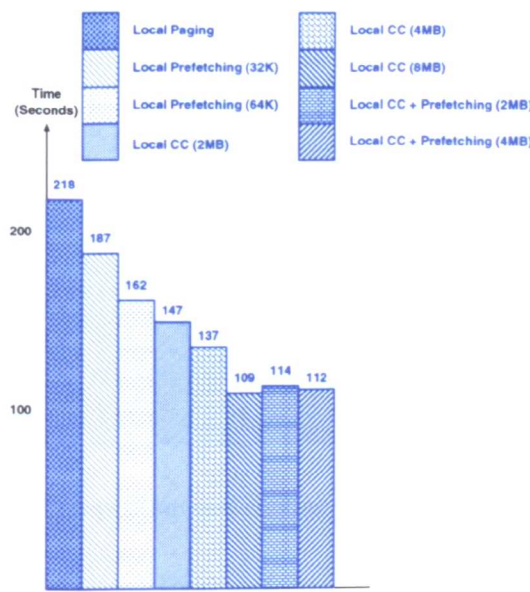


Figure 5.8: Matrix Multiplication Without Competition for CPU and Disk.

Figure 5.8 shows the performance of the matrix multiplication application (1000x2000), run with 16MB of physical memory, for paging to the local disk with and without a compressed cache. There is no competition for CPU or disk. As can be seen, the application benefits greatly from a compressed cache. By splitting the memory 50-50 between the cache and the active pool, the application’s performance increases by 50%. There are two main reasons for this improvement. Firstly, because there is no competition for the CPU, disk accesses are relatively more expensive: disk accesses cannot be interleaved with processor bandwidth. Secondly, because the time taken to service a page fault from the disk is an order of magnitude slower than de/compression, the CPU time spent performing the de/compressions is far outweighed by the time the application would spend blocked on I/O.

Figure 5.9 shows the same application run with its CPU bandwidth limited to 1 milli-second every 10 milli-seconds, i.e., 10% CPU bandwidth (see chapter 7 for details on OS support for this). The application was the only disk client and was assured 100% bandwidth. As can be seen, the effect of limiting the CPU, in light of 100% disk bandwidth, is quite significant. By scheduling the application in such a way as to afford it only 1ms/10ms, the disk accesses can be



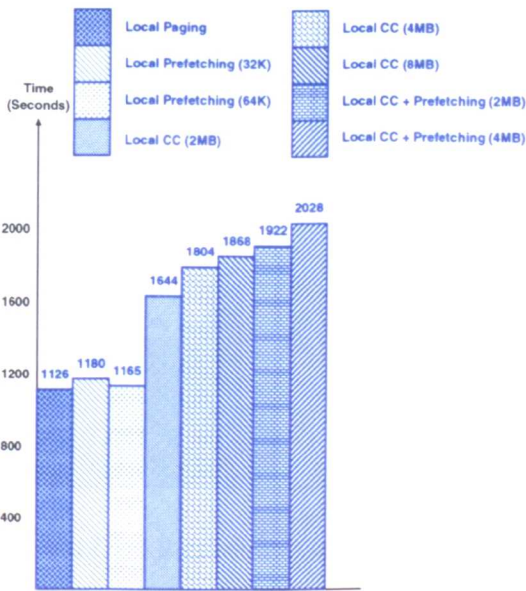


Figure 5.9: Effect of CPU Competition and Having to Perform One’s Own De/Compression.

interleaved with CPU access over that 10ms period in such a way that the application can resolve its faults and still get most of its allotted CPU bandwidth. The de/compression, however, now actively eats into that bandwidth more significantly than performing I/O.

Figure 5.10 shows the same application run with its disk bandwidth also limited to 10% (100ms every 1 second). As can be seen, the effect of limited CPU and limited disk bandwidth has a strong impact on the best hierarchy configuration for the application. When CPU bandwidth was at a premium with respect to disk bandwidth, the compressed cache severely curbed the application’s performance. However, when access to both resources is equally competitive, compressed caching once again becomes more viable.

5.3 Discussion

We have discussed the issues pertinent to the employment of a compressed caching scheme and can conclude that such a scheme could offer real benefits to applications in an environment where paging is likely. CPUs are sufficiently fast to make software compression/decompression extremely fast and memory-data is compressible by, on average, a factor of two. However, we

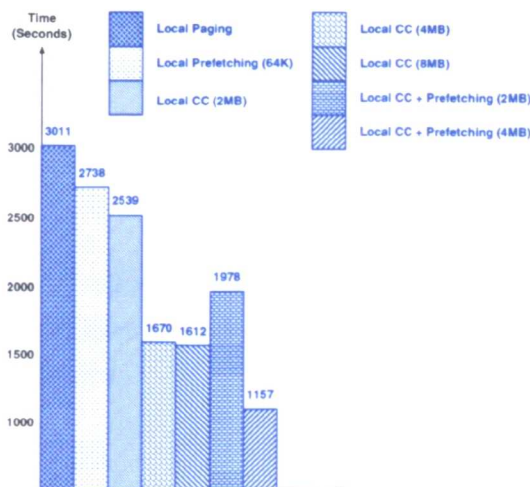


Figure 5.10: Effect of CPU and Disk Bandwidth Competition on Compressed Caching Performance.

have also shown that the amount of memory allocated to each pool (active and compressed) has a significant effect on cache performance, as indeed does an application’s memory reference behaviour. An interesting, though previously unexplored, aspect of compressed caching is how it interacts with disk group writes and prefetching. It has been shown that, even when a particular memory split seems to inhibit application performance, adding group writes and prefetching can significantly improve an application’s performance.

Although investigated by several researchers, a compressed cache implementation within a commercial operating system is still a long way away. This could be attributed to the deficiencies of the schemes proposed or it could be due to the deficiencies of modern operating systems. In an environment where the operating system performs the virtual memory management on behalf of all processes there is, by necessity, compromise. The result of this is that researchers have been looking at adaptive caching methods that try and guess what configuration would be best for all the running processes. I would argue that this is the wrong approach for software compressed caching. The benefits of a compressed cache are there to see and it is obvious that these benefits will increase as the disparity between disk and CPU speeds increases. However, a compressed cache does not benefit every application and an environment that attempts to adjust the cache size according to all running applications, in essence, penalises them all.

A significant oversight in the work carried out on the potential benefits of compressed caching has been the effect of competition for CPU and resources. I have shown here that competition

for the CPU severely affects application performance. It has also been shown that applications benefitting from compressed caching under light CPU load, i.e., where there is a small number of competing processes, can perform better without it when that load gets heavier. It is not until competition for resources starts to balance out that compressed caching once again comes into its own.

We have now covered the issues relevant to local paging, remote paging and compressed caching. The next part sums up what we have discussed and describes the environment within which the Heracles VMM system is provided.

## **Part II**

# **Towards a Better Environment**



# Chapter 6

## Discussion

*Never hold discussions with the monkey when the organ grinder is in the room*  
**Sir Winston Churchill.**

In the previous part of this dissertation, we have looked at how operating systems manage memory on behalf of applications. We concluded that the approach taken by operating systems is inadequate for the needs of modern applications. The operating system is not aware of the needs of each individual application. Consequently the OS meets the general needs of many applications while meeting the specific needs of very few. External memory management has been seen as a way of meeting the needs of each individual application. Unfortunately, some of the systems offering external memory management have not gone far enough; for instance the Mach kernel still manages the physical frames, limiting applications to performing their own paging. The systems that allow the application to handle all aspects of its memory management provide a great deal of flexibility that could be exploited by developers to provide truly application-specific memory management. While this represents a real step forward in virtual memory management, it could be construed as defeating one of the points of virtual memory management in the first place: to free the programmer from the constraints of managing their own memory.

An ideal environment would consist of an operating system with application-specific memory management where that management would be handled by the operating system itself or a third-party external memory manager. The application developer could choose what components would be used to meet their specific requirements and the operating system would construct

the hierarchy appropriately. Furthermore, each component in the hierarchy could be parameterised to meet the specific requirements of each individual application. Such a hierarchy could call upon many mechanisms for managing virtual memory.

Chapters 3 and 4 discussed the issues in paging to the local disk and across the network to a remote host. Chapter 5 looked at a way of reducing the number of pages expelled to backing store by compressing them in memory. Each technique discussed offers differing performance improvements over the others. Furthermore, different combinations of these techniques offer differing improvements again. For instance, I showed that a key factor in remote paging is network bandwidth and that compressing pages could improve paging performance. Consequently, combining a compressed cache with remote paging has the effect, provided there are sufficient cache hits, of increasing paging bandwidth for free.

Techniques for improving page fault handling should not be exclusive. Many techniques benefit different applications in different ways and it should be the application developer who chooses which is best for his application. The operating system developer should not provide their “preferred” method of servicing page faults or impose a system wide policy for deciding which pages should be evicted to make way for new ones. The operating system developer is not in a position to make these decisions. However, neither should the operating system simply hand the problem over to the application developer. The choice between default memory management or write your own is, for most developers, no choice at all.

The next part in this dissertation describes the design and implementation of the *Heracles* virtual memory management system. Built on top of the Nemesis operating system, Heracles allows the application developer to specify a virtual memory hierarchy specific to their application. The choices available to the developer include using a compressed cache, a local disk or remote paging. The developer can further specify exactly how much physical memory their application requires, how much should be given over to the compressed cache, whether to buffer disk writes and whether to perform any page prefetching. Furthermore, the remote paging can be augmented with compression, to save bandwidth and space, and local or remote mirrors in order to survive server failure. The entire system is defined in an interface definition language and developers, should they wish, can extend individual components and/or provide their own implementation. The system is, to an extent, reconfigurable at run-time in that the size of the compressed cache can be increased or reduced from within the application. Importantly, different regions of an application’s virtual address space can have different hierarchies associated with them.

Before going on to describe Heracles, we will first look at the platform on which it was built. The Nemesis operating system is described in the next chapter, focusing on the aspects important to understand in order to appreciate the Heracles VMM system.

# Chapter 7

## Overview of the Nemesis Operating System

*The translation of ideas into action is usually in the hands of people least likely to follow rational motives. Hence, it is that action is often the nemesis of ideas, and sometimes of the men who formulate them.*

**Eric Hoffer.**

The emergence of 64-bit architectures has prompted interest in the development of single address space operating systems (SASOS). Some of the benefits of a single address space include lower context switch time, reduced data copying and a rich sharing environment.

Modern attempts to exploit these features include Angel [WMSS93], Opal [CLBHL93] and Mungi [HEV<sup>+</sup>97]. The designers of these systems were motivated by the possibility of such a large address space providing the impetus for distributed shared virtual memory (DSVM) and persistent memory mapped objects. Nemesis [Ros95], on the other hand, considers these issues to be the domain of the application developer. Its focus is on providing the mechanism for a rich and highly specialised environment where the operating system domain is responsible for the secure multiplexing of resources in a QoS framework and the application domain is responsible for everything else.

## 7.1 Structure

The vast majority of code traditionally executed by the operating system on behalf of an application need not be executed in a different protection domain. Code that accesses and updates important shared data structures is usually executed infrequently and is usually associated with out-of-band operations like opening and closing a file [Bar96]. Nemesis exploits this to move the majority of operating system functionality into the application domain. The result is a vertically structured operating system (figure 7.1). This structure lends itself to more accurate accounting as each domain is “billed” not only for the execution of its own code but also for execution of OS code. In traditional systems, and in systems which provide shared servers, the OS would perform many functions on behalf of a process making accounting difficult.

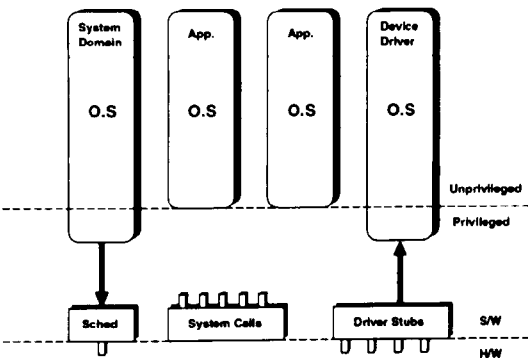


Figure 7.1: Nemesis Structure.

## 7.2 Quality of Service

Nemesis offers domains the ability to acquire resource guarantees from the system. This means that applications can specify the amount of CPU and disk bandwidth they receive. For instance, an application may specify that it would like a 1ms CPU slice every period of 10ms (thus acquiring 10% of CPU bandwidth). Furthermore, it may state that it requires a slice of 100ms out of every 800 from the disk driver (thus acquiring 12.5% of disk bandwidth).

This ability to specify the parameters within which an application runs provides a great deal of predictability. An application running at different times, with different system loads, but with

the same QoS parameters will behave in almost exactly the same way each time. There will be variances due to factors outwith the system’s control: such as rotational latency in the local disk or the time taken for the disk head to reach the correct track.

### 7.3 Virtual Memory Management

In terms of memory management, Nemesis is responsible for the allocation of virtual and physical addresses. The application domain can set up its own protection and virtual address mappings, and has to handle its own memory faults.

Nemesis splits the address space into sections called stretches [Han99, Han98]. A stretch is an abstraction over a contiguous region of the virtual address space where every page has the same access rights. A stretch cannot shrink or grow once created and different stretches cannot overlap. A stretch is only meaningful when bound to a stretch driver. Any attempts to address memory in an unbound stretch will result in an un-resolvable page fault.

The stretch driver handles page faults, implements the replacement strategies, and performs virtual to physical mappings. The closest analog to a stretch driver is a memory object in Mach [RTY<sup>+</sup>88]. Figure 7.2 shows the relationship between a stretch and stretch driver and the rest of the system (from [Han97]).

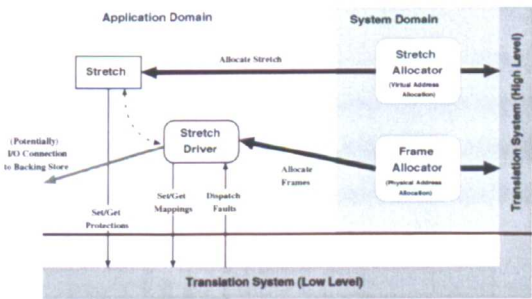


Figure 7.2: Nemesis VM System Architecture

On a page fault, the NTSC<sup>1</sup> sends an event to the appropriate domain. When that domain is reactivated it should resolve the fault. This may involve replacing a page currently in physical

<sup>1</sup>Nemesis Trusted Supervisor Code: equivalent to an extremely small kernel.

memory – the method used to do this is entirely the responsibility of the stretch driver for a particular domain.

Note that unlike the  $\mu$ -kernel model page faults are not serviced by “external” pagers, but are rather satisfied “internally” within the application’s domain. This method of paging is known as self paging (figure 7.3) and Nemesis can be said to support “internal” virtual memory management.

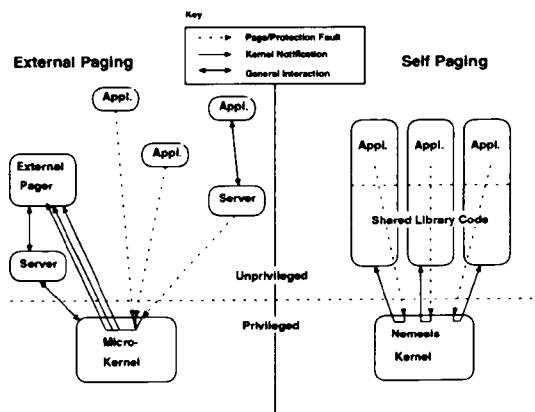


Figure 7.3: Self Paging versus External Paging

7.4 Devices in Nemesis

Nemesis device drivers minimise QoS-crosstalk by separating the data and control paths [Bar96]. Figure 7.4 shows the architecture for a device driver under Nemesis. Nemesis places additional demands on device drivers compared to those under traditional systems<sup>2</sup>:

- Drivers do not hide the shared nature of the underlying physical resource but instead provide explicit control over the multiplexing of that resource.
- Applications need to be aware of the current level of resources to which they have access. Negotiated QoS-guarantees are provided to each client of the driver.

<sup>2</sup>Taken directly from [Bar96].

- There are simple and effective feedback mechanisms to allow an application to monitor its progress and adapt its behaviour in light of the rate at which I/O requests are being serviced.

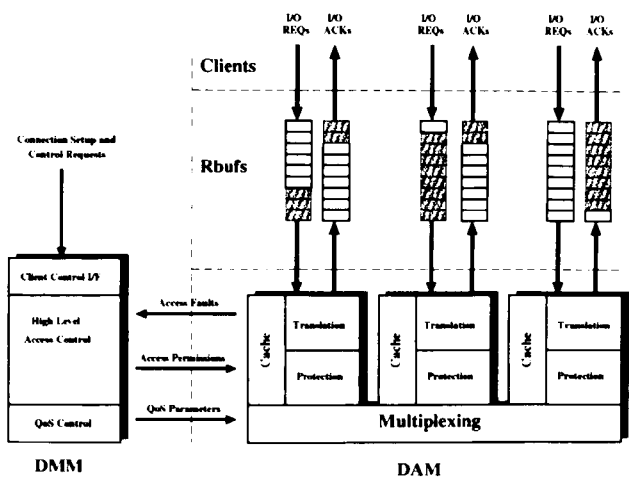


Figure 7.4: Nemesis Device Driver Architecture

The architecture comprises of two components: the Device Abstraction Module (DAM) and the Device Management Module (DMM). The DAM resides on the I/O datapath and contains the minimal functionality to provide secure user-level access and support for QoS guarantees. The DMM supplies out-of-band control and is **never** involved in in-band operations. It is responsible for the setup of new connections and for adjusting the QoS of existing connections. Thus, the more frequent in-band operations are subject to QoS guarantees and the servicing of one client has no effect on the servicing of others.

7.5 Sharing in Nemesis

The entire Nemesis system is defined using an interface definition language (IDL) and all interfaces are strongly typed. This has led to a programming paradigm where invocations across an interface pass a reference to the interface as the first argument, known as a closure pointer. This closure pointer constitutes a pointer to the method table for that interface and a reference to per-client state (which is unavailable to the client) as shown in figure 7.5. The closure pointer, along with the other arguments, represent the calling environment [Ros94].



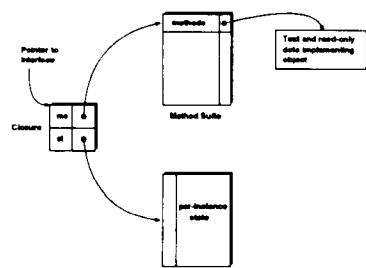


Figure 7.5: Module Interface Data Structures in Nemesis

This method of programming allows modules<sup>3</sup> to be shared without the need for synchronisation techniques or convoluted hacks to try and get round the problem of static data. For instance, Opal [CFL93] allows domains to attach to only one application module that defines static data because the static data is stored as an offset from a Global Pointer (GP) register. The difficulty of coordinating such offsets proved prohibitive in allowing more sharing.

## 7.6 Nemesis IO System

The design for high volume packet-based data transfer in Nemesis is based on the Rbufs scheme in [Bla95] (shown in figure 7.6).

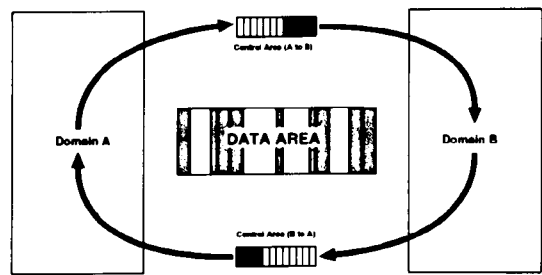


Figure 7.6: High Volume I/O using Rbufs

The data area consists of a small number of large contiguous regions backed by physical memory. Protection of the area is determined by use: the sender must have at least write permissions and the receiver at least read permissions. The regions can be grouped together using a data

<sup>3</sup>A module is just a piece of code with no external references and no internal shared state.

structure known as an I/O Record, or **iovec**, which consists of a header followed by a sequence of (base,length) pairs.

The control area is a circular buffer in a producer / consumer arrangement. It is used to transfer **iovec** information where the sender is responsible for populating the area the **iovecs** refer to and the consumer is responsible for returning **iovecs** referring to the consumed areas to the sender.

### 7.6.1 The Nemesis QoSEntry

I/O channels can have a scheduling policy attached to them in the form of a QoSEntry. A QoSEntry allows quality of service to be defined for each I/O channel in the form of a slice  $s$  every period  $p$  of time. Additional parameters indicate whether the I/O channel will accept extra time and the amount of latitude that should be afforded clients with poor blocking behaviour. The QoSEntry interface is shown in appendix F.

On a call to “Rendezvous” (inherited from an IOEntry) an I/O channel is selected according to its QoS parameters and its used time. This channel is then serviced and charged for the service time via a call to ‘Charge’. The scheduling algorithm “operates using three internal queues of I/O channels - ‘Waiting’, for channels that have work pending but have run out of allocation, ‘Idle’, for I/O channels that have no work pending, and ‘Runnable’, for I/O channels that have work pending and have remaining time in their current period.” (taken from the QoSEntry interface).

## 7.7 Discussion

There are many reasons why Heracles was implemented in the Nemesis operating system. The Nemesis SAS model affords more potential flexibility for managing virtual memory management: different stretches of virtual memory can be backed by different stretch drivers, allowing many different policies to be implemented within the same application domain. Also, the Nemesis notion of internal virtual memory management means that it is possible to provide true application-specific memory management: different applications do not share the same third-party pager. The in-built support for quality of service makes it possible to provide applications with guarantees about access to resources. While this in itself is not seen as central to the Heracles VM system, it allows applications to be tested in the equivalent of a competitive environ-

ment: by limiting access to resources such as CPU and disk, the application can be measured as if sharing these resources with other applications.

The flexibility of Nemesis as well as the philosophy that the operating system should expose as much information as possible, instead of hiding implementations behind ill-fitting, generic interfaces, provides a powerful framework for experimenting with virtual memory management. The result of this experimentation is the *Heracles* virtual memory management system.

## **Part III**

# **The Heracles VMM System**

# Chapter 8

## Heracles System Design Overview

*A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.*

**Douglas Adams.**

I have discussed the relative benefits of compressed caching techniques in light of the growing disparity between CPU and disk speeds and concluded that, for some applications, the benefits of utilising a compressed cache can be quite significant. I have also looked at the problem of paging across the network to memory on remote hosts and similarly concluded that it can hold real benefits for the page fault handling performance of applications. In addition, I have briefly mentioned some mechanisms for improving the performance of a local disk by masking its inherent inefficiencies for paging. I have not, at any time, stated that any one of these methods is better than any other when it comes to managing virtual memory. Instead, I have tried to highlight the possible benefits that applications can get from each one. Further, I have stated that some applications benefit more from one technique than another and that no technique provides the best solution for all applications.

In this chapter I focus on bringing all these techniques together in a user-level virtual memory management scheme that allows the application developer/user to define the virtual memory hierarchy for individual applications. The result is the Heracles virtual memory management system. This caters to the requirements of individual applications and provides them with virtual memory management specific to their needs. The system is defined in an interface definition

language (IDL) allowing each component to be easily replaced should more specialisation be required.

Figure 8.1 shows the design of the Heracles virtual memory management sub-system, while figure 8.2 shows the possible hierarchy combinations. The focus of this is on the provision of a flexible user-level virtual memory management unit that allows the developer/user to define an application’s VM hierarchy.

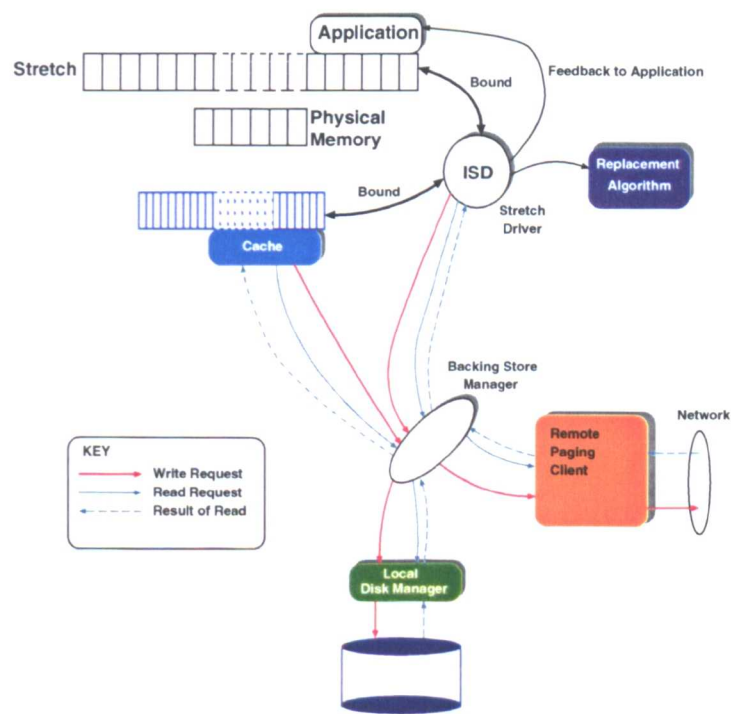
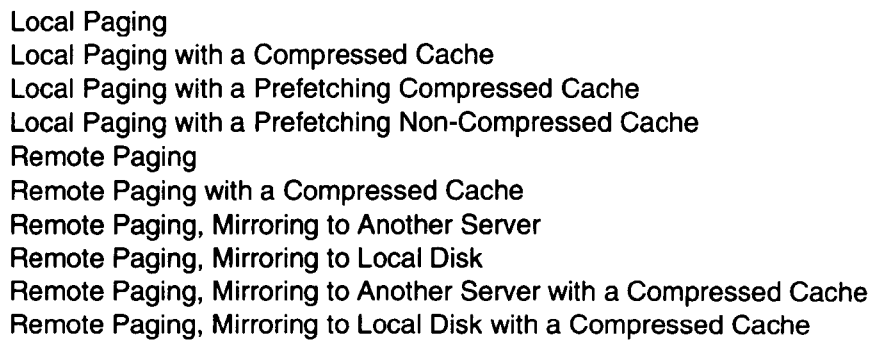


Figure 8.1: System Design

There now follows a bottom-up look at the individual components that make up the sub-system.

## 8.1 Local Disk Manager

There are three possible Disk Managers to choose from: the *basic disk manager* reads and writes pages to the local disk upon request; the *asynchronous disk manager* buffers write requests, emptying the buffer periodically, and performs reads on request; and the *prefetching disk manager* performs bulk reads and writes as well as single page read/write operations.



- Local Paging
- Local Paging with a Compressed Cache
- Local Paging with a Prefetching Compressed Cache
- Local Paging with a Prefetching Non-Compressed Cache
- Remote Paging
- Remote Paging with a Compressed Cache
- Remote Paging, Mirroring to Another Server
- Remote Paging, Mirroring to Local Disk
- Remote Paging, Mirroring to Another Server with a Compressed Cache
- Remote Paging, Mirroring to Local Disk with a Compressed Cache

Figure 8.2: VM Hierarchy Combinations.

The disk manager is responsible for acquiring space on the local disk, the placement of pages within that area, acquiring the appropriate QoS parameters, and, in the case of the *prefetching disk manager*, implementing the prefetch algorithm. Chapter 9 describes the disk managers in more detail.

## 8.2 Remote Paging

The remote paging subsystem includes a trader, which holds current adverts from servers; a remote paging server, which services requests for the storage and retrieval of pages; and the remote paging client, which acquires resources on remote servers and handles paging at the client side. Note that a host can become a server or a client at any time, depending on current commitments. There are no fixed servers that clients contact for resources.

The remote paging client can spread the application's pages over many servers and can employ mirroring to increase the application's tolerance to faults. The exchange of pages is built on top of the remote paging protocol (RPP) which ensures reliable delivery of data. The remote paging environment is covered in chapter 10.

## 8.3 Backing Store Manager

The *backing store manager* coordinates page reads and writes to the appropriate backing stores. This may be to the local disk and/or to the *remote paging client*. The *backing store manager* takes requests for page reads and page writes. Based upon the type of service it is providing (i.e. whether performing remote paging with local mirroring, remote paging with remote mirroring, remote paging only, or just paging to the local disk) it passes the request to the appropriate sub-component/s.

The backing store manager is also responsible for populating the local disk with pages revoked by remote servers. If this should happen, the backing store manager is passed a reference to a local disk manager by the stretch driver and instructed to populate it with the appropriate range of pages. The backing store manager forks a thread which, in turn, requests each page in the range from the remote paging client and passes them to the disk manager. When this process is complete, it informs the remote paging client which in turn frees the resources at the appropriate server.

While this population of the local disk is taking place, the backing store manager is also taking evicted pages from further up the hierarchy. If any of these pages fall in the relevant range, they are passed to the disk manager directly and are subsequently not requested from the remote paging client.

## 8.4 Cache Manager

The *cache manager* comes in three forms: the basic cache buffers writes and prefetching; the compressed cache compresses pages in an attempt to “extend” the size of physical memory; and the extended compressed cache is a combination of the basic cache with compression.

The basic cache manager splits its memory into two sections: the eviction area and the fetch area (where the fetch area maybe of zero size). On a cache write, the page is placed in the eviction area for writing to backing store. When the eviction area is full, all pages are written contiguously to backing store. On a cache miss, the page request is forwarded to backing store along with the size of the fetch area (if greater than zero). The backing store manager returns the faulted page and enough other pages in the same group to populate the fetch area. If the fetch area is



of zero size, a single read is performed directly from backing store to the appropriate location in memory.

The compressed cache manager takes evicted pages and compresses them into the cache. On a read, the cache manager decompresses the page directly to the appropriate location. When the cache is full, the appropriate number of pages are evicted to make way for new entries.

The extended compressed cache manager combines the ideas of the previous two. It splits the cache into three areas: a working pool, a fetch area and an eviction area. On a page write, the page is compressed to the cache. If the working pool is full, space is created by evicting pages to the eviction area. Once the eviction area is full, all the pages are written contiguously to backing store.

A full discussion about the design and implementation of the compressed cache managers is provided in chapter 11.

## 8.5 Page Replacement

The default page replacement algorithm for the Heracles VM system is a second chance FIFO implemented via a single hand clock. Research has shown (for example [LCC94]) that some applications, admittedly a small subset, can benefit greatly from implementing their own page replacement algorithm. An LRU algorithm is also provided but this performs rather poorly on Nemesis running on Intel. This is because accessing the page tables to update LRU information requires a system call. This greatly slows down the process of selecting victim pages for expulsion. In addition to these two algorithms, Heracles provides a page replacement interface that allows custom page replacement algorithms to be plugged in.

## 8.6 Stretch Driver

The *stretch driver* is the fulcrum of the virtual memory management system. It performs page mappings, handles page faults and controls the use of the physical memory. In the case where a cache is being utilised, the stretch driver allocates a range of physical frames to the cache and can alter this split at run-time.

The *stretch driver* is notified of page faults via a call from the memory management unit. The stretch driver is passed the faulting virtual address and is responsible for bringing that page into memory. Instead of dealing with the faults internally, the stretch driver passes the request for the faulting page to the appropriate sub-unit/s.

The application can interact with the stretch driver to lock down pages and subsequently unlock them, alter the size of its compressed cache and provide its own page replacement implementation.

The stretch driver knows exactly what components are used to make up an application's virtual memory hierarchy at any time.

## 8.7 Dynamic Behaviour Modification

Applications can choose to implement an *AppHandler* interface, which is passed to the stretch driver on instantiation. This allows the stretch driver to propagate changes in resource guarantees to the application. The application can subsequently adjust its behaviour to reflect these changes. For instance, an application requesting the use of the remote paging subsystem may find its resources revoked by servers. This in turn may affect its page fault handling rate. Should it choose, the application can decide to alter its behaviour, or its configuration, to reflect this change. For example, it may decide to increase the size of its compressed cache to reduce the number of cache misses.

## 8.8 Application Startup

When an application is loaded Nemesis sets up its environment: its virtual memory areas, stack and heap; its physical memory; its scheduling domain and its run-time context. This is controlled primarily by the *builder*. When the builder is setting up an application's virtual memory hierarchy, it looks up the application in the namespace for its virtual memory management preferences. It then instantiates the appropriate sub-units and passes them to the stretch driver.

The namespace configuration includes information on how much physical memory an application requires; how much, if any, of that memory should be allotted to the compressed cache; what

compression algorithm the compressed cache should use; what the quality of service parameters should be for local CPU, local disk, and remote paging; and what type of paging the system is performing: local, remote, local with prefetching, local mirroring or remote mirroring.

If desired, the application can choose to start up with no paging, i.e., with as much physical memory as virtual memory. This could be due to the fact that its memory requirements are reasonable, or that it wants to take more control over its virtual memory management. This control is covered next.

## 8.9 Application Control

The Heracles VM system is intended to provide sufficient flexibility to empower the application to control its own environment as much as possible. Each stretch bound to a stretch driver is subject to the hierarchy that stretch driver is employing. A feature of Nemesis is that an application can utilise as many stretch drivers as it deems necessary. This means that an application can set up a stretch of virtual memory with one instantiation of a hierarchy and another with a totally different one. This flexibility coupled with the Heracles VMM, gives the application tremendous control over its virtual memory management. For instance, the application developer may know that part of its virtual memory would benefit from a compressed cache but others would not. In this case, the application can set up a stretch of memory backed by a stretch driver utilising a compressed cache. This allows it to utilise its physical memory more effectively.

## 8.10 Extending Heracles

The Heracles VM hierarchy aims at providing the application with a rich choice of alternatives for constructing a customised hierarchy. However, there will be applications that benefit from providing more specialised solutions for a particular component of the hierarchy. The use of an interface definition language allows developers to provide a particular implementation of a Heracles component without the need to recompile the others.

Also, the design of the hierarchy is such that there is no inferred knowledge in a particular component about where requests come from or where they are serviced from. This allows the

components of the hierarchy to be used in novel ways. For instance, the remote paging client receives an instruction to store a page  $n$  of size  $s$ . The identifier  $n$  could be an arbitrary identifier for a piece of data that is  $s$  bytes in length. Thus, the remote paging client could just as easily be used for some remote caching scheme as it could for a remote paging scheme.

# Chapter 9

## Local Disk Manager

*I have not lost my mind - it's backed up on disk somewhere.*  
**unknown.**

The disk manager is responsible for acquiring space on the local disk and for the placement of pages on that space. It is also responsible for setting the disk QoS parameters. There are three types of disk manager available to the Heracles VM system: basic disk manager, asynchronous disk manager and prefetching disk manager. The basic disk manager performs single-page reads and writes to the local disk. The asynchronous disk manager performs single-page reads but buffers writes, emptying the buffer periodically. This disk manager is primarily intended as a backup method for remote paging. The prefetching disk manager supports single-page reads and writes but also supports bulk reads and writes.

### 9.1 Basic Disk Manager

The basic disk manager maintains a table of pages stored on disk. On a request for a page, it performs a table lookup to find the appropriate disk block. It then issues a request to the underlying device driver for the appropriate number of blocks. A page write involves allocating a disk block and issuing the request to the device driver.

## 9.2 Asynchronous Disk Manager

The asynchronous disk manager maintains a small buffer, the size of which can be determined by the application. On a page write, the buffer is checked for a free page. If a page is free the page is copied into the buffer. If there is no free page in the buffer, a page is expelled to make way. Although small, making a hit unlikely, the buffer must first be checked on a read request before it is forwarded to the device driver.

The asynchronous disk manger forks a thread on the first entry into the buffer. The thread awakens periodically and writes a page to disk. Once the page is written to disk, the space is freed for re-use.

As mentioned previously, the asynchronous disk manager was designed primarily to be used in conjunction with remote paging. By combining these two methods, pages are written to the local buffer before being sent across the network. When the thread blocks on the network interface, the disk manager thread chooses a page to expel from the buffer. Issuing a write request to the disk device driver causes the disk thread to block on completion of the write request, allowing the application thread to resume. By interleaving this blocking behaviour, it is possible to reduce the overhead of maintaining a local copy of pages to ensure recovery in light of a server crash.

## 9.3 Prefetching Disk Manager

The prefetching disk manager extends the basic disk manager by providing support for bulk reads and writes. It still supports single reads and writes but the assumption is that single writes will seldom be used. The prefetching disk manager hides the prefetching strategy employed from the cache layer above. Consequently, when the cache wishes to fetch a number of pages, it passes the faulting page, the address of the read area and its size to the disk manager.

In the case where the disk manager is used for performing bulk writes without any prefetching, the disk manager employs the friends algorithm, described below, to group the pages together. However, this is simply a matter of convenience and disk reads are performed as in the basic disk manager. The following section describes the algorithm used to determine which pages are fetched along with the faulting page.

### 9.3.1 The Friends Algorithm

The *friends* algorithm works on the basis of “pages evicted together are used together”. When a group of pages are passed to the disk manager for writing to disk, those pages become members of the same group. Members of the same group must always be stored contiguously on the disk. A page can leave one group and become a member of another group. If a page leaves to join another group, it must be removed from the old group. This has ramifications for that group’s structure. If the leaving page was at the start, or the end, of the group on disk, it can be removed easily and the appropriate disk space returned to the free list. However, if the page resided somewhere in between the first and the last page in a group, the group no longer meets the requirement of being contiguous. This causes the pages occurring after the leaving page to form a breakaway group (see figure 9.1).

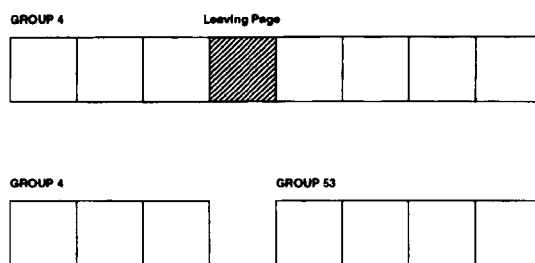


Figure 9.1: A Group Split.

On a request to read a group of pages from the cache manager, the disk manager is passed the faulting page and the size of the read area. The disk manager then examines the group to which the faulting page belongs. If the group will fit in the read area, the entire group is read from disk to the appropriate location. The cache manager is passed a list of page descriptors containing the virtual page number (VPN) and size (as pages maybe in compressed form) of each page. If the group is larger than the read area, a subset of the group is read from disk. When a subset of a group is to be read from disk, the disk manager chooses those pages nearest the faulting page to accompany it into memory.

Although the friends algorithm was shown to increase performance by up to a factor of two over demand paging, much of the benefit was derived from buffering pages in order to perform contiguous writes. However, the real benefit of prefetching was observed when used in conjunction with a compressed cache. When combined with a 2MB compressed cache, the prefetching disk manager increased the cache hit rate from 79% to 98% for the matrix multiplication program,

and from 45% to 85% for the filter program. The performance of the disk manager when used in conjunction with other components is further discussed in chapter 12.

## 9.4 Summary

The disk managers described here are presented as a means of highlighting some of the various possibilities available to the application developer. They are not meant to provide a solution for all cases, but rather to provide interesting differences in how paging to the local disk can be approached. The asynchronous disk manager highlights the possibility of adding fault tolerance to remote paging while incurring very little overhead. The basic disk manager is provided as a basic read/write model for applications that do not require more sophisticated paging strategies. This basic manager is extended to incorporate features such as contiguous writes and prefetching as a means of examining the possible performance improvements available to applications that choose a more sophisticated paging strategy.



# Chapter 10

## Remote Paging in Heracles

*The charm, one might say the genius of memory, is that it is choosy, chancy, and temperamental.*

**Elizabeth Bowen.**

This chapter describes the design of the Heracles remote paging system. Section 10.1 defines the protocol for the transfer of pages between hosts before going on to describe the participants that make up the scheme. There are three components that make up the remote paging system: the trader; the remote paging server; and the remote paging client. The trader, described in section 10.2, tracks the availability of memory in the distributed system. The remote paging server is described in section 10.3 and the client is covered in 10.4. Section 10.5 discusses how the problem of revocation can be resolved and the provision of QoS is described in section 10.6. This chapter concludes in section 10.7 with a description of how the remote paging system copes with server crashes.

### 10.1 Remote Paging Protocol

The Remote Paging Protocol (RPP)<sup>1</sup> is used to implement the passing of data between a client and a remote paging server. Although the transmission of data uses the unreliable delivery proto-

---

<sup>1</sup>Similar to that described by [CG90].

col (UDP), the RPP ensures that all data is delivered reliably. There are three allowed operations: page read, page write and page exchange.

Each packet transmitted between client and server, UDP header information aside, consists of a payload (part of a page) and a *request* block. The request block contains the following information:

```
REQUEST : TYPE = RECORD [  
    op    : Op,  
    vpn   : LONG CARDINAL,  
    size  : CARDINAL,  
    off   : CARDINAL,  
    dest  : LONG CARDINAL,  
    dsz   : CARDINAL,  
    tot   : CARDINAL,  
    num   : CARDINAL,  
    ack   : ACK ];
```

The *op* field informs the recipient what information is contained in the packet and can be either *read*, *write* or *exchange*. The *vpn* refers to the virtual page number and the *size* refers to the size of the payload in the packet. The *off* field informs the recipient of the offset from the start of the page to the start of the payload. The *dest* and *dsz* fields are used to denote the virtual page number and size of a page being returned in an exchange. The *tot* and *num* fields represent the total number of packets the page is split into and the current packet number. Finally, the *ack* field is used to piggyback acknowledgements (ACKs) on data packets. An ACK is identical in structure to a NACK and consists of two words. The first word contains the VPN and, in the case of the ACK, the second word contains an error code. The error code for an ACK is NONE.

Under the RPP, only full pages are acknowledged. When a page is transmitted, the sender expects to receive an ACK for the entire page — not each packet. This reduces the amount of information that has to be exchanged by both participants. Figure 10.1 shows the steps in a successful page write to a server. However, we must have a method of detecting the loss of individual packets. This is done by use of negative acknowledgements (NACKs). When the recipient of a page detects a missing packet, for example if it receives packets number one and three but not two, it sends a NACK to the sender. The NACK contains two words, the first of which is the VPN.

Because the packet number can be represented using only the bottom eight bits of the second word, the top twenty-four bits can be used to represent the type of error that occurred. Figure 10.2 shows what happens on a dropped packet during a page write operation.

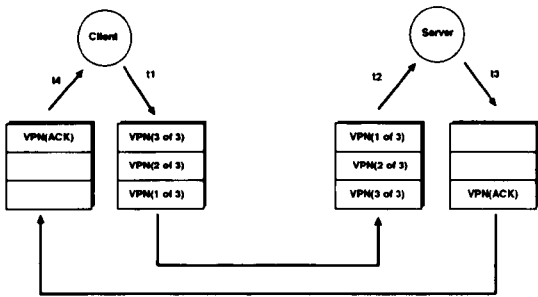


Figure 10.1: Successful Page Write.

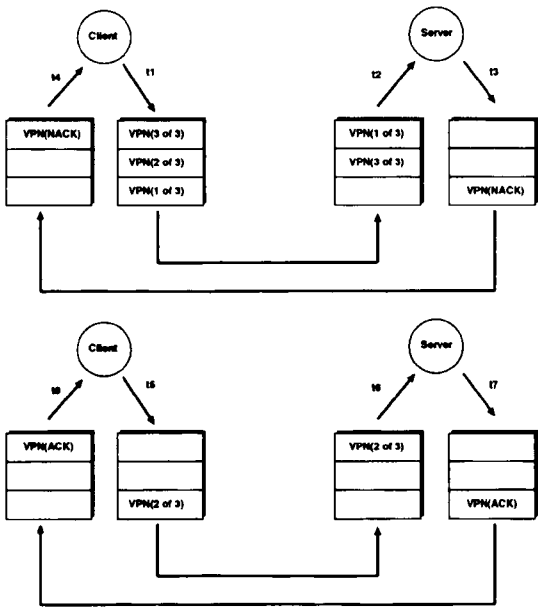


Figure 10.2: Dropped Packet on Page Write.

The RPP ensures that in the normal case, i.e., with no dropped packets, the overhead of transferring a page is one ACK. Only when packets are dropped do NACKs have to be transmitted.

Another way of handling dropped packets is to use a bitfield in the NACK which represents missing packets. Under this scheme, a single NACK could be sent for all dropped packets in

a page. For instance, if the receiver sees packets 1, 2, 5 and 6, it could send back a NACK containing a bitfield indicating that packets 3 and 4 were not received.<sup>2</sup>

### 10.1.1 RPP Operations

On a request to read a page, the client sends a packet containing the virtual page number (VPN) and the size of the data to the server. The server examines the request block and translates the VPN into a local address by way of a table lookup. The server then splits the page into multiples of the maximum transfer unit (MTU), minus the size of the UDP header field and the size of the request block. Each packet is stamped with the relevant information and transmitted to the client. The server then awaits an acknowledgement for the **page**. Note that if the client does not receive any data within a certain period of time, it assumes its request has not reached the server and retransmits it.

The steps in a page write request are exactly the same as those in a page read except the roles are reversed: the client breaks up the page, transmits it and awaits acknowledgement.

A page exchange is simply an optimisation of a page write followed by a page read. On an exchange, the client performs a page write as normal but it also includes information about the page it wants returned. The server, upon receiving the last packet for the page sent to it, then sends the page the client requested as per the page read operation. The only difference is that the server also stamps the request block with an ACK for the page it received.

There are three main components that make up the remote paging environment: the trader; the client; and the server.

## 10.2 Remote Paging Trader

The remote paging trader is responsible for maintaining information about servers advertising free memory. When a client wishes to find a server it contacts the remote paging trader with information about how much physical memory it requires. The trader maintains a list of servers, ordered by amount of physical memory, and returns up to the first five servers advertising enough

---

<sup>2</sup>Thanks go to my external examiner for this suggestion.

frames (section 10.4 explains why). Once the client receives this information, it can then contact the appropriate server/s to arrange suitable resources (see figure 10.3). The trader only keeps information on current adverts for free memory, it is not involved in the acquisition of resources. This process is carried out entirely between a client and a prospective server.

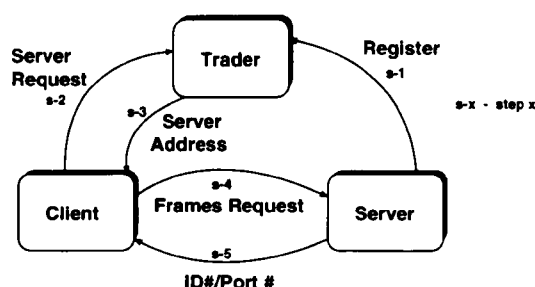


Figure 10.3: Setting up a Server Link

### 10.2.1 Locating the Trader

There are currently up to three traders on the local area network at any one time and servers send their advert to all three.<sup>3</sup> This is a probability based scheme. The probability that all three traders will be down at once is considered extremely low. Furthermore, due to the fact that the trader is not directly involved in the paging process (although having no trader means that clients cannot lookup servers) it does not affect existing arrangements. Although a more complicated and robust (with respect to the availability of a trader) election scheme was considered, the trade-off between its complexity and its potential benefits were not deemed to be worth the effort.

Currently, the traders reside at well known addresses and a client can request the location of a server from any one of them. Note that because the trader only tracks current adverts and does not keep any information about server commitments, consistency is not considered a problem. A server assesses a client's request for resources based on its current commitments, not on what the advert held at the trader was when the client contacted it. Therefore, the advert held at the trader can be considered more an indication of available memory rather than an absolute.

A trader can become unreachable from a server due to a number of reasons: network congestion; process failure; or machine failure. If a trader is deemed to be unreachable from a server it

<sup>3</sup>Under the current implementation, the adverts consist of three simultaneous unicasts.

does not necessarily mean that that trader is not running. Consequently, the next time the server polls its local resources, it will attempt to update the previously unreachable server, regardless of whether its advert has changed.

From the perspective of a client, if a trader is seen to be unreachable, the client will simply try another. If no trader is available, the client must utilise another method, e.g., paging to the local disk.

## 10.3 Remote Paging Server

The remote paging server (RPSrv) is separated in the same way as normal device drivers under Nemesis (see 7.4). The device management module, represented by the interface *RPPCtl* (appendix C), provides the mechanisms for setting up a connection, via remote procedure call (RPC), to a paging server and adjusting the QoS parameters. The device abstraction module is solely responsible for dealing with page fetch/store requests from the remote paging client. The QoS parameters are in the form of a slice  $s$  every period  $p$  with an additional flag stating whether the client will take additional *slack* time.

### 10.3.1 Server Control Path

On a call to *CreateStream*, the paging server instantiates the necessary I/O components for passing pages between itself and the client and allocates physical memory for their storage. The server returns an id for the client and the UDP port on which it is bound. The client can then set up its end of the I/O channel using the port number passed to it by the server. Apart from freeing the resources, adjusting QoS parameters and resource revocation (all relatively infrequent occurrences), all other interaction is via the I/O channel.

When a client has acquired resources, it can request further resources by calling the RPSrv directly, i.e., without performing a lookup via the trader. These further resources are subject to the same QoS guarantees agreed on when the initial guarantees were acquired. Of course, the client can, should it wish, request to have the guarantees altered at any time.

When a request from a client arrives, the RPSrv checks the request against current commitments to determine if the request will be accepted. If the request involves additional memory for an

existing client, the RPSrv only examines current memory commitments. If the request is for a new client, the RPSrv also checks the CPU and network commitments. If these commitments do not allow the RPSrv to meet the client's request, the request is denied.

### 10.3.2 Server Data Path

When a client has acquired memory guarantees from a server, the pages, and requests for pages, are transferred in UDP packets in accordance with the RPP. The thread servicing requests for pages sits atop a QoSEntry (see section 7.2 for a description and appendix F for the interface). Requests are passed up by the QoSEntry according to each client's scheduling parameters.

On receiving a packet, the request block is examined to determine the nature of the transaction, i.e., whether a read, write or exchange. A page exchange is equivalent to a write followed by a read.

**Page Read:** On a read request, the server translates the VPN into an address within the stretch of memory being used to hold the client's pages. The page is then broken down to be transferred across the network in packets. When the last packet has been sent, the server waits on an ACK from the client (as described in section 10.1). If the translation does not yield an address, the server sends a request block with a NACK for the requested VPN.

**Page Write:** On a write request, the server translates the VPN into an address. If it is the first instance of that particular page, the server allocates a "page" for its storage.<sup>4</sup> The page is then copied to the address plus the offset from the start of the page (which may be zero). If the packet is marked  $n$  of  $n$ , and all packets up to  $n$  have been received, an ACK for the page is returned to the client. However, if the packet is marked  $n$  and  $n-1$  has not been received, the server sends a NACK for the missing packet.

### 10.3.3 Advertising Threshold

On startup, the remote paging server forks a thread that keeps track of current system commitments and previous commitments in an attempt to determine how much of its resources should

---

<sup>4</sup>Note that the page size at the client may be different from the server. Thus, allocating a page refers to allocating an area equivalent to the size of a page at the client.

be advertised with the remote paging trader. The period between these checks is currently set to 10 seconds but this can be altered to suit the needs of an individual server. The algorithm for determining how much physical memory a server should advertise is shown in figure 10.4.

```
if(total_committed_memory > MAX_THRESHOLD)
    revokeClient();
else if((total_committed_memory <= THRESHOLD) &&
        (free_mem >= prev_free_mem) && cpu_committable)
    if((total_committed_memory + amount_advertised) <= THRESHOLD)
        increase_advert(10%);
    else
        decrease_advert(5%);
else if((prev_free_mem > free_mem) || (prev_cpu > curr_cpu))
    decrease_advert(20%); /* back-off */
```

Figure 10.4: Algorithm for Monitoring Local Memory Commitments.

The algorithm is designed to give local domains priority over remote paging clients. If the total memory usage rises above `MAX_THRESHOLD` then we revoke memory from a remote client (see 10.5). Note that increasing the size of the advert does not affect the `free_mem` or `curr_cpu`, only a client acquiring resource guarantees, or a local domain starting up, can do this. An important point to also note is that a connection by a client can never cause another client to be revoked. Only local domains starting up can cause this. This is because the algorithm is designed to balance memory usage and memory advertised, keeping the combined total around the `THRESHOLD` mark: more specifically, an advert cannot cause this total to rise above `THRESHOLD` plus 10%.

The algorithm itself is not the mechanism by which requests are granted; each attempt to acquire resources causes a separate check on system commitments. Instead, the algorithm represents an attempt to advertise as much memory as possible while maintaining “headroom” for local domains to start up. If the current memory usage is more than it was the last time the algorithm was run, we back-off on the basis that this *may* be due to a spurt of local activity.



Another aspect of the algorithm is its monitoring of CPU usage. The `cpu_committable` flag is the result of looking at CPU guarantees for local domains and the scheduling commitments of the remote paging server and determining whether there is still enough available to take on another client. As is the case for rising memory commitments, if there is a change in local guarantees, the server reduces its advert.

10.3.3.1 Analysis of Advertising Threshold Algorithm

Table 10.1 shows the memory startup costs of some typical UNIX applications. The figures represent the amount of physical memory being used by each application within five seconds of being started on a PC-based machine running Linux version 2.2.12-20. The values were obtained using the command `'cat /proc/<pid>/statm'`. These costs only represent the amount of memory to start the application in physical memory, they do not represent the ongoing memory consumption of each application. Note also that the memory consumed by shared libraries is included in the cost. If all these applications were started within the same ten second period, the initial memory consumption would total around 28MB.

Application	Startup Cost (MB)
Netscape 4.7	12.58
Gnuplot 3.7	1.16
Emacs 20.3.1	4.91
Tgif 4.1 (patchlevel 16)	1.52
Xterm	1.66
Python 2.1	1.35
Perl 5.005_03	0.11
Find	0.45
mpg123 0.5q	2.02
ghostscript 5.10	2.30

Table 10.1: Memory startup costs of ten typical UNIX programs.

Table 10.2 shows some common memory sizes for desktop computers and the amount of memory represented by 10, 20 and 30% of the total available. If we consider this data in conjunction with the application startup costs shown in table 10.1, we can see that the value of `MAX_THRESHOLD` must necessarily vary with the total amount of physical memory on a given

host if we are to maintain reasonable headroom. In the case of a host with only 64MB of RAM, we may wish to set the MAX\_THRESHOLD value as high as 50%. While, in the case of a host with 512MB of RAM, 10% would be more than enough for most casual use. Note that the value of THRESHOLD need not be set as this is determined by the value of MAX\_THRESHOLD, i.e., MAX\_THRESHOLD - 11%. Recall that this prevents resources obtained on behalf of remote clients from causing memory utilised by other clients to be revoked.

Memory Size (MB)	10% (MB)	20% (MB)	30% (MB)
64	6.4	12.8	19.2
128	12.8	25.6	38.4
256	25.6	51.2	76.8
512	51.2	102.4	153.6

Table 10.2: Possible MAX\_THRESHOLD values as a percentage of physical memory.

Let us consider a distributed system of 200 hosts where each machine has at least 512MB of RAM. If we set MAX\_THRESHOLD at 10% for each, there may be as much as 10GB of memory that will never be advertised for use as a backing store. However, each machine is capable of advertising up to 460MB (minus memory reserved for the OS and any running daemons). Let us assume, for the sake of argument, that at any time, some of the machines are idle, some are being used heavily and others are utilising half of their available memory. This means that, on average, there will be around 40GB of free memory available as backing store. Furthermore, each host is capable of starting all ten of the UNIX applications entirely in physical memory well before relying on memory being returned from client’s whose resources have been revoked.

In the case where all machines have only 64MB there will be less available memory at any given time but the MAX\_THRESHOLD value will be set to reflect the lower memory capacity of each host, thus still affording hosts advertising free memory enough headroom to cope with a spurt of local activity.

Note that although the MAX\_THRESHOLD value could be deduced on a reasonable guess basis, it would be much more flexible to have this as a parameter. Thus, the value could be altered to afford more, or less, headroom according to a host’s particular requirements.

## 10.4 Remote Paging Client

The remote paging client (RPCInt) is responsible for acquiring resources on remote servers, handling page store/fetch requests and negotiating the appropriate QoS guarantees with the servers. The RPCInt is also responsible for handling resource revocation, fault-tolerance, reliable delivery of data and, where appropriate, compression and decompression of data.

On a call to the RPCInt's initialisation method, it is passed the start address and number of pages in the stretch on behalf of whom it is acting. In addition, the RPCInt is also passed a flag to inform it whether to provide fault-tolerance in the light of a possible server crash.<sup>5</sup> Currently, this consists of simply mirroring pages to another server. A parity scheme was considered but was rejected on the grounds that it offered, if any, very little benefit over mirroring and yet required more management.

The RPCInt also provides an interface to the server to allow the server to revoke resources and to inform the RPCInt when it has off-loaded the client's pages to another server. What happens when remote memory is revoked is discussed in detail in section 10.5.

When a client requests a suitable server from the trader, it is returned a list of potential hosts. This allows the client to make an informed decision based on what resources it already holds. For instance, a client may be employing a suite of servers to provide backing store for different stretches of memory. It may be doing this to help increase throughput or reduce the possible effects of revocation. Currently, the RPCInt is informed via a flag whether to attempt to spread the load or not. When the client receives the list of potential servers, it can reject those it already has a connection to. If the client is not attempting to spread the load, it can simply contact an existing server requesting additional resources without again contacting the trader.

As mentioned previously, a server can refuse a client's request for resources. If this should occur, the client will then attempt to acquire resources on another server, repeating this process until a server is found, or the list of servers is exhausted. If the client cannot find a suitable server, it will contact the trader for suitable servers once more before giving up. If the acquisition of a server is unsuccessful, the client will return the value *false*, informing the backing store manager that it should attempt to acquire local resources.

---

<sup>5</sup>This is a separate issue from mirroring to the local disk; that decision is taken further up the hierarchy and the remote paging client is unaware whether a local disk is being used or not.

If the client cannot make contact with a trader and it does not already have a connection to a server, it cannot acquire resources and returns *false*. If the client already has a server connection it will attempt to increase the amount of memory reserved for its use.

### 10.4.1 Page Compression

A new `RPCInt` can take, as an argument, a reference to a compression interface. This is used to compress data as it is sent to the server and decompress it when it arrives back. The viability of compression is obviously affected by the network bandwidth and CPU availability. The time taken to send a compressed page and receive an ACK is twice as fast compared to an uncompressed page on a 100Mb Ethernet. However, the compression requires CPU bandwidth, and if this is at a premium could cause the overall performance to degrade.

A typical compression ratio of 2:1 has the effect of reducing the number of payload packets from three to two per page (for an Intel host where page size is 4KB). This has the additional effect of reducing the time taken to copy the page from the receive buffer to the appropriate location in memory, thus reducing servicing cost and increasing the effective QoS rate at the server.

## 10.5 Resource Revocation

If a remote paging server decides that a client's resources must be revoked (due to an increase in local demands for resources) it contacts the client and informs it of its decision. The client can then attempt to locate other resources for the storage of the revoked pages. If this new resource is in the form of another remote server, the client informs the old server of the new location and requests that the pages be forwarded there.

The process of selecting a stretch of memory to be revoked consists of up to five rounds:

- least-recently-made-contact
- clients with local disk
- server versus mirror

- QoS weighting
- memory weighting

The first round of checks attempts to single out clients that have not made contact for a long time. Currently, this “long time” is set to sixty seconds, with an additional weighting of “times two” for diskless clients. If the first round produces more than one possible client then the least-recently-made-contact client is selected for revocation. If no possible victims are found in round one, the server proceeds to round two. Note that it is still possible for a diskless client to be selected in round one. However, the server shows preference to such clients in the form of the “times two” weighting.

In the second round of checks, the server removes all diskless clients from the list of possible victims and proceeds to round three. Round three consists of checking the role the server is playing in each client’s page fault handling scheme. The server attempts to find a client on whose behalf it is acting as a mirror of another server. Round four consists of examining the QoS parameters for each client. Clients running with only extra time, i.e., with no set guarantees, are assumed to be less likely to be affected by the revocation. The final round, and that used if rounds two, three or four do not produce any suitable victims, consists of selecting a client based on the amount of memory being held on its behalf. Currently, the client that is using the most memory is revoked. If, after revoking a client, there is still a need to free memory, the process is repeated.

The reason that the client using the largest amount of memory is selected is due to a pessimistic outlook on what is about to happen next. Only a local domain starting can cause a client to have its resources revoked. Consequently, the server assumes that there is about to be a further spurt of local activity and so tries to free as much memory as possible. Note that there is a certain in-built assumption here. There is no need, under Nemesis, to have a single large heap like there is in UNIX. It is perfectly feasible, in Nemesis, to instantiate different heaps for different purposes. As a consequence, it is not expected that heaps will be in the region of 100s of megabytes; rather they are more likely to be 10s of megabytes, or smaller, in size. However, there are many possibilities for selecting a client to revoke and the possible impact of this algorithm on a large system will need further investigation (see section 12.5). One particular problem that could arise is that a distributed system operating near the threshold could thrash, passing stretches from one host to another. This is considered an unlikely scenario as a client that has its resources revoked would, especially in the case where the stretch is large, most likely not find a server willing to

take its pages. Each server controls how much memory it allocates for clients use. If a server is advertising  $b$  bytes of memory, it will not permit a single client, or multiple clients, to overextend how much memory is reserved for remote clients.

However, if we consider a distributed system as it approaches full load, we can assume that there are some hosts acting as backing store for others and that those clients will have their resources revoked as a result of increased local activity. This means that it is possible, if each host's local activity increases gradually before the next host's activity increases, for a client's resources to be bounced from one server to another, increasing the load on the network. However, due to the nature of the remote paging system and the Nemesis operating system, the effect that this behaviour has on remote paging clients and other applications is actually quite small.

When a client has its resources revoked, in order to acquire new resources on another server, that server must have enough free memory, an under-utilised CPU and available network access. This means that a server offloading pages to another server does so within the constraints of all the resources agreed with the client. During the offloading, the client knows exactly where the most up-to-date version of a page resides and so never needs to request a page from more than one server. Indeed the client sees no appreciable performance penalty. The client cannot in itself cause that server to offload another client or penalise a local domain by utilising resources it is relying on. Furthermore, when any server admits a client, it immediately backs off from its advert and will not allow a further client to overextend its capabilities until the next advert is placed.

The problem could arise if a client is revoked from one server, acquires resources on another and is then revoked from the new server. If this happens, the client must back off from remote paging and resort to using the local disk. If it was already mirroring evicted pages to the local disk, it can simply inform both servers to free resources held on its behalf. Otherwise, it must take the pages back from the second server. It cannot take the resources back from the first server as this would cause the first server to be involved with the client for more time than it need be. The first server continues to offload to the new server, which continues to return the pages to the client.

The revocation scheme used in the Heracles remote paging system is active participation revocation (see section 4.2.7). This provides the client with the maximum control over the placement of its pages. For instance, the client may be mirroring its pages to another host and, as the server does not have this information, allowing the server to locate a new host may leave both sets of pages on the same host.

On notification of resource revocation, the RPCInt first attempts to acquire another server to hold the revoked pages. If a suitable server can be found, the client replies to the revocation notice with a request to hold the pages for offloading. The server marks the client as “being offloaded” and sets a timer for the offloading process. It then contacts the client for the address of the new server. When it has this, it sets up a connection to the new server and starts transmitting the pages. Note that the client could return the address of the new server on its reply to the revocation notice, reducing the need for the server to contact it once more.

Once the client has acquired a new server, it marks the status of the current server as “offloading” and stores the address of the new server. On subsequent page evictions, pages are sent only to the new server. The client maintains a temporary data-structure for pages sent to the new server. On a page request, the client examines this data-structure for the appropriate VPN. If the VPN is present, the request is forwarded to the new server, otherwise it is sent to the old server.

During the offloading process, the new server accepts pages from the old server. When it receives a page, it checks to ensure it has not already received a copy from the client. If it has already received the page, the new server simply discards the one received from the old server.

When the offloading is complete, the old server contacts the client informing it of the completion. The client calls the new server informing it that the process is complete. It then switches over to using the new server entirely and the resources on the old server are returned to the system.

If a new server cannot be found, the RPCInt contacts the stretch driver to inform it of the revocation. The stretch driver is aware of all the resources being utilised to meet the client’s needs and can make an informed decision based on this. For instance, if the pages were being mirrored to the local disk the stretch driver can inform the RPCInt to discard the relevant resources and switch to paging from the local disk.

Note that offloading pages on a client’s behalf is considered part of the “deal” of accepting requests for resources. Furthermore, Nemesis assumes that all parties behave in a cooperative fashion.

## 10.6 Quality of Service

An RPCInt is given QoS parameters for access to the local network interface in the form of  $x$  amount of time every  $y$  period. It is also passed the QoS parameters that should be given to the remote paging server. This is in the same form as the QoS at the client side and tells the server how much processing time every period that this client should receive. The server then processes requests according to these parameters and every client is charged for the processing time at the server. The client must also provide the server with information regarding the role the server is playing, i.e., a server or mirror, and inform the server if it has a local disk.

## 10.7 Handling Server Crashes

An application that chooses to utilise the remote paging system can currently choose between mirroring to the local disk or to another paging server (or not mirroring at all). All page-outs go to both the server and the mirror. If on requesting or sending a page to a server, the client receives no response, it retries a number of times before deciding that the host is unreachable. When a host is deemed unreachable, the client must take steps to again be in a state where it can cope with a further crash.

A server can be deemed unreachable if it crashes or if requests timeout due to network congestion or failure. The client has no way of knowing which of these is the reason for the inaccessibility of a server, it only knows it has become unreachable.

If an RPCInt using remote mirroring decides one server (server  $s_2$ ) is unreachable, it contacts the trader for another server (server  $s_3$ ). The RPCInt then acquires the appropriate resources from  $s_3$  and informs the existing server (server  $s_1$ ) to send its pages to the new server. In the meantime, the client sends evicted pages to both  $s_1$  and  $s_3$  and pages in only from  $s_1$ . Note that, unlike revocation, the client does not need to be informed when server  $s_3$  is up-to-date as it will only be paging in from server  $s_1$ .

If an RPCInt cannot acquire sufficient resources, it will issue a call to the stretch driver informing it that a server has become unreachable. The stretch driver will then contact the backing store manager to obtain local disk space to mirror the appropriate pages. When the backing store manager has initialised the disk, it forks a thread that, in turn, requests each page from the remaining



server and places it on disk. Note that this can have an effect on the application's performance, especially if it has QoS guarantees and no "slack" time. Thus, during the population phase, the RPCInt attempts to increase its guarantees at the server to reduce the impact on the application. Of course this request can be refused. However, as the client has stated that fault tolerance is important, it must be willing to make the necessary performance sacrifice if required.

While populating the local disk from a remote server, all page evictions are mirrored to the local disk on the way out. The backing store manager must keep track of which pages have been placed locally to prevent a request being issued for those pages. In all cases, pages evicted via the stretch driver are considered more recent than those on a remote server. Thus, should a page be received from the server that has since become resident on disk, due to being evicted, that page is simply discarded.

If a client has no local disk and cannot acquire further resources, it cannot get into a state where it can survive a further server crash. In this case, the client will periodically lookup a new server and populate it when it can. In the meantime, it continues paging as normal from the remaining server.

An important point to note about handling server crashes is that a client can be using many servers and many mirrors. If this is the case, it only seeks to find a server capable of taking those pages currently residing at one location.

In the case where a client is mirroring evicted pages to the local disk, it can simply switch over to using the disk for paging. In the meantime, it can attempt to obtain a new server and populate it from disk. Once the new server is up to date, the client then switches over to using the server, still mirroring evicted pages to the local disk.

One of the effects of a server becoming unreachable, due to network congestion or failure, is that memory reserved for clients who have deemed a server unreachable is not returned to the system. The implementation of the revocation selection process is seen as a way of cleaning up such memory. Clients that have not contacted the server for sufficient time are considered for revocation first.

# Chapter 11

## Cache Manager

*The English language is rather like a monster accordion, stretchable at the whim of the editor, compressible ad lib.*

**Robert Burchfield.**

Heracles takes the software approach to using a compressed cache. Memory is split into two chunks: one chunk consists of the active pool of pages; the other is used as a cache into which pages expelled from the active pool are compressed instead of being sent to backing store (see figure 11.1). Only when pages are expelled from the cache are they written to backing store. The idea here is to obviate the writing of pages that may be required again soon to backing store.

There are three types of cache manager available to the memory sub-system: the *basic cache manager*, the *compressed cache manager* and the *extended compressed cache manager*. The basic cache manager is used as a means of employing multiple simultaneous page writes and possibly group-based prefetching (described in section 9). The compressed cache manager compresses pages in an attempt to 'extend' the size of physical memory. The extended compressed cache manager represents a combination of the other two, utilising both compression and contiguous writes, possibly with group-based prefetching.

The physical memory allocated to the cache is managed by the stretch driver. Figure 11.2 shows the layout of the cache. The mapped pointer indicates the start of the cache backed by physical memory. An important point to note is that the physical frames are always contiguous. The

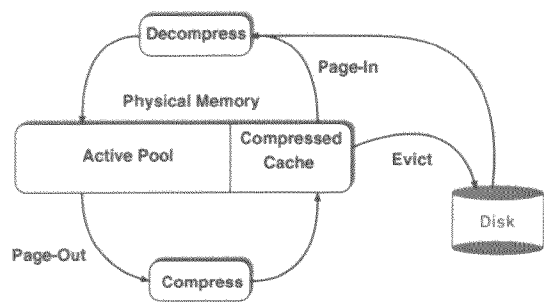


Figure 11.1: Paging to/from a Compressed Cache

reason for this is that the underlying disk uses direct memory addressing (DMA). Thus, for pages in the cache to overlap physical frames, these frames must be contiguous.

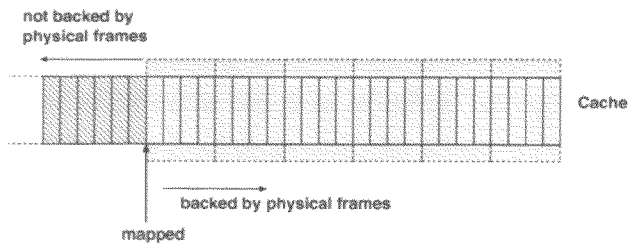


Figure 11.2: Cache Arrangement.

The cache grows down the way, meaning that when more physical frames are allocated for use by the cache, the mapped pointer is decremented by the appropriate number of frames. Similarly, when physical memory is revoked, the mapped pointer is incremented accordingly. The revocation of physical frames has obvious ramifications for the cache managers — the eviction must free up the bottom-most frames.

Note that all the cache managers split the cache into sub-pages. This is used to control fragmentation and to make managing the space easier. The design of each cache manager is discussed below.

## 11.1 Basic Cache Manager

The basic cache manager splits the cache into two parts: the eviction area and the fetch area (which may be of zero size). On a write request to the cache, the page is placed in the eviction

area. When the eviction area becomes full, all pages are written contiguously to backing store. On a read request, the cache is checked to determine if the requested page is resident. If the page is resident, it is copied out to the appropriate location. If the page is not resident, a request for that page is forwarded to the backing store manager along with the size of the fetch area. The page and other pages in its group are read into the fetch area and the faulting page is returned. If the fetch area is of zero size, the faulting page is read directly from the disk to the appropriate location in memory.

## **11.2 Compressed Cache Manager**

The compressed cache manager divides the cache into sub-pages of 512 bytes in size. In the original prototype, pages in the cache did not need to be stored contiguously. However, since the physical memory allocated to the cache can grow or shrink, having the pages contiguous in the cache allows physical frames to be freed more easily for return to the stretch driver.

### **11.2.1 Servicing a Cache Read**

On a read request to the cache, a list of entries is searched for a match. If no match is found, the request is forwarded to the backing store manager. When the page is returned from the backing store, it is decompressed directly into the location supplied by the stretch driver. On a cache hit, the appropriate page is decompressed to the location specified by the stretch driver, adding the space to the free list. Note that a page never resides in both the cache and the active pool and that all pages, even clean pages, are placed in the cache once evicted from the active pool.

### **11.2.2 Servicing a Cache Write**

On a cache write, the page is compressed to a buffer. The page could be compressed directly to the cache but, at this time, the size of the compressed page is unknown and, as it is possible for a page to get larger when compressed, this would mean keeping at least a page of contiguous size free at all times. Consequently, it was decided to use a buffer for compressions and then copy the page to an appropriate location. Although this is slower than doing the compression directly to

the cache we can maintain more items in the cache and, should space need to be freed, we know exactly how much is required.

Once the page is in compressed form, the free space list is searched for a suitable location. If there is not enough free space in the cache, enough space must be freed to accommodate the new entry (see section 11.2.3).

### 11.2.3 Eviction Policy

When a new entry has to be added to a full cache, enough space to accommodate the entry must be created. This is achieved by removing enough entries to create the required free space. There are currently three methods available for choosing a victim to expel: strict FIFO, clean FIFO and weighted clean FIFO. Strict FIFO evicts the oldest page in the cache, i.e., the one that was placed in the cache first. Clean FIFO selects the FIFO clean page and expels it from the cache. This prevents the need for a write to backing store. If there is no clean page present, the oldest page is expelled to backing store. Weighted clean FIFO also attempts to select a clean page to be expelled as this is a cheaper operation than expelling a dirty page. However, this must be balanced against the probability that the clean page is more useful than the oldest dirty page. Thus, in an attempt to avoid the possible cost of fetching clean pages from the disk that may otherwise have been retained in the cache, a distance marker is used. The distance marker is used to prevent moderately used clean pages from being expelled before very infrequently used dirty entries. It represents the distance, in pages, from the FIFO clean page to the absolute FIFO page. The algorithm for calculating the entry to be expelled is as follows (where DISTANCE MARKER is currently greater than the number of pages in the cache):

1. Find FIFO clean entry (E).
2. If distance from absolute FIFO entry to E < DISTANCE MARKER then expel E.
3. Otherwise expel absolute FIFO entry.

Setting the distance marker to be larger than the number of pages in the cache, causes the algorithm to revert to being a clean FIFO policy. While the distance marker should have some potential, more investigation is required into its effectiveness.

### 11.2.4 Managing the free space

A free space node in the cache data-structure consists of an address and a number of free sub-pages.

Before a page is written to the cache, free space must be found for it. The first thing the cache manager attempts to do is to find an exact match according to the number of sub-pages a compressed page will occupy. If this is unsuccessful, it attempts to resolve the request by breaking a larger block.

When a page is expelled from the cache it is forwarded to the backing store manager for storage. The node referring to the location and number of the sub-pages it occupies are added back into the free-space data-structure. Freed nodes are first added according to their address. When the correct position is found, the previous and the next node are checked to determine if the space referred to by those nodes creates a larger block of free space.

## 11.3 Extended Compressed Cache Manager

Figure 11.3 shows the layout of the cache to accommodate prefetching and contiguous writes. We describe the case where the cache manager employs both contiguous writes and prefetching as this gives the full picture. In the case where the cache manager does not employ prefetching, the fetch area is of zero size.

As can be seen, the cache is split into three areas: the general pool (Area 1); the fetch area (Area 2); and the eviction area (Area 3). An important difference to note is that, although Area 1 can, theoretically, be split into sub-pages of any reasonable size, providing it is a power of two, Areas 2 and 3 are split into sub-pages identical in size to the blocks of the underlying disk. The reason for this is that it is not beneficial to have two pages with data on the same disk block: it complicates reads (a read must start at the beginning of a block) and it does not lend itself to the migration of pages from one group to another (see section 9).

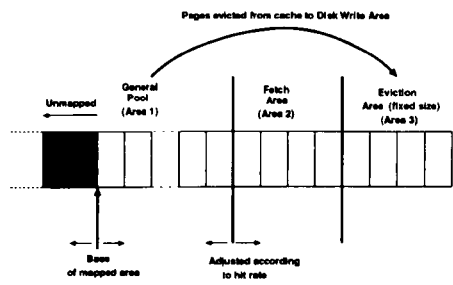


Figure 11.3: Cache Arrangement for Prefetching.

11.3.1 Servicing a Cache Read

On a cache read, each area is checked for the page in order Area 1 to Area 3. On a cache hit, the appropriate page is decompressed to the location specified by the stretch driver, and the space is added to the free list for the appropriate area. On a cache miss, a request for the page is forwarded to the backing store manager along with the location the page is to be read into and the size of the area. The backing store manager will then read in the faulting page and an appropriate number of other pages to fill that area. The faulting page is decompressed to the location specified by the stretch driver and the other pages are kept in the fetch area until required or until another cache miss.

Note that the cache manager can utilise contiguous writes without employing any prefetching. Under these circumstances, Area 2 is of zero size. Thus, a cache miss on a read will cause only one page to be fetched from disk and decompressed directly to memory.

11.3.2 Servicing a Cache Write

Writes to the cache are to Area 1 only and are exactly the same as those in the compressed cache manager (section 11.2.2). The only difference is that when pages are expelled, they are not written directly to disk, they are moved to the eviction area. When the eviction area is full, all pages are written contiguously to backing store.

### 11.3.3 Fetch Area Adaption

Although the eviction area is of a fixed size, the fetch area was designed to be dynamically adjustable to cope with changes in the performance of the prefetching algorithm. Thus, the fetch area could be the same size as the eviction area or it could be eliminated entirely.

## 11.4 Revocation of Physical Memory

The memory allocated to the cache can be adjusted at any time during the lifetime of an application. This allows the application to utilise more, or less, space for the compressed cache during different phases of execution. When the physical memory allocated to the cache is to be reduced, the stretch driver informs the cache how many frames must be returned. If all the frames are to be returned, the cache manager flushes all of its contents to backing store.

If, for example, the compressed cache is instructed to return a single frame to the stretch driver, the cache must free pages held in that frame. The cache first attempts to move those pages resident in the bottom frame to another location in the cache. If the cache is full, enough pages are evicted, in FIFO order, from the cache to make room for those pages resident in the revoked frame. The pages left in the revoked frame are then copied to other locations in the cache.

## 11.5 Compression Algorithm

The compressed cache managers take an interface reference to a compression algorithm as one of their arguments, allowing different algorithms to be used for different purposes. For instance, a compression algorithm like LZO which is good for data compression is not very good for the compression of code. Thus, a stretch containing code would require an algorithm suitable for code compression. The compressed cache can be instantiated with the appropriate algorithm without the need for re-compilation or re-linking.

The compressed cache initially used an LZO variant (miniLZO) but the overheads for compression (a 64KB buffer) proved prohibitive. The cache now uses the WKdm algorithm devised by Wilson et. al. [WKS99]. This algorithm performs faster than LZO with comparable compression ratios and only a 64byte overhead.



# Chapter 12

## Heracles Evaluation

*Never judge a book by its movie.*

**J. W. Eagan.**

In traditional operating systems, resources are managed by the kernel or privileged servers and untrusted applications are forced to adhere to the existing interfaces and implementations of system services. Such arrangements must meet all the needs of all the different types of application. This organisation is contrary to who holds the knowledge about a particular application. The notion that an operating system can anticipate the needs of every application for every environment is fundamentally flawed. Research suggests that attempting to provide such a system is infeasible and that the cost of mistakes is high [And92, CD94, BSP<sup>+</sup>95, EKO95, Ros95, SSS95, LMB<sup>+</sup>96, KEG<sup>+</sup>97].

Modern operating systems have attempted to provide a more flexible environment where the application can exert more control over its run-time environment [Ber94, CD94, SS94, Ros95, Lie96a, Eng98]. However, these operating systems can be accused of placing too much responsibility on the application developer. For most application developers, the cost of implementing a virtual memory management system for their application is simply too high. A consequence of this is that the developer is forced to use the system provided default. While assuming that most application developers will implement their own VMM systems seems somewhat far fetched, the assumption that they will understand their application seems fairly reasonable. Indeed, in order to implement their own VMM system, the application developer must know at least how their application behaves.

The Heracles VMM system provides a framework that allows experienced application developers to choose the particular VMM hierarchy that best meets the needs of their application. It also provides the novice developer with a framework for more adventurous experimentation. Heracles makes no assumption about the technology available to the application. It accepts that a particular VM hierarchy may be most suitable for one environment and an entirely different hierarchy may be required in another environment.

The key questions that we must answer are: can choice improve performance? And if so, can different choices yield better performance for different applications? Let us look at the three applications described in appendix G and consider their performance when utilising different hierarchy configurations. Note that only three applications were chosen because only three were required to answer the key questions. In order to show that application-specific memory management can benefit applications more than a general purpose solution, it is only necessary to prove that one application benefits most from a hierarchy that is not optimal for another application.

The particular applications presented here were chosen as a result of earlier experiments with trace-driven simulations of applications from the SPEC INT 95 suite, in particular, Perl, Go, Compress, and GCC. Traces from these applications were broken down and their memory reference behaviour was simulated on Nemesis. These experiments were exploratory in nature and were used to highlight how the separate components that go to make up a VM hierarchy interact. The three applications presented here all have linear memory access patterns, although they are linear in slightly different ways. For instance the data for compress consists of three large buffers, two of which are accessed sequentially in parallel during any given phase of execution. Each buffer is accessed starting from the bottom address and continues sequentially, a word at a time, until it reaches the top address. The matrix multiplication accesses the data in each column or row linearly but the move from row to row, or column to column, is not a linear access. The filter application operates on three separate buffers, accessing the data in each buffer linearly.

Applications with relatively similar memory access patterns were deliberately chosen because they were more likely to benefit from the same hierarchy.

# 12.1 Experimental Results

The results described in this section were taken on hosts with a Pentium PIII 450MHz processor, 64MB of RAM, an ATA IDE hard drive, and a 100Mb Ethernet connection to the local switched network.

Figure 12.1 shows the relative run-times for our three applications (filter, matrix multiplication, and compress) utilising only the local disk in a pure demand paging configuration. This configuration is found in most operating systems as the default.<sup>1</sup> Note that for the experiments described in this chapter, each application was run with extra CPU and bandwidth and was the only running process. The results obtained from limiting CPU and bandwidth showed comparable differences for the different hierarchies. The compress and the matrix multiplication applications were run with 16MB of physical memory and the filter application was run with 8MB of memory. The physical memory allocations reflected the size of each application’s data.

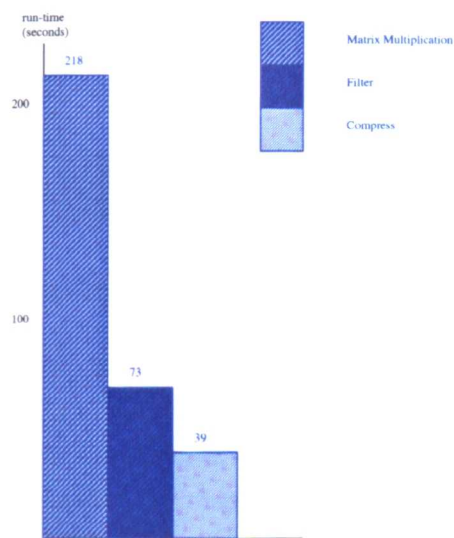


Figure 12.1: Demand Paging in Heracles.

Figure 12.2 shows the run-time performance of the same applications using the remote paging system described in chapter 10 on hosts connected via a 100Mb Ethernet switched network. As can be seen, adding remote paging increases the performance of all applications ( the filter

<sup>1</sup>Note that some operating systems will buffer pages for expelling and some will attempt to free up pages in idle time, however these strategies have not been applied on a per-application basis.

application is improved by around 70%). These figures would suggest that there is no need for utilising a local disk and we should instead use the remote paging system for all applications. Indeed, as shown in figure 12.3, providing fault tolerance via mirroring pages to another host still outperforms paging to the local disk. However, the remote paging system relies on there being enough hosts attached to the network with free memory that can be utilised by those that are heavily loaded. Furthermore, the speed of the network has an important impact on the potential benefits of remote paging.

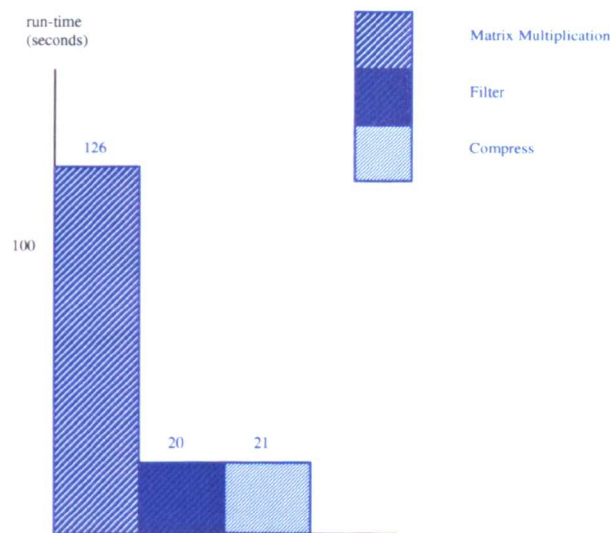


Figure 12.2: Remote Paging in Heracles.

While the performance of remote paging appears attractive, its dependency on free memory may mean that there will be times when it is not possible to acquire the necessary resources. Consequently, it is worth considering whether it is possible to increase run-time performance while relying only on local resources. In chapter 9 the friends algorithm was presented as a possible enhancement to demand paging. Figure 12.4 shows the results of utilising this algorithm for each of the applications.

While the performance of each application is significantly improved compared against local demand paging, only compress achieves an improvement over remote paging. This suggests that the best configuration for compress would not include remote paging. However, the other two applications still perform better with remote paging than they do using local paging with the friends algorithm. This does not mean that local paging with the friends algorithm should not be reserved as a choice for these applications. If the load on the distributed system is high and free

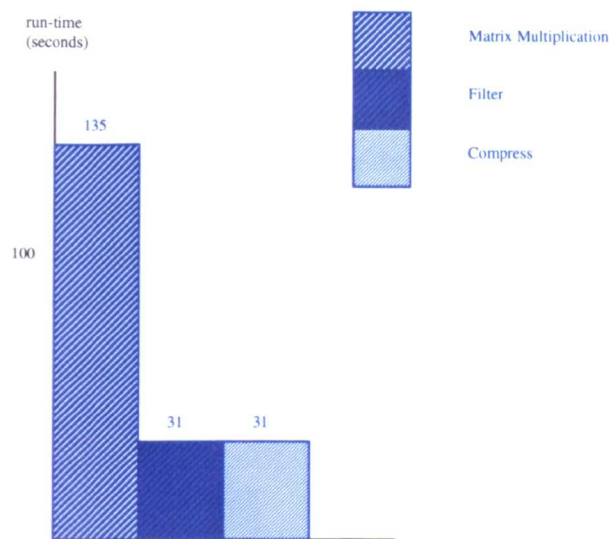


Figure 12.3: Remote Mirroring in Heracles.

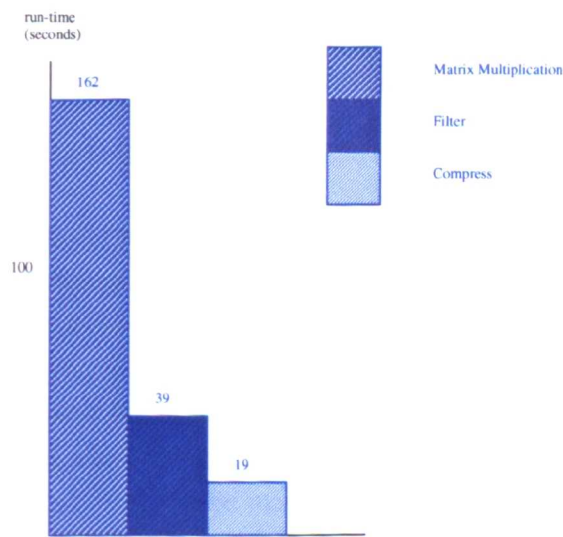


Figure 12.4: Local Paging using Friends.

memory is unavailable, both of these applications would benefit from using local paging with the friends algorithm compared to demand paging.

Now let us incorporate compressed caching into the list of possible configurations. Figure 12.5 shows the results of adding a compressed cache (cache sizes are 25 and 50% of physical memory) to the Compress application. As can be seen, in both the remote and the local case, the addition of a compressed cache hinders performance. This is because the Compress application accesses each buffer sequentially from beginning to end and, as such, never requires any of the pages held in the cache before they are expelled from memory.

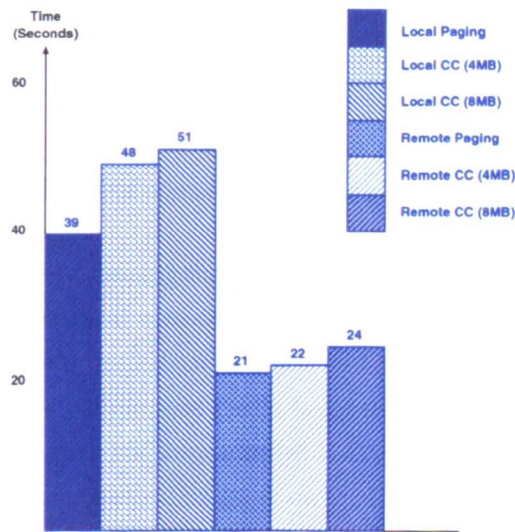


Figure 12.5: Effect of compressed cache on Compress.

Figure 12.6 shows the results of running the matrix multiplication application with various sizes of compressed cache (representing 12.5, 25 and 50% of physical memory) combined with paging to the local disk and across the network to remote hosts. All configurations show significant performance gains over paging to the local disk and paging across the network when used on their own. The maximum benefit is gained when the cache is 8MB in size, i.e. half of the physical memory allotted to the application. It is at this point where the applications ceases to expel any pages to backing store and all data can be maintained in memory. The matrix multiplication application was unique among the test applications in that it observed a steady benefit from a compressed cache regardless of the size. This is because the application had very good locality of reference and even when the compressed cache was only 2MB, it achieved a 79% hit rate.



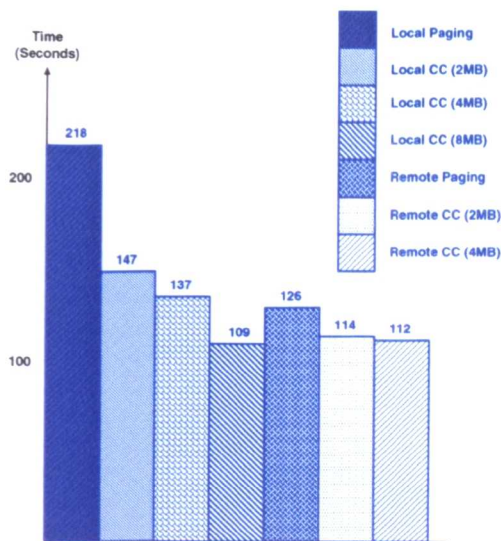


Figure 12.6: Effect of compressed cache on Matrix Multiplication.

Figure 12.7 shows the results of running the filter application with a compressed cache (the sizes represent 12.5 and 25% of physical memory). This application highlights an interesting difference between combining a compressed cache with local paging and combining one with remote paging. When the size of the compressed cache is set to 1MB, the run-time actually increases by around 28% for the local case but improves by 15% for the remote case. This is due to the fact that the hit rate for the compressed cache is only 14%. However, this does not explain the improvement observed when using remote paging. The run-time performance improves in the remote case because the pages being transferred across the network are compressed. Consequently, the time taken to evict and fetch a page is drastically reduced. Because the limiting factor for transferring a page across the network is bandwidth, the fact that the pages are now only requiring two packets to transfer instead of three means the application runs faster. This is not seen in the local case as the limiting factor for transferring a page to disk is latency.

Even when we increase the size of the compressed cache to 2MB, the performance gains for the local case are less dramatic than for the remote case. This is due to the fact that the filter application does not appear to exhibit very good locality. The cache hit rate for a 2MB cache is only 45%.<sup>2</sup> However, if we combine the compressed cache with the friends algorithm we can actually improve the cache hit rate. Figure 12.8 shows the same results as figure 12.7 with the

<sup>2</sup>The performance starts to further degrade as the cache size is again increased as the impact on the working pool with a relatively poor hit rate becomes more pronounced.

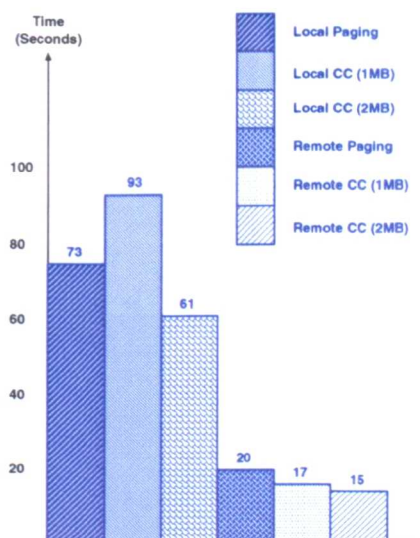


Figure 12.7: Effect of compressed cache on Filter.

addition of the friends algorithm to compressed caching.

As can be seen, the combination of the compressed cache and the friends algorithm greatly improves performance. This is because the cache hit rate is increased to 78% and 85% respectively for a 1MB and a 2MB cache. This is due to the fact that the filter application actually does exhibit some form of locality. It turns out that groups of pages close together tend to be used at around the same time. Even though the groups themselves always end up on disk before being required again, because they are fetched as a group, the cache hit rate increases significantly.

Note that for each application described here, each is affected in different ways by different hierarchies. This is because each references its data in memory in different patterns. The compress application performed best with a local disk and the friends algorithm. When compress was measured with a compressed cache, its performance degraded more significantly as the cache size was increased. This is because the memory access patterns of compress are linear. This means that when a page is expelled from the working pool into the compressed cache, it is compressed and then eventually written to disk. On a page fault, the page is fetched from disk and decompressed in memory. There will only ever be a cache hit when the cache is so large that all of the data can fit in memory at once. However, at this point the application's VMM is spending so much time compressing and decompressing pages to service page faults that the application has little time to carry out its task. The matrix multiplication application, on the other hand,



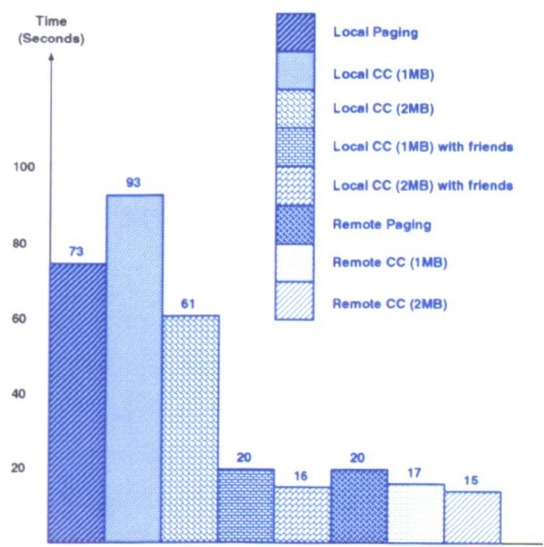


Figure 12.8: Effect of combining a compressed cache with the friends algorithm on Filter.

benefitted greatly from a compressed cache. This is because matrix multiplication has very good locality. Also, because it operates on a row by row and column by column basis, it actually makes better progress when all the data is in memory, even when more than half of that data is in compressed form. This is not the case for the filter application which benefitted most from a combination of a 2MB compressed cache and remote paging. This benefit was due to a combination of a moderate cache hit rate and the effect that the smaller pages had on page network transfer times. Interestingly, filter performed almost comparably well with the same sized cache and the friends clustering and prefetching algorithm.

It is important to realise that improving performance is not only about choosing the right hierarchy, it is also about the environment within which an application is executed, the application's failure model and about parameterising the different components that go into making up that hierarchy. For instance, although remote paging greatly improves performance compared to paging to the local disk, this is greatly dependent on the speed of the network. Performing remote paging over a 10Mb Ethernet will not outperform paging to a fast local disk. Similarly, network congestion, the availability of free memory on remote hosts, and resistance to failure all play a major part in the suitability of remote paging. For instance, the filter application performs best with a 2MB compressed cache in conjunction with remote paging. However, if the network is suitably slow or resources are unavailable the sensible approach would be to run filter with a 2MB compressed cache and local paging used in conjunction with the friends algorithm. Simi-

larly, if an application cannot tolerate failure, remote paging must be combined with some form of redundancy: this affects the potential benefits that can be gained from remote paging.

12.2 Experimental Variation

It is worth taking some time at this point to reinforce the difference between a conventional operating system and one supporting quality of service. In a conventional operating system applications are affected by other applications that happen to be running at the same time: other applications compete for resources such as physical memory, disk and network bandwidth and CPU bandwidth. In a QoS operating system such as Nemesis, each application is isolated from the effects caused by others running at the same time by acquiring guarantees for system resources. Applications running on Nemesis can specify the timeliness of their CPU scheduling, their disk bandwidth and their network bandwidth.<sup>3</sup> The consequence of this is that there is very little variability between runs of an application regardless of what other applications are also running. Table 12.1 shows the results of running the compress application with various virtual memory configurations. Compress was run with 12% disk/network bandwidth and 10% of the CPU. Each configuration was run between 10 and 25 times. The results show the mean, the shortest and the longest run in seconds.

Paging Type	Mean	Shortest	Longest	Longest - Shortest
Local Paging	2176.88	2161.64	2184.29	22.65
Remote Paging (10Mb)	1939.87	1938.76	1940.93	2.17
Remote Paging (100Mb)	1782.33	1776.48	1783.86	7.38
Local Paging with Comp. Cache	2495.80	2483.55	2521.32	37.77
Remote Paging (100Mb) with Comp. Cache	1813.74	1810.15	1817.46	7.31

Table 12.1: Mean value and range of results for different paging configurations.

As can be seen, the variability from the mean is extremely low. The worst case being just over 1%. Indeed it was not possible to present the data using error bars as the lines were completely flat. This was the case with all of the test applications.

<sup>3</sup>If the underlying network does not support this feature, then the guarantees received are for access to that network at the network interface level.

Most notable from the data is the difference in variation between remote paging and paging to the local disk. In section 7.2, where Quality of Service in Nemesis was described, the predictability of an application's behaviour was said to vary due to factors outwith the operating system's control. More specifically, the rotational latency of the disk and disk head positioning was highlighted as such a factor. This seems to be borne out by these experiments. The variation in the configurations utilising a local disk are significantly higher than those that do not.

## 12.3 The Cost of the Heracles VMM System

The Heracles VMM system is much more complex than a traditional VMM system due to the flexibility and choice that it offers. The code size is also bigger than a VMM system that simply pages to a local disk. However, only the code that is required by a particular application needs to be resident in memory. Also, because the code is provided as a shared library, only a single copy is required in memory. However, there are additional book-keeping costs associated with utilising a compressed cache, maintaining consistency of data on disk when using the friends algorithm, and paging to remote hosts. Unlike traditional operating systems, these costs must be met by the application. Additional data structures take up physical memory allocated to an application, reducing the amount of physical memory available for that application's data. Similarly, the compression and decompression of pages, when utilised, takes up CPU bandwidth that the application must pay for out of its guarantees.

The costs associated with Heracles are not hidden costs, but neither are the gains in performance false gains. If a compressed cache was added to a traditional operating system, the costs associated with it would be met by the operating system and consequently spread over all the applications executing at a particular time. This makes judging the possible benefits of new techniques potentially hazardous. This is not the case in the Nemesis operating system.

## 12.4 Using the Heracles VMM System

There are two ways of using the Heracles VMM system. The first is to configure the application's entry in the Nemesis namespace to specify the hierarchy preferences for the application. This will result in the application being initialised with the specified memory management hierarchy.

The second method, which can be used in conjunction with the first, involves accessing the Heracles components from within the application. This allows specific hierarchies to be created and “attached” to specific areas of memory. The Heracles interfaces are shown in appendices A through E.

### 12.4.1 Application Startup

```
b_env>mem=<| physBytes=16777216,  
cacheSize=4194304, prefetching=true, readSize=16384, writeSize=16384, paging="local_only",  
diskp=1000, disks=100, diskI=5, diskx=false, netp=30, nets=0, netI=1, netx=true, netc=false, svp=30,  
svs=0, svi=0, svx=true |>
```

Figure 12.9: Configuring the Nemesis Namespace to use Heracles.

Figure 12.9 shows the Heracles portion of an application’s entry in the Nemesis namespace. This information determines the components of the hierarchy that will be instantiated and what parameters are passed to them. The parameters determine the following:

- **physBytes**: the amount, in bytes, of physical memory the application requires.
- **cacheSize**: the amount, in bytes, of physical memory assigned to the cache.
- **prefetching**: whether to perform prefetching or not — only supported when performing paging to the local disk.
- **readSize**: the size, in bytes, of the prefetch area.
- **writeSize**: the size, in bytes, of the area used to perform contiguous writes.
- **paging**: the type of paging the application would like. This can be either “local\_only”, “remote\_only”, “local\_mirror” or “remote\_mirror”.
- **diskp**, **disks**, **diskx** and **diskI** represent the QoS parameters that should be given to the local disk. They represent period, slice, extra and latency respectively.
- **netp**, **nets**, **netx** and **netI** represent the QoS parameters for access to the network interface.
- **svp**, **svs**, **svx** and **svI** represent the QoS parameters for scheduling at the remote server.

- `netc`: determines if compression will be used in conjunction with remote paging.

When an application is started, its virtual memory hierarchy is determined from these parameters. The type of paging the application requests determines which parameters are examined and which are discarded. For instance, the disk QoS parameters have no meaning if the application is using remote paging. Similarly, “`netc`” is ignored if the VM hierarchy includes a compressed cache.

The parameters represented in figure 12.9 would result in a VM hierarchy that consists of: an extended compressed cache manager with 4MB of physical memory, 16KB of which will be assigned to an eviction area and another 16KB to a fetch area (see section 11.3); and a prefetching disk manager with 10% bandwidth to the local disk (100ms every second).

### 12.4.2 Accessing Components from Within an Application

Figure 12.10 shows code fragments from an application that performs the same function as the namespace configuration presented in the previous section. The code is simplified somewhat, in that the checking of return values is omitted.

As can be seen, harnessing Heracles is a very simple process even from within an application — there are only 15 lines of actual content in figure 12.10. Furthermore, the understanding required to use this technology is very limited. The application developer need only concern himself with understanding the nature of his application. The Heracles VM components provide a high-level notion of virtual memory management while hiding the complexity of the underlying system. An application developer may know, for instance, that the data, or a portion of the data, used by the application is fairly compressible and can choose to have a compressed cache in the application-specific hierarchy. This represents only a small insight on the developer’s behalf. Other components require even less insight into the behaviour of the application. The utilisation of remote paging in the management of the application’s virtual memory does not even require the knowledge that a network exists — Heracles will determine that such an option is unavailable and resort to using the local disk.

Operating systems that require application developers to implement their own strategy, by necessity, require the application developer to understand a great deal about virtual memory management. They must be aware of the page fault mechanisms implemented in a particular operating

```

/* Lookup the cache manager module */
cmod = NAME_FIND("modules>CacheManagerMod", CacheManagerMod_clp);
/* Lookup the compression module */
compmod = NAME_FIND("modules>CompressionMod", CompressionMod_clp);
/* Initialise the Compression Module */
comp = CompressionMod$New(compmod,
                           CompressionMod_ALGType_WKDM, /* The algorithm to use */
                           PAGE_SIZE, /* The size of the input - a system page */
                           heap /* A reference to a heap */);

/* Initialise the Cache */
cache = CacheManagerMod$NewCompPF(cmod,
                                   st->cachestr /* Stretch used by Cache */,
                                   st->buff /* Buffer for compression */,
                                   comp /* Our compression algorithm */,
                                   NULL /* No backing store manager yet */, heap,
                                   CacheManagerMod_Replacement_Weighted_Clean_FIFO,
                                   ptype /* type of paging */,
                                   readarea, writearea /* size of fetch and eviction area */);

/* Lookup namespace for Disk QoS parameters */
if (Context$Get(st->env, "mem", &any)){
    DEFVAL(qos.x, "mem>diskx", bool_t, st->env, &any);
    DEFSIZE(qos.p, "mem>diskp", st->env, &any);
    DEFSIZE(qos.s, "mem>disks", st->env, &any);
}

/* Lookup Disk Manager Module */
dmmod = NAME_FIND("modules>DiskManagerMod", DiskManagerMod_clp);
/* Get disk */
ldisk = DiskManagerMod$NewPF(dmmod, &qos, offer, 512 /* sub-page size */, heap);
/* Lookup the stretch driver module */
isdmod = NAME_FIND("modules>ISDriverMod", ISDriverMod_clp);
/* Lookup the backing store manager module */
bsmmod = NAME_FIND("modules>BSManagerMod", BSManagerMod_clp);
/* Get our Backing store manager and give it the disk manager */
bsm = BSManagerMod$New(bsmmod, ptype, ldisk, rppclnt /* NULL */);
/* Initialise our stretch driver */
isdriver = ISDriverMod$NewRes(isdmod, Pvs(vp),
                              heap, st->strtab,
                              Pvs(time), my_pmem,
                              iostr, cache,
                              cachestr, bsm,
                              cacheSize,
                              prep);
/* Bind the stretch to the stretch driver */
StretchDriver$Bind(isdriver, userStretch, PAGE_WIDTH);

```

Figure 12.10: Using Heracles Within an Application.

system and how these interact with their application. They must further understand the issues of clean and dirty pages and any synchronisation problems to do with the mapping and unmapping of pages in memory. To provide a compressed cache, they must understand the issues involved in cache performance and how these can affect the performance of an application. To provide a remote paging system they must understand about distributed systems and network protocols and failure models. They must also grasp the effect of issues such as CPU crosstalk at the paging server and providing an appropriate scheduling policy.

While the potential for allowing the talented team of developers to take advantage of an operating system's flexibility should not be ignored, we must accept the fact that such teams are by no means the norm. It is for the majority that Heracles offers the real benefit. Furthermore, Heracles is sufficiently flexible to not only allow for such instances, but to aid the development process by providing components ready to be plugged in to existing designs.

## 12.5 Future Work

The Heracles VM system is aimed primarily at developers who know how their application behaves. In order to extend the flexibility that Heracles offers to less well informed developers, tool support for program behaviour analysis would have to be provided. In particular, a tool that analysed the memory access behaviour and attempted to discern the best possible configuration for an application would greatly extend the applicability of Heracles. An initial implementation of such a tool was used to analyse the paging behaviour of applications using Heracles. However, this only recorded pages expelled and brought in from backing store, storing these in a file. It did not attempt to identify key regions of memory that would benefit from a particular configuration.

An interesting aspect of Nemesis is that programs run with a particular configuration and particular QoS guarantees, will perform almost exactly the same each time they execute. This makes off-line analysis much more useful than it would be in a UNIX system where an application's performance can be affected by other processes.

In addition to the provision of proper tools, there are aspects of Heracles itself that require further analysis to determine their suitability. In section 10.5, the remote paging revocation procedure selected the largest stretch for revocation at the end of round five. The effect that this has on a distributed system operating near the threshold requires further investigation. While the expecta-

tion is that choosing the largest stretch is more likely to reduce the chance of a system operating near threshold from thrashing, more investigation is required.

A further area of future research would be to look at the impact of a multiprocessor environment on the Heracles VMM. A multi-processor version of Nemesis was not available while Heracles was being developed. In theory the additional CPUs would simply mean that there is more processor bandwidth available and that more applications could run with guarantees at the same time. However, this could affect the demand for other resources such as disk and network bandwidth. Future work would concentrate on analysing the effect this had on a machine running close to capacity.



# Chapter 13

## Conclusions

*That life is worth living is the most necessary of assumptions, and, were it not assumed, the most impossible of conclusions.*

**George Santayana.**

This dissertation has highlighted the deficiencies of virtual memory management in operating systems. The notion of user-level virtual memory management has been welcomed but with reservations. These reservations centre around the likelihood of application developers taking up the challenge of implementing application-specific memory management. It has been concluded that the overhead of implementing one's own virtual memory manager is too high for the majority of developers. Consequently, providing the extra flexibility for user-level implementations goes widely unused.

The *Heracles* virtual memory management system has been introduced as a method for providing application-specific memory management without the need for application developers to provide the implementation. Heracles is extremely flexible and highly configurable. Application developers can choose their own VM hierarchy from a suite of options and can parameterise the individual components to best suit the needs of their application. Although this system was implemented in a single address space operating system supporting quality of service, the principles of its design are just as relevant to other operating systems supporting user-level memory management schemes.

The work presented in this dissertation represents a break from the traditional incremental, and competitive, approach to improving system performance. The notion that one VMM technique offers a better solution to others has been discarded in favour of a more inclusive approach to virtual memory management. Each approach is seen to have merit and may be the most suitable solution for particular circumstances.

I have examined, amongst other things, the issues relating to paging to the local disk, paging to the memory of a remote host, and “extending” the size of physical memory by storing a portion of virtual memory in compressed form. I have further looked at how these techniques interact with each other and offered a more complete understanding of their behaviour than when examined in isolation.

Chapter 4 discussed the issues related to paging across the network to the memory of remote hosts. This was extended in chapter 10 with a description of remote paging in the Heracles VM system. The remote paging scheme differs from other schemes proposed in that it is highly flexible and suitable for general distributed systems on local area networks. The scheme allows hosts to determine their role within a distributed system dynamically according to local commitments. The load monitoring algorithm described in 10.3.3 takes account of memory and CPU commitments, the latter of which having gone previously unrecognised as a factor in a remote paging system. The scheme also ensures reliable delivery of data and supports active participation within a resource revocation framework. The potential benefits of on-line compression in conjunction with remote paging was also examined and found to offer potential benefits to applications.

Chapter 5 introduced the issues pertaining to providing a compressed cache within a VMM hierarchy and concluded that the application of such a scheme was not universally beneficial but offered potentially large performance improvements for some types of application. The resultant compressed caching scheme described in chapter 11 focused on deployment within an application-specific environment. While other researchers have focused on the compressibility of data and the speed of compression, the effect of CPU and disk bandwidth has been largely ignored. This can be considered a hangover from the UNIX mind-set where such costs are absorbed by the operating system. Investigation of these issues showed that the potential benefits of a compressed cache could not be measured simply in terms of  $x$  compressions plus  $y$  disk accesses versus  $z$  disk accesses. It was found that the timeliness of page faults could have an effect on the potential benefits of a compressed cache. The compressed cache was also examined in conjunction with the remote paging system and it was found that the compressed cache enhanced

the performance of the remote paging system, due to smaller network transfers, in a way that it did not enhance paging to the local disk.

While proving the worth of novel approaches to remote paging and compressed caching, the major contribution highlighted by the design of the Heracles VM system is that individual solutions to VMM can be combined in diverse and powerful ways to provide a specialised environment for all applications. Application developers do not need a deep understanding of VMM issues to harness the power of an application-specific VM hierarchy.

Future work on Heracles could focus on an implementation on other operating systems. In particular, it would be interesting to port Heracles to L4 and an exokernel as these systems offer the same level of control, without the QoS, as the Nemesis operating system. Within the system itself, there is still scope for the on-line analysis of how a particular configuration is performing and ways in which this could be used to dynamically alter the application's hierarchy at run-time. While this is not necessary for short-lived applications (simple profiling would be much more sensible) it could be beneficial to large long-lived systems. Another way of extending Heracles would be to leave the decision about the components required for a particular application in the hands of the system. The user could provide a high-level notion of expected performance and the system would construct a suitable hierarchy.

To conclude, this dissertation started with the realisation that while user-level virtual memory management offers much more scope for an application-specific environment, OS developers have been largely ignorant of what an application developer wants from an operating system. This may be due to the fact that OS developers understand the issues and the techniques for virtual memory management and assume these to be universal. For the most part, the application developer does not understand how virtual memory is provided by the underlying system. This should not be seen as an excuse to deny the accessibility to the power and flexibility offered by modern operating systems that support user-level policies.

I have described the Heracles virtual memory management system. As well as affording applications real gains, Heracles is simple and easy to use. It places very little responsibility on the application developer: the system can be configured easily without the need for the developer to provide any code. However, Heracles is flexible enough to offer more power to the advanced developer. Components of the hierarchy can be created and combined from within applications to meet the requirements of specific memory areas. The result is a powerful, flexible and accessible virtual memory management hierarchy.

# Bibliography

- [And92] T. Anderson. The case for application-specific operating systems. In *Third Workshop on Workstation Operating Systems*, pages 92–94, 1992.
- [AS99] Anurag Acharya and Sanjeev Setia. Availability and Utility of Idle Memory in Workstation Clusters. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 35–46, New York, May 1–4 1999. ACM Press.
- [Bar96] Paul R. Barham. Devices in a Multi-Service Operating System. Technical Report 403, University of Cambridge, England, October 1996.
- [BBD<sup>+</sup>98] M Beck, H Böme, M Dziadzka, U Kunitz, R Magnus, and Verworner D. *Linux Kernel Internals*. Addison-Wesley, 2nd edition, 1998.
- [BCD72] A Bensoussan, C T Clingen, and R C Daley. The Multics virtual memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972.
- [BE98] M. Barcellos and P. Ezhilchelvan. An End-to-End Reliable Multicast Protocol Using Polling for Scalability, 1998.
- [Ber94] B. Bershad. SPIN - an extensible microkernel for application-specific operating systems services. Technical Report TR-94-03-03, University of Washington, Seattle, WA, 1994.
- [BFR00] Caroline Benveniste, Peter A. Franaszek, and John T. Robinson. Cache-Memory Interfaces in Compressed Memory Systems. Technical Report RC 21662, IBM Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, February 2000.

- [BJLM92] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 2–9, New York, NY, September 1992. ACM Press.
- [Bla95] Richard Black. Explicit Network Scheduling. Technical Report 361, University of Cambridge, England, University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, April 1995.
- [BSP<sup>+</sup>95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Ermin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 267–283. ACM Press, December 1995.
- [CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 120–133, New York, NY, USA, December 1993. ACM Press.
- [CD94] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, 1994.
- [CE97] Michael B. Cox and David Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization*, pages 235–244. IEEE, November 1997.
- [CFL93] Jeff Chase, Mike Feeley, and Hank Levy. Some issues for single address space systems. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 150–154, 1993.
- [CG90] Douglas Comer and James Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *Proceedings of the USENIX 1990 annual technical conference: June 11–15, 1990, Anaheim, California, USA*, pages 127–135. USENIX, June 1990.

- [CKV93] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):257–266, June 1993.
- [CLBHL92] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. How to use a 64-bit virtual address space. Technical Report TR-92-03-02, University of Washington, Department of Computer Science and Engineering, March 1992.
- [CLBHL93] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.
- [Dah96] Michael D. Dahlin. The Impact of Trends in Technology on File System Design. available from <http://www.cs.utexas.edu/users/dahlin/techTrends/>, January 1996.
- [DD68] Robert C. Daley and Jack B. Dennis. Virtual memory, processes and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.
- [Dou93] Fred Douglass. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *1993 Winter USENIX*, pages 519–529. USENIX, January 1993.
- [Edw99] W. Keith Edwards. *Core Jini*. Prentice Hall, 1999.
- [EGK95] Dawson R. Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM : Application-Level Virtual Memory. In *Proc. Wshp. on Hot Topics in Operating Systems (HotOS-V)*, April 1995.
- [EKO95] Dawson R. Engler, M. Frans Kaashoek, and James Jr. O’Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.
- [Elp93] Kevin Elphinstone. Address space management issues in the Mungi operating system. Technical Report SCS&E Report 9312, University of New South Wales, Australia, November 1993.
- [Eng98] Dawson R. Engler. *The exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology, October 1998.

- [FHW99] Peter A. Franaszek, Philip Heidelberger, and Michael Wazlowski. On Management of Free Space in Compressed Memory Systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 113–121, New York, May 1–4 1999. ACM Press.
- [FM98] Michail D. Flouris and Evangelos P. Markatos. The network ramdisk : Using remote memory on heterogeneous NOWs. Technical Report TR98-0226, Foundation for Research and Technology - Hellas. Institute of Computer Science, August 1998.
- [FMP+95] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 201–212, December 1995.
- [Fot61] John Fotheringham. Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store. *Communications of the ACM*, 4(10):435–436, 1961.
- [FR98] Peter A. Franaszek and John T. Robinson. Design and Analysis of Internal Organizations for Compressed Random Access Memories. Technical Report RC 21146, IBM Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, October 1998.
- [Han97] Steven Hand. Deliverable 2.3.1: Virtual address management. deliverable 2.3.2: Virtual memory management. Deliverable of project ESPRIT LTR 21917, July 1997.
- [Han98] Steven Hand. *Providing Quality of Service in Memory Management*. PhD thesis, University of Cambridge, 1998.
- [Han99] Steven Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of 3rd Symposium on Operating Systems' Design and Implementation*, New Orleans, February 1999.
- [HERV93] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochtelloo. Mungi: A distributed single address-space operating system. Technical Report SCS&E Report 9314, University of New South Wales, Australia, November 1993.

- [HEV<sup>+</sup>97] Gernot Heiser, Kevin Elphinstone, Jerry Vochtelloo, Stephen Russell, and Jochen Liedtke. Implementation and Performance of the Mungi Single-Address-Space Operating System. Technical Report UNSW-CSE-TR-9704, University of New South Wales, June 1997.
- [HHL<sup>+</sup>97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of  $\mu$ -Kernel-based Systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 66–77, New York, October 1997. ACM Press.
- [HV84] L. W. Hoevel and J. Voldman. Range-driven prefetching. *IBM Technical Disclosure Bulletin*, 26(10A):4874, March 1984.
- [HY96] Yiming Hu and Qing Yang. DCD-disk caching disk : A new approach for boosting I/O performance. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 169–178, New York, May 22–24 1996. ACM Press.
- [JFV<sup>+</sup>96] Hervé A. Jamrozik, Michael J. Feeley, Geoffrey M. Voelker, James Evans II, Anna R. Karlin, Henry M. Levy, and Mary K. Vernon. Reducing Network Latency Using Subpages in a Global Memory Environment. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 258–267, Cambridge, Massachusetts, October 1996. ACM Press.
- [Kap99] Scott F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, The University of Texas at Austin, 1999.
- [KAS98] Samir Koussih, Anurag Acharya, and Sanjeev Setia. Dodo: A User-Level System for Exploiting Idle Memory in Workstation Clusters. Technical Report TRCS98-35, University of California, Santa Barbara, 1998.
- [KCE92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single address space operating systems. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, 1992.
- [KEG<sup>+</sup>97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and



- Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 52–65, New York, October 5–8 1997. ACM Press.
- [KGJ96] Morten Kjelsø, Mark Gooch, and Simon Jones. Design and Performance of a Main Memory Hardware Data Compressor. In *Proc. of EUROMICRO Conference*. IEEE Computer Society Press, 1996.
- [KGJ98] M. Kjelsø, M. Gooch, and S. Jones. Empirical Study of Memory-Data: Characteristics and Compressibility. In *Computers and Digital Techniques*, volume 45, pages 63–67. IEEE, 1998.
- [KKT96] Sneha K Kasera, Jim Kurose, and Don Towsley. Scalable Reliable Multicast Using Multiple Multicast Groups. Technical Report 96-73, University of Massachusetts, October 1996.
- [KLVA93] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the Development of Application-Specific Virtual Memory Management. In *Proceedings of the OOPSLA '93 Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 48–64, October 1993.
- [LCC94] Chao-Hsien Lee, Meng Chang Chen, and Ruei-Chuan Chang. HiPEC: High performance external virtual memory caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 153–164, 1994.
- [Lie95] Jochen Liedtke. On  $\mu$ -Kernel Construction. In *In Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Review*, pages 237–250. ACM, December 1995.
- [Lie96a] Jochen Liedtke. L4 Reference Manual - 486 Pentium Pentium Pro. Technical Report RC 20549, T.J. Watson Research Centre, P.O. Box 218, Yorktown Heights, NY 10598, 1996.
- [Lie96b] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, Sept 1996.

- [LMB<sup>+</sup>96] Ian M Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Media Applications. *IEEE journal on Selected Areas in Communications*, 14(7):1280 – 1297, September 1996.
- [LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Inc., 1989.
- [MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc., 1996.
- [MD96] Evangelos P Markatos and George Dramitinos. Implementation of a Reliable Remote Memory Pager. In *Proceedings of the USENIX 1996 Annual Technical Conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, pages 177–189, Berkeley, CA, USA, January 1996. USENIX.
- [MSP98] Philip Machanick, Pierre Salverda, and Lance Pompe. Hardware-Software Trade-Offs in a Direct Rambus Implementation of the RAMpage Memory Hierarchy. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 32, pages 105–114, San Jose, California, October 1998. ACM.
- [Nel86] Michael Newell Nelson. Virtual Memory for the Sprite Operating System. Technical Report CSD-87-301, University of California, Berkeley, June 1986.
- [Org72] Elliot I. Organick. *The Multics System: An Examination of Its Structure*. The Massachusetts Institute of Technology, 1972.
- [PGG<sup>+</sup>95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.
- [PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pages 109–116, Chicago, IL, June 1988.

- [RC96] Mark Russinovich and Bryce Cogswell. RAM Compression Analysis. O'Reilly Online Publishing Report, February 1996.
- [RO92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [Ros94] Timothy Roscoe. Linkage in the Nemesis Single Address Space Operating System. *Operating Systems Review*, 28(4):48–55, October 1994.
- [Ros95] Timothy Roscoe. The Structure of a Multi-Service Operating System. Technical Report 376, University of Cambridge, August 1995.
- [RTY<sup>+</sup>88] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Bar, D. Black, W. Bolosky, and J. Chew. Machine-independent Virtual Memory Management for paged Uniprocessor and Multiprocessor Operating System. *IEEE Transactions on Computers*, C-37:896–908, 1988.
- [SD91] Bill Schilit and Dan Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report CUCS-004-91, Colombia University, February 1991.
- [Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.
- [SS94] C. Small and M. Seltzer. VINO: An integrated platform for operating systems and database research. Technical Report TR-30-94, Harvard University, Cambridge, MA, 1994.
- [SS97] M. Seltzer and C. Small. Self-monitoring and Self-adapting Operating Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, May 1997.
- [SSS95] M. Seltzer, C. Small, and K. Smith. The Case for Extensible Operating Systems. Technical Report TR-16-95, Harvard University, Cambridge, MA, 1995.
- [SUN98] SUN. *The UltraSPARC<sup>TM</sup>-III Processor*. Sun Microsystems Laboratories, Inc, M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, 1998.
- [TPG97] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume

- 25,1 of *Performance Evaluation Review*, pages 100–114, New York, June 15–18 1997. ACM Press.
- [VAK<sup>+</sup>98] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System. In *Proceedings of 1998 ACM Sigmetrics Conference on Performance Measurement, Modeling, and Evaluation*. ACM, 1998.
- [VK96] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, September 1996.
- [WKS99] Paul Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Technical Conference*. USENIX, June 1999.
- [WMSS93] T. Wilkinson, K. Murray, A. Saulsbury, and T. Stiernerling. Compiling for a 64-Bit Single Address Space Architecture. Technical Report TCU/SARC/1993/1, Systems Architecture Research Centre, City University, London, March 1993.
- [YTR<sup>+</sup>87] M Young, A Tevanian, R Rashid, D Golub, J Eppinger, J Chew, W Bolosky, D Black, and R Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, volume 21, pages 63–76. ACM, November 1987.

## **Part IV**

### **Appendices**

# Appendix A

## Read/Write Interface

RW : LOCAL INTERFACE =

BEGIN

```
Read : PROC[IN vpn : WORD,
            IN loc : ADDRESS]
    RETURNS[size : CARDINAL];
-- Reads page referred to by vpn into location
-- pointed to by loc. Returns size if Read was
-- successful, 0 otherwise.

Write : PROC[IN vpn : WORD,
            IN loc : ADDRESS,
            IN size : CARDINAL]
    RETURNS[succ : BOOLEAN];
-- Stores page referred to by vpn situated at location
-- loc. The size is required because different sub-units may
-- get a different sized chunk of memory.
-- Returns True for success, False for Failure.
```

END.

# Appendix B

## Disk Manager Interface

```
DiskManager : LOCAL INTERFACE =
EXTENDS RW;
NEEDS FSTypes;

BEGIN

    Kind : TYPE = {STANDARD, ASYNCH, PREFETCH};

    PageDesc : TYPE = RECORD [ vpn : WORD,
                                size : CARDINAL];
    PageDescs : TYPE = SEQUENCE OF PageDesc;

    Init : PROC[IN pages : WORD]
        RETURNS[succ : BOOLEAN];
    -- initialises apt swap space to hold "pages" of VM pages.

    AdjustQoS : PROC    [IN OUT qos : FSTypes.QoS ]
        RETURNS [ok : BOOLEAN];

    WriteContiguous : PROC[IN pages : PageDescs,
```

---

```
IN                                loc  : ADDRESS]

    RETURNS[];

ReadContiguous : PROC[IN vpn : WORD,
                      IN loc : ADDRESS,
                      IN num : CARDINAL]
    RETURNS[ pages : PageDescs];

Free: PROC[]
    RETURNS[];
-- Free up resources.

ResetStats: PROC[]
    RETURNS[];
-- Reset measurements

PrintStats : PROC[]
    RETURNS[];
-- Prints statistics about the Disk Manager

END.
```



# Appendix C

## Remote Paging Server Interface

```
RPPCtl1 : INTERFACE =
```

```
NEEDS NetAddr;
```

```
NEEDS RPP;
```

```
BEGIN
```

```
Error : TYPE = { None, Failure, NoResources, DoesNotExist, OverCommitted };  
-- Return type of method calls, as we do not really want to be  
-- raising exceptions over RPC.  
-- OverCommitted is used when a new stream is requested with given  
-- QoS parameters and, although there are resources available, the  
-- given  
-- QoS causes the server to become over committed. In this case,  
-- the QoS value is set to none.  
-- It is also used if a stream attempts to attain more than the  
-- maximum allowed stretches per stream.
```

```
STR_INFO : TYPE = RECORD [ mem  : CARDINAL,  
                             pout : CARDINAL,  
                             pin  : CARDINAL,
```

---

```
        role : RPP.Role,
        stat : RPP.Status,
        qos  : RPP.QoSParms,
        charge : RPP.ns];

-- mem value in KBytes

MEM_USAGE : TYPE = RECORD [ used : CARDINAL,
                           free : CARDINAL];

-- values in KBytes

CreateStream : PROC[IN client : NetAddr.SAP,
                   IN bytes : CARDINAL,
                   IN qos    : RPP.QoSParms,
                   IN role   : RPP.Role,
                   IN cbport : CARDINAL,
                   OUT cid   : RPP.ClientID]
    RETURNS[error : Error];
-- Takes the number of pages in the stretch and the QoS
-- parameters.

CreateShallowStream : PROC[IN client : NetAddr.SAP,
                          IN qos    : RPP.QoS,
                          IN sid    : RPP.StreamID,
                          OUT cid   : RPP.ClientID]
    RETURNS[error : Error];
-- Used by a server to set up a connection to another server
-- for the offloading of a client's pages

RealiseStream : PROC[IN cid : RPP.ClientID]
    RETURNS[error : Error];
-- Activates stream cid set up on behalf of a client by another server.

AdjustQoS : PROC[IN sid : RPP.StreamID,
                IN qos : RPP.QoSParms]
```

---

```
    RETURNS[error : Error];
-- Adjusts the QoS parameters for the given client.

DestroyStream : PROC[IN sid : RPP.StreamID]
    RETURNS[error : Error];
-- Free up resources held on behalf of client

AddResources : PROC[IN sid    : RPP.StreamID,
                    IN bytes : CARDINAL]
    RETURNS[error : Error];
-- Takes the starting page in the client's stretch, the
-- amount of memory and the QoS
-- parameters. Tries to attain enough memory to meet the
-- requirements.

ForwardResources : PROC[IN server : NetAddr.SAP,
                       IN sid    : RPP.StreamID,
                       IN new_sid : RPP.StreamID]
    RETURNS[];
-- Forwards resources held on behalf of stream sid
-- to new server where the sid there is new_sid.

-- These functions are for interaction from the graphical
-- swap daemon

RevokeStream : PROC[IN pos : RPP.StreamID]
    RETURNS[];
-- Revokes stream at streams[pos]

GetStreamInfo : PROC[IN sid : RPP.StreamID,
                    IN OUT strinf : STR_INFO]
    RETURNS[];
```

```
GetMemUsage : PROC[IN OUT  musage : MEM_USAGE]
    RETURNS[];
```

```
GetAdvert : PROC[]
    RETURNS[ adv : CARDINAL];
-- Memory registered with trader
```

```
END.
```

# Appendix D

## Remote Paging Client Interface

```
RPPCInt : LOCAL INTERFACE =  
EXTENDS RW;  
NEEDS RPP;  
NEEDS SDriverCallBack;  
BEGIN
```

```
Init: PROC[ IN start  : ADDRESS,  
            IN pages  : WORD,  
            IN mirror : BOOLEAN]  
  RETURNS[success : BOOLEAN];  
  -- Initialises the Remote Paging Protocol layer.  
  -- Returns True if RPP layer was able to acquire  
  -- 'pages' of physical frames with server/s.
```

```
Exchange : PROC[IN src_vpn : WORD, IN src : ADDRESS,  
                IN size  : CARDINAL,  
                IN dest_vpn : WORD, IN dest : ADDRESS]  
  RETURNS[size : CARDINAL];  
  -- Takes the page src_vpn at address src and stores it.
```

---

```
-- Fetches the page dest_vpn and reads it into address dest.

AdjustQoS : PROC[IN qos : RPP.QoSParms]
  RETURNS[succ : BOOLEAN];
-- Adjusts the QoS. Returns True iff successful.

Free : PROC[IN addr : ADDRESS]
  RETURNS[];
-- Free up the resources we hold for the relevant stretch.

FreeAll: PROC[]
  RETURNS[];
-- Free up all resources at server/s

RegisterCallBack : PROC[ brok : IREF SDriverCallBack]
  RETURNS[];
-- register a callback to propogate changes to stretch driver

SwitchMirror : PROC[ vpn : WORD]
  RETURNS[ succ : BOOLEAN];
-- Promote the mirror to server for the apt vpn

END.
```

# Appendix E

## Cache Manager Interface

```
CacheManager : LOCAL INTERFACE =  
NEEDS BSManger;  
NEEDS Mem;  
BEGIN
```

```
Init : PROC[IN num : CARDINAL]  
RETURNS[succ : BOOLEAN];
```

```
-- The cache manager maintains a stretch backed by  
-- some physical memory. The cache maybe compressed, in which  
-- case, loads involve decompressing a page to the  
-- location specified.
```

```
Read : PROC[IN vpn : WORD,  
IN loc : ADDRESS,  
OUT clean : BOOLEAN]  
RETURNS[ hit : BOOLEAN];  
-- Reads the VPN into address loc.  
-- Needs to tell the stretch driver if the entry was clean or
```

```

-- not. This is because, a dirty entry may be expelled by the
-- stretch driver to the cache and then read in again from the
-- cache. If the cleanliness of the page is not passed back
-- then the stretch driver would assume it was clean. Thus, if
-- the page is expelled again without being touched, it may find
-- itself not being written to disk.
-- Returns whether there was a cache hit or not.

```

```

Write : PROC[IN vpn    : WORD,
IN      loc    : ADDRESS,
IN      size   : CARDINAL,
IN      clean  : BOOLEAN]
    RETURNS[ succ : BOOLEAN];
-- Stores page referred to by vpn situated at location
-- loc. The size is required because different sub-units may
-- get a different sized chunk of memory.
-- Returns True for success, False for Failure.

```

```

Exchange : PROC[IN src_vpn : WORD, IN src : ADDRESS,
                IN size : CARDINAL,
                IN dest_vpn : WORD, IN dest : ADDRESS]
    RETURNS[sz : CARDINAL];
-- Takes the page src_vpn at address src and stores it.
-- Fetches the page dest_vpn and reads it into address dest.

```

```

RegisterBSM : PROC [IN bsm : IREF BSManager]
    RETURNS[];
-- pass reference to Backing Store Manager

```

```

Present : PROC[IN vpn : CARDINAL]
    RETURNS[present : BOOLEAN];
-- Returns True if vpn
-- present in the cache.

```



---

```
MemInc : PROC[IN mem : CARDINAL]
    RETURNS[];
-- Increases physical memory available to cache

MemDec : PROC[]
    RETURNS[ ok : BOOLEAN];
-- Reduces physical memory available to cache

Free : PROC[]
    RETURNS[];
-- Frees up space being used by CacheManager and returns
-- stretch to the system.

Flush : PROC[]
    RETURNS[];
-- Flushes the contents of the cache

PrintStats : PROC[]
    RETURNS[];
-- Prints statistics about the cache manager.

Debug : PROC[IN stop : BOOLEAN]
    RETURNS[];

END.
```

# Appendix F

## QoSEntry Interface

```
-- A "QoSEntry" is an extension of an "IOEntry". It incorporates a
-- scheduling policy for "IO" channels based loosely on the Nemesis
-- EDF Atropos scheduler. This allows a quality of service to be
-- defined for each IO channel in the form of a period and a slice of
-- 'time' to allocate during that period. When "Rendezvous" is
-- called an IO channel is selected according to the QoS declarations
-- for service by a single server thread. The server thread is then
-- expected to serve a single packet from the IO channel, call
-- "Charge", and return for another IO channel to service using
-- "Rendezvous".
```

```
-- The scheduling algorithm operates using three internal queues of IO
-- channels - "Waiting", for channels that have work pending but have
-- run out of allocation, "Idle", for IO channels that have no work
-- pending and "Runnable", for IO channels that have work pending and
-- have remaining time in their current period.
```

```
QoSEntry: LOCAL INTERFACE =
  EXTENDS IOEntry;
  NEEDS Time;
```

BEGIN

OverAllocation : EXCEPTION [];

-- OverAllocation will be raised if a call to "SetQoS" requests a  
-- level of service that would take the overall allocation over  
-- 100\%.

InvalidIO : EXCEPTION [];

-- An attempt was made to "Charge" or "SetQoS" on an IO that is  
-- not managed by this particular "QoSEntry".

-- As far as the "QoSEntry" is concerned, each binding it has  
-- responsibility for is in one of a number of states:

State : TYPE = {

Idle, -- open, but with nothing pending.  
Waiting, -- out of allocation.  
Runnable, -- ready to go.  
CloseRequested, -- to be closed down (any thread can do this work).  
Closing -- being closed down by a thread.  
};

-- By default all channels added to an entry will have extra time service  
-- only. This means that they get a fair proportion of the service  
-- time that is currently unused being either unallocated or not  
-- used by a channel with a guarantee.

-- "SetQoS" is invoked by the server to set the quality of service for  
-- "io". "p" is the period, "s" is the slice and "x" allows the IO  
-- channel share extra time in the system. The "l" parameter gives  
-- the amount of latitude to be granted in the case that a client  
-- has poor blocking behaviour; its default should be zero unless  
-- you have a good reason for increasing it.

---

```
SetQoS : PROC [ io: IREF IO,  p:Time.T, s:Time.T, x:BOOLEAN, l:Time.T ]  
    RETURNS []  
    RAISES OverAllocation, InvalidIO;
```

```
-- Charge "io" for "time" nanoseconds of work. This allows a server  
-- to charge an IO channel for the work that it has carried out on  
-- its behalf. This need not correspond to the wall-clock time  
-- actually spent in the server for example, but could be estimated  
-- time to transmit a packet or the seek time of a disk.
```

```
Charge : PROC [ io: IREF IO,  time: Time.T ]  
    RETURNS []  
    RAISES InvalidIO;
```

```
-- Dump a scheduler log. For debugging use only, requires  
-- a custom-compiled "QoSEntry" to work. Will probably vanish in  
-- the future.
```

```
dbg: PROC[] RETURNS [];
```

```
END.
```

# Appendix G

## Test Applications

### G.1 Matrix Multiplication

Given two matrices, A and B, the product C is derived from the formula:

$$C_{ik} = A_{i1}B_{1k} + A_{i2}B_{2k} + \cdots + A_{in}B_{nk} = \sum_{j=1}^n A_{ij}B_{jk}$$

MatMult creates two matrices (of dimensions  $x, y$  and  $y, x$ ), populates them with random numbers from 1 to 10 and then multiplies them together. Table G.1 shows the different sizes of matrices used and the associated memory consumption of each.

Dimensions	Memory Required (KB)		
	Total	Row	Column
500 × 600	1171	1.95	2.3
1000 × 2000	7812	3.9	7.8
2000 × 2000	15625	7.8	7.8

Table G.1: Matrix dimensions and their associated memory costs.

Each matrix is implemented as a two dimensional array. The  $y$  dimension is allocated first and then each row is allocated in turn from the top down. This means that rows are allocated contiguously. The effect of this is that, during multiplication, matrices A and C have good locality and matrix B has poor locality (figure G.1).

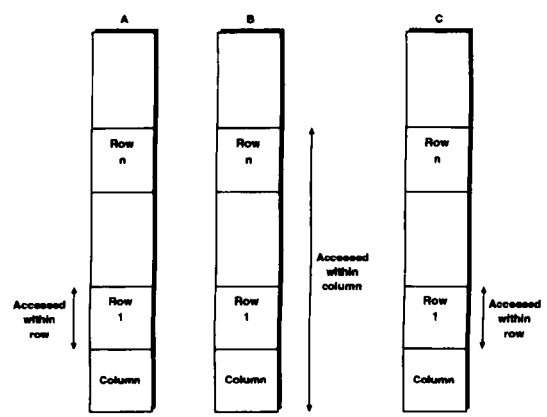


Figure G.1: Locality of Reference in Matrix Multiplication  $C = A * B$

As a consequence of the poor locality observed by matrix B, memory allocation of matrix B was optimised such that columns were allocated contiguously. This gives matrix B the same memory access pattern as matrices A and C. Thus:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{pmatrix}$$

would become:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

This slightly changes the previously mentioned formula to:

$$C_{ik} = A_{i1}B_{k1} + A_{i2}B_{k2} + \dots + A_{in}B_{kn} = \sum_{j=1}^n A_{ij}B_{kj}$$

## G.2 Image Filter

The filter program loads an image into memory (a 328KB bitmap file). It then splits the RGB values into three separate buffers (each of which is 4MB in size) and applies a sharpen filter to the data. The filter works on a pixel and those surrounding it, applying weightings to each. The result is essentially linear memory access patterns. This also results in linear page eviction.

### G.3 SPEC Int 95 Compress

The compress program allocates three buffers of 14160KB on the heap. It then populates the first buffer, compresses it to the second, decompresses it to the third and then compares the decompressed results with the original. This application is close to the best case scenario for paging to a local disk as page eviction patterns are linear.

The memory access patterns for compress are linear. From the population of the initial buffer to the compression/decompression and finally to the comparison, each buffer is accessed linearly. More interesting are the page eviction patterns. Even though, after the initial population, the program is accessing two separate buffers, it is only ever writing to one of them. This means that dirty pages are evicted linearly (figure G.2). This is extremely good behaviour for buffered writes but extremely bad for a compressed cache. The behaviour is also very good for data prefetching.

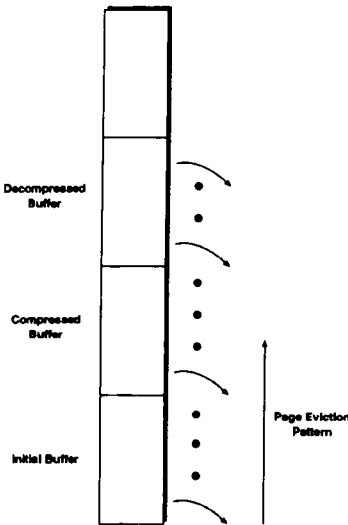


Figure G.2: Page Eviction Pattern in Data Compression Program

In a conventional UNIX system, compress would most likely be implemented using sequential files. While this is possible in Nemesis, measuring the performance of the application using sequential files would have introduced the file system performance as a factor in the results. For this reason, the memory was populated and faulted out to disk in advance of the measurements commencing. Thus, when the actual experiment starts, data is “faulted” in instead of read from a file and, as that particular data is never dirty, it is never written back out.