Global Memory Management for Workstation Networks

by

Michael Joseph Feeley

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree____

Date____

University of Washington

Abstract

Global Memory Management for Workstation Networks

by Michael Joseph Feeley

Chairperson of the Supervisory Committee: Professor Henry M. Levy
Department of Computer Science
and Engineering

Advances in network and processor technology have greatly changed the communication and computational power of local-area workstation networks. However, operating systems still treat workstation networks as a collection of loosely-connected processors, where each workstation acts as an autonomous and independent agent. This operating system structure makes it difficult to exploit the characteristics of current networks, such as low-latency communication, huge primary memories, and high-speed processors, in order to improve the performance of network applications.

This dissertation describes the design and implementation of global memory management in a workstation network. Our objective is to use a single, unified, but distributed memory management algorithm at the lowest level of the operating system. By managing memory globally at this level, all system- and higher-level software, including VM, file systems, transaction systems, and user applications, can benefit from available network memory. We have implemented this algorithm in the Digital Unix operating system running on an ATM-connected network of Alpha workstations. System measurements show that, on a suite of memory-intensive programs, our system improves performance by a factor of 1.5 to 3.5. We will also show that our algorithm has a performance advantage over others that have been proposed in the past.

Another issue for global memory management is the implication it has on page size; large pages, while well suited for disks because they amortize seek time, increase the latency of remote-memory reads. Unlike disk access, a small message can be transferred from remote memory significantly faster than a full 8-Kbyte page. Furthermore, the per-

message overhead on modern systems is low enough to make small transfers practical. We will describe several techniques that exploit these important performance features of remote memory to reduce page fault latency seen by application programs. One promising technique is to prefetch 1- or 2-Kbyte subpages, allowing execution to continue before an entire 8-Kbyte page has been transferred into the faulting node's memory. We will present measurements that show that subpage prefetching improves performance by 24% compared with fullpage fetch from remote memory.

Vita

Mike Feeley, born in New Jersey in 1960, has lived his life in Idaho Falls, Salt Lake City, Canoga Park, and Seattle. He received degrees in Computer Science from the University of Washington: a B.S. in 1990 and a M.S. in 1992. With the completion of this dissertation, he takes his Ph.D. to Vancover B.C. to try something new.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

The Computer Science Department at the University of Washington is an amazing place. I have learned much about Computer Science and about myself during the past six years. I owe the success of this experience to two people primarily: my adviser, Hank Levy, and fellow student, Jeff Chase. I wish I could have learned half of what Hank had to teach me about how to think and especially, how to write. Thanks to Jeff for his insight, wit, creativity, and friendship. Thanks also to Ed Lazowska, whose enthusiastic support always seemed to come at just the right moment. I would also like to thank Anna Karlin for showing the tremendous benefits of combining theory with practice, and for her encouragement, and Brian Bershad for asking the tough questions. Finally, thanks to all of the faculty, staff, and students of the department, and particularly to Frankye Jones, who makes the whole place work.

The GMS project is a collaboration with many outstanding people; thanks to Hank Levy, Anna Karlin, Mary Vernon, Chandu Thekkath, Geoff Voelker, Herve Jamrozik, Gene Morgan, Jim Evans, and Fredric Pighin. Many other people helped in various ways, particularly Dylan McNamee. David Becker deserves special thanks for keeping the machines running. Thanks also to Digital Systems Research Center for supporting this research in many ways, including allowing me to spend several months of 1995 there while debugging the system and running the experiments.

Finally, thanks to my friends and family, who have supported me in so many ways; I am blessed to have such wonderful friends. Thanks to Philippa, who shared much of this journey with me. And to my family, I would not have made it without you; your love and friendship gave me strength and understanding, and made it all fun. And of course, in the end, there are two people that deserve more thanks than I can possibly express. The love and support of my mother has made all things possible; and my life is guided by what I continue to learn from my father: to work hard, to pursue my dreams, and to enjoy a good pint of Guinness.

Coinnus bheadh an t-ubhaillín acht mar bheadh an t-abhaillín.

To the thirst that is yet to come.
— an Irish toast

Chapter 1

INTRODUCTION

We have to continually be jumping off cliffs and developing our wings on the way down.

— Kurt Vonnegut Jr.

1.1 Overview

Recent technology advances have significantly changed the communication and computational power of loosely-coupled distributed systems such as networks of workstations (NOWs). Until recently, the processor/disk and processor/network performance ratios of these systems had been steadily increasing as neither disks nor networks could keep up with rapid improvements in processor technology; over a period in which processor performance improved by 200 times, disks improved by only 5 times and networks remained virtually unchanged. The gap between processors and disks is likely to continue to widen, but network performance is undergoing a 100-fold improvement that is dramatically lowering the processor/network performance ratio. This change is having a revolutionary effect on the nature of distributed systems, the way they are managed, and the applications that run on them. Operating systems, however, still treat the system as a collection of independent processors, a structure that makes it difficult for the system to exploit the characteristics of modern networks, such as low-latency communication, large primary memories, and high-speed processors, to improve performance.

Memory management is one operating system function that is greatly impacted by this new technology. Modern local-area networks (e.g., ATM [4, 32], FDDI [2], and 100-Mb/s Ethernet [52]) supply more than ten times the throughput of earlier networks and cluster networks (e.g., Myrinet [11] and Memory Channel [37]) supply throughput in the 1-Gb/s range. These advances are significant because the network is now considerably faster than disk for transferring data to and from main memory, a performance advantage due primarily to the fact that remote-memory access avoids the expensive seek and rotational latencies associated with disk access. We show in this dissertation, for example, that an 8-Kbyte page can be transferred from the memory of a remote ATM-connected workstation ten times faster than the page can be read from local disk. Thus, the memory of remote processors can be used as a valuable shared resource for every processor in the system.

There are two ways that remote memory can eliminate disk accesses and thus improve performance. First, when programs running on different nodes share data, a page fault on one node can be satisfied by reading from the memory of another node, if that node has recently accessed the page (e.g., program executables are often shared in this way). Second, idle memory in the network can be managed as another level of the memory hierarchy, to store data discarded from active nodes. This use of remote memory is particularly important, because the amount of memory per workstation is steadily increasing and a portion of this memory is typically underutilized in most networks; for example, many workstations use only part of their memory and idle workstations used none of it. This underutilized remote memory, if managed properly, can significantly improve overall system performance.

A central question for remote memory management is what part of the system should be responsible for key management tasks, such as locating remote data and choosing nodes to store data discarded from active nodes. Our primary thesis is that these tasks are best handled by the operating system's core memory management subsystem, the part of the system that manages local memory today. The key motivation for integrating local and remote memory management is that remote memory is just a collection of the local memories of all nodes in the network. Each workstation memory is used both in the traditional way, to store data for local programs, and in a new way, to store data for programs running on remote nodes. Managing remote memory thus requires intimate involvement with local memory management on every workstation in the network.

Integrating remote- and local-memory management is challenging, because current systems manage each workstation's memory locally and independently from the rest of the network. The goal of local-memory management is to optimize the use of workstation memory for local performance. Remote memory, however, is a distributed resource that is shared among all programs running in the network. Modern workstation networks are thus similar to tightly-coupled multiprocessors, where resource allocation and management strategies are global rather than local.

This dissertation proposes *global memory management* as a new approach for managing workstation memory. We will argue that, at its core, memory management is now a *global* optimization problem. The network's memories are managed collectively, by a distributed algorithm that runs on every workstation in the network. The algorithm uses global information to optimize management decisions for overall system performance; for example, each page fault replaces the network's globally-best victim page. Remote memory is supported in the lowest levels of the operating system and its use is thus fully integrated with all caching activities managed by the system: virtual memory, local and distributed filesystem caches, etc.

In the next two sections we will discuss the high-level goals for global memory management and the key issues that must be address if it is to be effective.

1.2 Goals of Global Memory Management

There are three overall high-level goals that guide the design of our global-management algorithm and its implementation.

- Avoid harming local performance.
- Keep pages on the right nodes at the right times.
- Maintain the most valuable pages in network memory.

The first goal is simply to ensure that replacing local memory management with a global algorithm does not degrade local performance. There are two issues to consider: the allocation of memory to remote programs and the overhead added by the global algorithm. First, if local programs *need* all of a node's memory (i.e., there is no idle memory on the node) the node should not house any remote pages. Second, compared to the traditional local approach, a global algorithm will have higher protocol overheads for gathering information, maintaining page directories, satisfying remote page-in requests, etc. These overheads, however, should not significantly slow any node in the system, particularly a non-idle node.

The second and third goals relate to the overall page placement and replacement objectives of global management. The *placement* goal is to keep pages on the right nodes at the right times and the *replacement* goal is to maintain the most valuable pages in memory somewhere in the network.

1.3 Issues for Global Management

There are a number of issues that must be addressed before global management can replace traditional workstation memory management. This section reviews these issues in three categories. First, we discuss the tension between local and remote uses of workstation memory. Second, we discuss the importance of making global policy decisions as efficiently as possible. Third, we discuss reliability concerns. The discussion includes pointers to the sections of this dissertation in which each issue is addressed.

1.3.1 Tension Between Local and Global Performance

A global workstation memory hierarchy is comprised of a collection of memories each of which is a local memory to one node and a remote memory to all other nodes. Each of these memories can be divided into two regions, one that stores pages of *local* interest and the other that stores *remote* pages on behalf of other nodes. An active node might hold only local pages and an idle node only remote pages, while an active node that does not use all of its memory may have some of each. Note that this local/remote partitioning is imperfect, as sharable local pages are also remote pages (i.e., they can be accessed by any node).

Managing the workstation memory hierarchy requires consideration of two factors: one global, the other local. The global goal is to provide the best overall system performance by minimizing access time (e.g., average access time) for all programs. The local goal is to use the main memory of a node to the maximal benefit of that node. An important issue for global management is thus to resolve the tension between these two goals to find the right tradeoff between local and remote caching for each node's primary memory. This tradeoff may vary depending on the properties of workstation nodes. Some workstation owners, for example, may be willing to tolerate less performance impact due to global management than others. Section 4.2 describes the definition of an objective function for global memory management that incorporates these two factors.

There are two ways that local performance might be adversely impacted by global memory management. Section 4.3 describes how our algorithm prevents the first, by ensuring that active nodes are not burdened by global management. The second impact, however, is an unavoidable consequence of the decision to use idle memory to store data for active nodes. Doing so uses network memory more efficiently and thus improves overall performance. The *responsiveness* of a workstation's transition from idle to active status is degraded, however, because idle memory may store data for programs that, while not executed frequently, are nevertheless important to the workstation's owner. When such a program is restarted on a previously idle node — to read mail or browse the web, for example — the workstation will appear less responsive while pages are faulted back into local memory, replacing remote pages currently stored on the node.

Three techniques can be employed to improve responsiveness of these idle-to-active transitions. The first is a natural consequence of global management: shared pages such as program executables might be stored in remote memory and thus will be faulted back in rapidly. The second technique is to identify key pages and lock them in memory; this reduces the amount of memory available for remote storage, but locking a relatively small number of pages may provide substantial benefit. The third technique is to group the key pages into segments that are written contiguously to disk when the node becomes idle so that they can be faulted back in using a single large disk read.

1.3.2 Making Global Decisions Efficiently

It is crucial that global memory management be efficient; if overhead is too high, our integrated local/global approach will not be feasible. If may be difficult, however, to achieve this goal because global management replaces many local algorithms with more costly distributed algorithms. There are three key management tasks to consider: (1) making global policy decisions, (2) locating pages in remote memory, and (3) maintaining global page state.

Key to global memory management are the global policies for placement and replacement. These policies make decisions that are *globally* optimized. It is crucial that these decisions be made efficiently and that global communication be minimized. Chapter 5 addresses this issue, describing how our algorithm makes global decisions and discussing alternative approaches.

The system must be able to locate pages stored in remote memory. Section 6.1.4 describes our approach, which maintains a directory with an entry for every page currently stored in network memory. Directory entries for non-sharable pages are stored locally while entries for sharable pages are distributed among all the nodes in the network. This global structure is used so that nodes can easily locate sharable pages stored anywhere in the network. Section 6.5 describes alternative approaches.

The system must also maintain certain state information for the pages it stores. Page

Table 1.1: The key global memory management issues covered by this dissertation.

Issue	Covered In
Balancing the tension between local and global performance	Section 4.3
Defining the objective function for global management	Section 4.2
Making global policy decisions efficiently	Chapter 5
Locating pages in remote memory	Sections 6.1.4 and 6.5
Maintaining and using page state	Section 6.4
Fault tolerance	Section 4.5

replication information, for example, is needed by the replacement algorithm that runs on each node, because it treats replicated and unique pages differently. Section 6.4 describes how we maintain this information and discusses alternatives.

1.3.3 Reliability Concerns

Finally, there are issues related to the reliability of the distributed algorithms needed for global memory management. In any distributed system, the probability of failure increases with the number of nodes. A key concern, therefore, is to protect global memory from the effects of node or network failures. In particular, such a failure must not result in permanent data loss. In Section 4.5 we describe our technique, which protects programs from remote memory failures by ensuring that every page in global memory is replicated on disk; that is, modified pages are written to disk before being sent to remote memory. This section also addresses coherence concerns.

1.3.4 Summary

Table 1.1 summarizes the key global memory management issues and indicates where in this dissertation these issues are covered.

1.4 Contributions of this Dissertation

This dissertation makes the following contributions.

- It presents a formal description of the problem of global memory management.

- It presents a novel global-information-based global memory management algorithm.
- It provides a detailed comparison of our algorithm and alternative approaches.
- It describes a complete implementation of global memory management in a commercial operating system and reviews our experiences with the prototype.
- It analyzes alternatives for subpage transfer techniques for page faults satisfied from remote memory.

1.5 Organization of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides the necessary background and reviews related work. Chapter 3 formally describes the problem of global memory management. Chapters 4, 5, and 6 describe our global memory management system: Chapter 4 describes the basic algorithm; Chapter 5 describes how the algorithm uses global information to make replacement decisions; and Chapter 6 describes our prototype implementation. Chapter 7 describes the performance of our prototype. Chapter 8 describes how the use of remote memory changes assumptions about optimal page size. Finally, Chapter 9 summarizes and concludes.

Chapter 2

BACKGROUND ON WORKSTATION NETWORKS AND MULTIPROCESSOR SYSTEMS

Begin at the beginning . . . and go on till you come to the end: then stop.

— Lewis Caroll, Alice's Adventures in Wonderland, 1865

This chapter provides background information for this dissertation in two areas. First, we will discuss the detailed characteristics of a broad range of multiprocessor systems, from workstation networks to high-performance multiprocessors, with the goal of understanding how modern high-performance workstation networks compare to more traditional tightly-coupled multiprocessor systems. Second, we will review the current memory management approaches taken in workstation networks and in tightly-coupled multiprocessors, contrasting the approaches and using the similarities to motivate global memory management for workstation networks.

The first key question is how do workstation networks compare with other types of multiprocessor systems. All of these systems consist of a collection of *processors*, *memories*, *disks*, and a *network*. They differ in how processors are connected to memory and to each other. Multiprocessors use specialized high-performance interconnection networks and may also provide hardware support for direct addressing of all system memory (as well as for other tasks such as cache coherence). Workstation networks use commodity local-area networks to connect processors; direct memory addressing is provided only for a workstation's *local* (*main*) memory. Until recently, the communication differences between these two types of systems have been profound. Multiprocessors have provided a broad range of implementation techniques and performance characteristics, while workstation networks have occupied a single point in the design space: independent processors connected by a slow network. Recent advances in local-area network technology, however, are closing this communication gap, thus blurring the distinction between the systems. Workstation networks now occupy several interesting points in the multiprocessor design space.

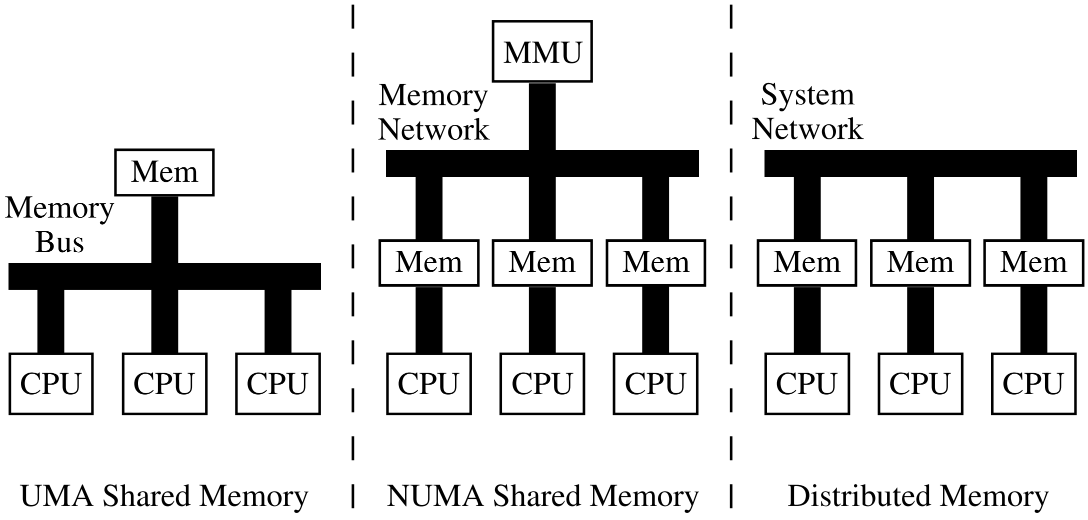


Figure 2.1: Multiprocessor configuration alternatives.

2.1 Memory Configuration of Multiprocessors and Workstation Networks

We begin with a description of the configuration of tightly-coupled multiprocessors and of workstation networks. This section reviews these systems, the communication performance they provide, and the memory hierarchy they implement.

2.1.1 Multiprocessors

Multiprocessors can be either *shared-memory* or *distributed-memory*, as shown in Figure 2.1. A shared-memory multiprocessor provides hardware support for direct addressing of all memory from any processor. Hardware shared memory simplifies parallel programs because every memory location is accessed in the same way, using basic machine load/store instructions. Shared memory is implemented in on of three ways: uniform-access (UMA), nonuniform-access (NUMA), or cache-only (COMA).

Uniform-access machines (UMA) consist of a collection of processors, each with a local cache, and a main memory connected by a shared memory bus. The shared bus provides uniform access time to all of memory from any processor but can become a bottleneck as the number of processors in the system increases, because all processor/memory traffic must share the single bus.

Non-uniform access machines (NUMA) allow a shared-memory architecture to scale beyond the limitations of a shared memory bus. A NUMA consists of multiple memories;

Table 2.1: Network performance of multiprocessor systems.

System	Type	Interconnect	Throughput (Mb/s)	Latency (μs)
Challenge XL	UMA	Bus	9600	0.3
DASH	NUMA	2-D Mesh	41	3 – 4.1
Alewife	NUMA	2-D Mesh	356	1.9 – 21
Cray T3D	NUMA	3-D Torus	2400	0.8
KSR-2	COMA	Ring Hierarchy	2560	9.5 – 23.8
CM-5E	DM-MP	(4-ary) Fat Tree	160	1.8 – 3.8
Paragon XP/S	DM-MP	2-D Mesh	720	42
SHRIMP	DM-MP	2-D Mesh	84	4.8 – 7.8

processors and memories are connected in a more complex way by multiple busses and a network. Each processor is directly connected to a *local* memory and by a slower network to all of the other memories in the system, the *remote* memories. *Cache-only* systems (COMA) are a type of NUMA that implement key memory management operations in hardware [43, 27]. COMA hardware is unique in two key ways. First, the unit of data movement is a *cache line* instead of a page. These cache lines are typically much smaller than a page (the KSR, for example, has 128-byte cache lines and 16-Kbyte pages). Second, the memory-management policy, which decides when to bring cache lines into main memory and when to replicate them, is implemented in hardware instead of software.

Distributed-memory multiprocessors (DM-MP) lack support for direct access to remote memory. Instead, each processor has a directly-connected *local memory* and processors are connected to each other by a general-purpose, but high-performance, network (i.e., a network with a general-purpose interface but highly-specialized implementation). Programs access remote memory by executing a software protocol on the local and remote processors, which exchanges network messages between the processors. These systems thus provide two different interfaces to memory. Local memory is accessed in the standard way using memory-access instructions, while remote memory is accessed using a general-purpose network send/receive interface. Accessing remote memory is thus more complex and less efficient: more complex because programs must distinguish between local and remote access; less efficient because accessing remote memory involves software overhead on both

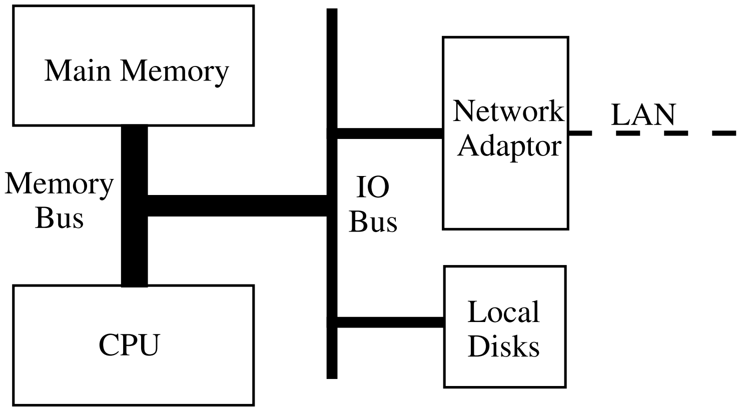


Figure 2.2: Configuration of a workstation-network node.

local and remote nodes to translate between memory accesses and network messages.

Table 2.1 shows the interconnection-network performance of several multiprocessor systems. For each system, the table shows the system type (NUMA, COMA, or DM-MP), the interconnect type, the peak node-to-node throughput the system can sustain, and the network-communication latency. The table lists the following systems: SGI Challenge XL [35, 65], Stanford DASH [49, 66], MIT Alewife [1], Cray T3D [5, 21], KSR [43, 71], CM-5 [76, 71], Intel Paragon [39, 71], and Princeton SHRIMP [30].

2.1.2 *Workstation Networks*

A network of workstations (NOW) is similar in many ways to a distributed-memory multiprocessor It consists of a collection of processing nodes, each with local memory, connected to each other by a general-purpose network, as shown in Figure 2.2. Programs access remote data in the same way, by exchanging network messages with a remote node. The similarities, however, end there; there are three main differences. First, workstation nodes are connected by a commodity local-area network (e.g., ATM, FDDI, or Ethernet), instead of a high-performance multiprocessor interconnect. Second, workstations run a more diverse but primarily sequential workload in which programs on different nodes are typically independent; this contrasts with multiprocessors, which are designed for a select class of computationally-intensive parallel programs. Third, workstation operating systems function independently on each node and manage all system resources locally; in contrast, multiprocessor systems typically manage some resources globally (e.g., processor scheduling) and other resources locally (e.g., memory management).

Until recently, workstation networks were built, almost exclusively, using Ethernet [52]. Traditionally Ethernet is a shared network with link bandwidth of 10-Mb/s. Modern networks have improved on this older technology in two ways: they provide at least a one order-of-magnitude increase in link throughput and they are usually switched instead of shared. A switched network has the advantage that full link bandwidth can be delivered to many nodes in the network simultaneously.

In addition to these advances in local-area networks, a new class of commodity workstation interconnect has emerged. *Cluster-area networks* (CAN), based on multiprocessor interconnect technology, provide inexpensive high-performance and reliable communication, but with restrictions on network topology; the network is normally limited to 16 or so nodes connected within 10 to 20 meters of each other. CAN networks provide latency and bandwidth comparable to some multiprocessor systems and allow the use of light-weight protocols that minimize host CPU overheads.

These changes in workstation interconnect technology are closing the communication gap between workstation networks and distributed-memory multiprocessors. Dramatic as these changes are, however, workstation networks are still different from traditional multiprocessors in at least one important way. This fundamental difference is that a network of workstations is a general-purpose computing system and as such must be able to run the widest possible class of workloads (traditional uniprocessor applications, databases, web browsers, etc). Tightly-coupled multiprocessors, in contrast, can typically be targeted to a narrower class of computationally-intensive parallel applications. Thus, from a hardware and performance perspective, workstation networks are becoming more like tightly-coupled distributed-memory multiprocessors; but, from a workload and system-management perspective, they may be more like a very large-scale distributed timesharing system.

Survey of Local Area Network Technology

Table 2.2 details the performance of several LAN technologies, which are described below.

100-Base-T Ethernet The Ethernet protocol has been extended from 10 Mb/s to 100 Mb/s throughput. Ethernet switches make point-to-point networks possible.

Table 2.2: Network performance of workstation networks.

	Network	Throughput (Mb/s)	Latency (μs)
LANs	100-Base-T	100	120
	FDDI	100	420
	AN2-ATM	155	64
	Fore-ATM	140	80
CANs	Myrinet	250 – 1250	8 – 38
	Memory Channel	800	5 – 20

FDDI Fiber Distributed Data Interface (FDDI) is a 100-Mb/s shared token-passing ring; dual counter-rotating rings provide reliability [2].

ATM Asynchronous-Transfer-Mode networks (ATM) have gained great popularity [4, 32]. Link bandwidth is typically between 100 and 155 Mb/s. Messages are divided into small fixed-size cells before injecting them into the network, thus simplifying switching, flow control, and bandwidth guarantee. Digital's AN2 network [4], for example uses *credit-based flow control* to provide reliable in-order message delivery in hardware, permitting a substantial reduction in software protocol overheads.

Survey of Cluster-Area Network Technology

Table 2.2 also details the performance of several CAN technologies, which are described below.

Myrinet Myrinet is limited to 16 nodes connected by either 8-m or 25-m cables. Theoretical peek bandwidth is 1.28 Gb/s for 8-m cables, 550 Mb/s for 25-m cables. Small-message latency is 38 μs for standard interfaces and as low as around 8 μs for customized interfaces [11].

Memory Channel Memory Channel (MC) [37] connects a maximum of 10 to 20 nodes, with 800 Mb/s bandwidth and a raw 5 μs hardware latency (20 μs total latency using a minimal-overhead software messaging protocol). MC implements a memory-mapped

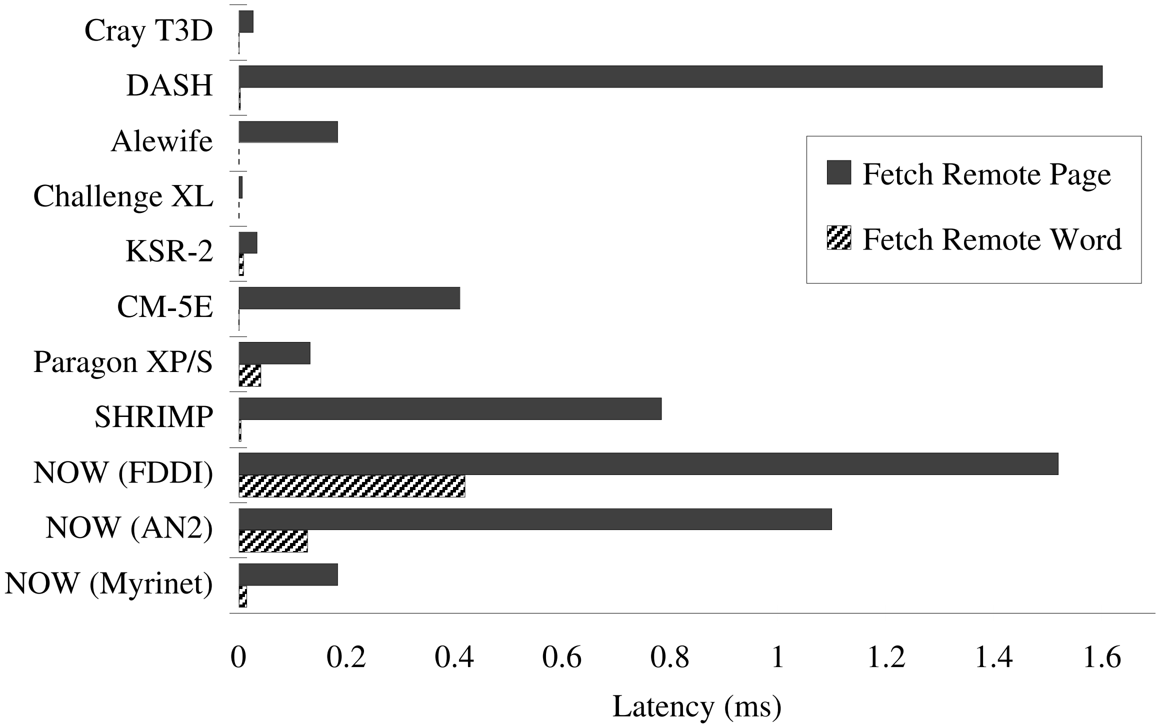


Figure 2.3: Remote-memory performance of multiprocessor systems and workstation networks.

interface (similar to the SHRIMP multiprocessor) that substantially reduces the host overhead required to format a message, transfer to the operating system, and perform protocol processing; like a shared-memory multiprocessor, communication is initiated by simply writing to a memory location. MC differs from multiprocessor technology in that it does not provide hardware cache coherence and in that it requires that programs explicitly bind memory locations to remote nodes. In addition, nodes cannot directly access the *local* memory of a remote node. To read a remote memory location, for example, requires interrupting the remote CPU to process the read request, adding as much as 150 μs overhead (on a 266-Mhz DEC Alpha running Digital Unix [29]).

2.1.3 The Memory Hierarchy

All of these systems, multiprocessors as well as workstation networks, implement a memory hierarchy. At the lowest level of each hierarchy is a processor cache or local memory and at the highest level is remote memory or disk. Figure 2.3 shows the remote-memory performance of each system described in this section. The two bars for each system indicate the round trip hardware latency for a single-word access to remote memory and a full 8-Kbyte page transfer from remote memory. The actual latencies for non-shared memory systems will also include software overheads, which are not shown in this figure. In our system, for example, these overheads add 315 μs to a remote-page fetch (see Section 7.1). This graph clearly indicates the closing communication gap between workstation networks and tightly-coupled multiprocessor systems. In every system, remote memory is significantly faster than disk, which has a random-access page transfer latency of around 14 ms.

A unique and important feature of this memory hierarchy is that *remote memory* is really just a collection of local memories. This is an important characteristic when considering memory management, because it indicates that local memory will be managed from two perspectives: as local memory for one processor and as a remote memory to every other processor.

2.2 Memory Management

Memory management is the task of moving data among system memories and between memory and disk at the right time in order to provide good performance and to maintain certain high-level properties of the data, such as coherence of cache-replicated data. In this dissertation, we are primarily concerned with the core data-movement aspect of memory management, which can be summarized by its three constituent subpolicies [74, 64].

Three Key Memory Management Subpolicies

Fetch The *fetch* subpolicy determines how and when the system fetches data from a distant memory to a closer memory. Demand paging, for example, is the common fetch policy for virtual memory; pages are transferred from disk to local memory on demand as they are accessed by programs. A common alternative is to *prefetch* data into memory in anticipation of its access. Prefetching is often speculative because future access patterns are not often know with precision; but, when effective, it can substantially improve performance by hiding fetch latency from a program, overlapping normal program execution with the fetch itself.

Placement The *placement* subpolicy selects the memory in which data is stored. Clearly, placement is only an issue in multiprocessor systems that have multiple memories.

In systems, such as workstations, that consist of only a single main memory and disk, the placement decision is trivial.

Replacement The *replacement* policy decides which item to discard from a target memory to make room for a fetched item. Replacement is necessary when the placement policy decides to locate an item in a memory that is already full, which is the common case.

The remainder of this section is organized into three parts. First, we will review the current approaches to memory management in workstation networks. Second, we will look at memory management in multiprocessor systems. Third, we review three areas in which the use of remote-workstation memory has been examined.

2.2.1 Workstation Memory Management

The main memory of every workstation is a data cache managed jointly by two distinct operating-system mechanisms. The *virtual memory system* manages program data such as executables, stacks, and heaps, storing active portions in memory and inactive portions on disk, if necessary. The *filesystem* manages a cache of recently accessed file blocks from both *local* and *remote* files (e.g., NFS [59] or AFS [61]).

Workstation memory management involves two main policy issues. First, both the VM system and the file-buffer cache implement a *replacement policy*. The polices for each subsystem are typically independent; a fault in one subsystem thus replaces a page (or block) managed by that subsystem. Second, the operating system implements an *allocation policy* to determine how much memory is allocated to each subsystem.

We will now discuss both of these key policy issues — replacement and allocation — in detail.

Replacement

The replacement policies of both VM and the file cache are similar. The goal is to replace pages that are least likely to be accessed again soon, using some heuristic to predict this future access behavior. The most common approach is LRU (Least Recently Used); it uses reference history to predict that pages accessed furthest in the past are those least likely to be accessed in the future. Most virtual memory systems actually use an approximation of LRU, as we describe in Section 6.3.4. Less common are frequency-based schemes such as LFU (Least Frequently Used), which are less popular because access frequency is more difficult to collect and interpret than access time [55, 82, 34].

There are three well-known problems with LRU (and LFU), which are all related to the fact that LRU considers only system-wide page access-time (-frequency) information and thus does not consider characteristics of the programs that are accessing those pages or the files to which they belong [70].

1. Some programs and/or files exhibit better locality than others and thus benefit more from having their pages cached. Replacing the oldest page in the system might be a poor choice if that page belongs to a high-locality program or file and there are other low-locality pages in memory. This is particularly important for file pages because LRU is a particularly bad policy for some files; sequential access to a large file, for example, could flush useful pages from memory, filling it with pages from the large file that are unlikely to be accessed again.

2. Some programs may be able to improve replacement decisions by providing future-reference hints to direct replacement. It is difficult to incorporate application-specific directions into a system-wide replacement policy.

3. LRU provides no degree of fairness in its selection of victim pages. A misbehaving program can induce excessive paging on other programs on a node by simply accessing a large number of pages in rapid succession. If accessed rapidly enough, all of these pages will be newer than pages belonging to other programs. Such a strategy can result in a program growing to consume all of the memory of a node.

Working Set LRU (WS) [24, 77] is an approach used by some virtual memory systems that provides a framework for solving these three problems. Instead of using system-wide-LRU replacement, WS assigns a separate memory region to each executing program and uses LRU *within* each region, ensuring that a faulting program replaces only its own pages. A system-wide *allocation* policy grows and shrinks regions in order to: (1) adjust to workload non-uniformity by allocating more memory to high-locality programs; (2) integrate application control by using application-specific policies instead of LRU for some regions; or (3) provide fairness by ensuring that every region receives a fair share of system memory. The main drawback of WS is that if regions are adjusted too slowly, programs many not be able to fully utilize available memory in a dynamic multi-programmed environment; and if adjusted too rapidly, many of WS's benefits over LRU are lost.

File-buffer caches often use an approach similar to WS to guard against misbehaving file accesses such as sequential access to a large file. Many systems, including Digital Unix, dynamically detect sequential accesses and limit the fraction of the file cache that a sequentially-accessed file can occupy. When this limit is reached, subsequent accesses replace the *file's* LRU page instead of the overall file-cache LRU page. Another approach is to allow applications to explicitly specify which files are sequentially accessed or which file pages are no longer needed (e.g., the **madvise()** Unix call provides these capabilities).

LRU-SP [12] is another approach, which uses LRU as the system-wide allocation policy and allows applications to make replacement decisions about their own pages.

Allocating Memory to VM and the File Cache

The second key policy question for workstation memory management is how to allocate memory between VM and the file-buffer cache, similar to the problem of allocating memory to program regions in the WS algorithm. In early systems, it was common to allocate a small fixed-size region of memory to the file cache and the remainder of memory to VM. This static allocation is clearly inefficient, however, because the memory needs of the programs running on a workstation change over time. When the demand for virtual memory is low, programs can often benefit from using excess memory to cache file pages; but, when VM demand is high, a small file cache and a large VM region usually provide the best performance tradeoff. The Sprite system explored dynamic allocation policies that grow and shrink the file cache with changing memory needs [54, 53]. This research concluded that VM pages are more valuable than file pages and thus they heavily favor allocating memory to the virtual memory system. The file cache only grows when there is excess memory not needed by VM and it shrinks rapidly when VM paging rates increase. Most modern system use similar dynamic allocation policies (e.g., the Digital Unix *Unified Buffer Cache*).

2.2.2 Multiprocessor Memory Management

The primary difference between multiprocessor and workstation memory management is that in a NUMA multiprocessor, the emphasis is on the *placement policy*, not on replacement. The performance of a program running in a NUMA depends critically on what memories store its data. Data should be stored as close as possible to the processors that access it in order to minimize the number of expensive remote accesses programs must make. When data is accessed by only a single processor, the best choice is clearly to try to place the data in the accessing processor's local memory. When data is shared among processors, however, *migration* and *replication* strategies are needed in order to achieve a good placement for all processors.

Migration is an effective strategy for data that is shared *sequentially* among processors. In Platinum [20], for example, remote pages are moved into local memory when they are accessed. Excessive ping-ponging of pages between nodes caused by concurrent sharing is avoided by freezing a page in place when its migration rate exceeds a threshold value; frozen pages are periodically defrosted in order to adapt to phase changes in program access patterns.

Replication is used to achieve local placement on multiple processors for data that is shared *concurrently*. Replication is particularly effective for read-only pages, because modifications to a replicated item require coherence operations to prevent processors from accessing stale data. A drawback of replication, even for read-only pages, is that storing multiple copies of a page reduces the number of unique pages that can be cached in memory. Replication may thus hurt overall performance by requiring a disk read for data that could otherwise be stored in memory.

A final issue of particular importance for workstation global memory management is the interaction between replacement and placement on NUMA systems. On a NUMA, replacement is typically handled locally; each node runs an independent replacement algorithm that handles local faults by replacing local pages. LaRowe and Ellis [41] proposed several alternative algorithms that improve on local replacement by allowing faults to use remote memory when programs need more memory than is locally available. Faulted pages are allocated on the local node as long as it has idle pages; that is, if doing so does not cause excessive paging on the node. When a node has no idle pages, faulted pages are allocated on remote nodes that do have idle pages. On the BBN Butterfly, on which their study was conducted, each node runs a version of the LRU Clock algorithm [6] to identify the idle pages on that node; a similar algorithm is used by most workstation systems such as Digital Unix. A key problem with the Clock algorithm, which we discuss in detail in Section 6.3.4, is that clock hands are advanced only by page faults; the pages of an idle node thus do not appear idle to other nodes until faults occur on the idle node. LaRowe and Ellis rectify this problem by advancing *every* node's clock when *any* node incurs a fault. This synchronized clock also maintains a global hint of the node that has the most idle pages, which is used by active nodes to pick the node to receive faulted pages.

2.2.3 Remote Memory in Workstation Networks

In this section we will discuss the three research areas that have examined uses of *remote memory* in workstation networks: remote paging for virtual memory, cooperative caching for filesystems, and distributed shared memory.

Remote-Memory Paging for Virtual Memory

Remote paging uses remote memory as backing store, instead of disk, in an otherwise standard virtual-memory paging system. Leach et al. were the first to describe remote paging in the context of the Apollo DOMAIN System [46]. Each machine in the network has a paging server that accepts paging requests from remote nodes. This system allowed local users to statically restrict the amount of physical memory available to the paging server.

Comer and Griffioen described a *remote memory model* in which the cluster contains workstations, disk servers, and *remote memory servers* [19]. The remote memory servers were dedicated machines whose large primary memories could be allocated by workstations with heavy paging activity. No client-to-client resource sharing occurred, except through the servers. Felten and Zahorjan generalized this idea to use memory on idle client machines as paging backing store [31]. When a machine becomes idle, its kernel activates an otherwise dormant memory server, which registers itself for remote use. Whenever a kernel replaces a VM page, it queries a central registry to locate active memory servers, picking one at random to receive the replacement victim. Felten and Zahorjan used a simple queueing model to predict performance. Recently, Markatos and Dramitinos built a remote paging system for Digital Unix [51].

In a different environment, Schilit and Duchamp have used remote paging to enhance the performance of mobile computers [62]. Their goal is to permit small memory-starved portable computers to page to the memories of larger servers nearby; pages could migrate from server to server as the portables migrate.

Cooperative Caching for Distributed Filesystems

A distributed filesystem (e.g., NFS [59] and AFS [61]) consists of a server node and multiple client nodes. Servers export files from their local disks and programs on client machines access these remote files as if they were local files. The client operating systems translate remote-file accesses into network messages exchanged with the server to request file blocks or to apply file updates.

Distributed filesystems cache file blocks in the memories of both server and client nodes. When a client accesses an uncached file block, for example, the block is transferred from the server disk into the memory of both the server and the requesting client. Subsequent access from that client can be satisfied from either client or server memory; access from another client can be satisfied from server memory. Distributed-filesystem blocks are typically stored in the standard file-buffer cache of client and server machines, whose management is entirely local.

Franklin [33] points out three key weaknesses of this local-management approach to client caching in the context of distributed object databases, which use the same style of client-server caching.

1. It should be possible to satisfy a cache miss from any client that currently caches a page, not just from the local-client and server caches.

2. Server-cache pages that are also cached by a client should be favored for replacement. Keeping such pages in server memory is wasteful when a cache miss can be satisfied from remote-client memory.

3. When a client replaces the last copy of a page, it may be beneficial to forward the page to the memory of another node (e.g., to server memory), instead of discarding the page from memory. By forwarding the page, the system permits a subsequent access to read the page from memory instead of from disk.

Franklin extended the Exodus [14] database to incorporate several features that address these concerns [33]. His ideas inspired a similar technique for distributed filesystems called *cooperative caching*.

Cooperative caching was introduced as part of the xFS serverless filesystem [22, 3]. In xFS, a file access can be satisfied from any client cache. Each client cache is managed using

LRU, extended to provide a degree of global cooperation. The goal of this cooperation is (1) to minimize duplication of pages on multiple nodes in the network and (2) to use idle memory to store pages recently replaced by active nodes. Dahlin et al. studied several cooperative-caching policies that provide varying degrees of cooperation. For example, *Greedy*, the policy implemented in the xFS prototype, satisfies cache misses from remote clients but clients still manage those caches locally. The most ambitious of these policies, and the one that their simulations showed works best, is *N-Chance Forwarding*. With N-Chance, each node manages its cache locally using LRU. When the last copy of a page is replaced, it is forwarded to a randomly selected node where it becomes the youngest page; a page can be forwarded again up to N times before being dropped. The hope is that a series of random forwarding choices results in the page arriving at an idle node, if there is one.

An alternative policy, recently proposed by Sarkar and Hartman, uses hints to direct forwarded pages to nodes most likely to have idle memory [60]. Hints are collected when a page is forwarded from one node to another; each node informs the other of the age of its oldest page by piggy-backing this information with the messages needed to forward the page.

We analyze both of these algorithms in detail, and compare them to our algorithm for global memory management, in Sections 5.4.1, 5.4.2, and 7.5.

Distributed Shared Memory

Distributed shared virtual memory (DSM), an active research area for several years [50, 15, 42, 9], is a software-implemented shared-memory abstraction for the distributed memories of a collection of workstations (or a distributed-memory multiprocessor). This abstraction permits a workstation network to be programmed as if it were a shared-memory multiprocessor but with certain performance and functionality limitations.

DSM systems provide an on-demand *fetch policy* and a local *placement policy*, but, importantly, they have not dealt with *replacement*. Instead, DSM-managed memory is allocated in VM and the VM system handles replacement in the normal way. Unfortunately, the VM system lacks important *global* information about DSM pages and thus will often make bad replacement decisions. If a replicated page is replaced, for example, the VM system writes the page to disk and later reads it back from disk when accessed again. These disk accesses are, of course, unnecessary, because a copy of the replicated page is in memory on another node. If the DSM system handled replacement in an integrated way, it could discard the replicated page and later read it from remote memory, avoiding the costly disk accesses. In fact, one way to think about global memory management, is that it extends DSM by adding replacement and moving key functions into the operating system.

2.3 Summary

Network advances are closing the communication gap between workstation networks and multiprocessors. Workstation networks now reside at several interesting points along the cost-scale-performance spectrum of multiprocessor systems, all of which implement a memory hierarchy for caching disk data.

Memory management is the task of moving data between the levels of the memory hierarchy and between memory and disk, optimizing the placement of data in memory for the best possible performance. Key to memory management are three subpolicies: the *fetch* subpolicy determines when data is transferred into memory, the *placement* subpolicy determines what memory receives the data, and the *replacement* subpolicy determines what is removed from memory to make room for the new data. For workstations, the key memory management issue has traditionally been replacement. Placement is important for systems that implement a multi-level memory hierarchy, such as NUMA multiprocessors. When a workstation is viewed discretely as a single node, placement is not an issue because the system contains only a single memory. Placement becomes important for workstation networks, however, because the network contains a hierarchy consisting of the memory of local and remote workstations.

In all types of systems, replacement is typically performed using a locality-based policy similar to LRU. Memory is divided into regions and LRU is applied within each region. Systems vary in how regions are used, how many there are, and whether they change size dynamically. Most systems divide memory into at least two regions: virtual memory and the file-buffer cache. It is common for the size of the file cache to change dynamically based on the memory demands of the system as a whole. The Working-Set replacement policy divides virtual memory of a single node into regions for each active program. In a non-uniform- or distributed-memory multiprocessor, each local memory naturally defines such a region.

Workstation memory management is traditionally local and the memory system non-hierarchical. Remote memory, however, has been incorporated in three different ways:

remote paging, cooperative filesystem caching, and distributed shared memory. Remote paging is a technique for using the memory of an idle machine as backing store for an otherwise local virtual memory system. Cooperative caching allows filesystem blocks to be retrieved from *any* client cache, while minimizing the duplication of blocks at multiple clients and using caches of idles nodes to store blocks replaced from active nodes. Distributed shared memory provides per-program distributed fetch and placement policies but no replacement policy.

The interaction between placement and replacement is important and has been studied in the context of NUMA multiprocessors. The overall goal of memory management is to minimize access time by eliminating unnecessary disk and remote-memory accesses. The placement subpolicy takes the local view of keeping data as close as possible to programs that access it, thus minimizing remote-memory access. The replacement subpolicy takes a more global view of keeping the most useful data in memory somewhere, thus minimizing disk access. LaRowe and Ellis demonstrated a policy that combined placement and replacement. Their system monitors the paging rate on every node and when it is low, page faults replace local pages. When paging is high, faults leave local memory alone and instead replace remote pages. This approach forges a compromise between the placement goal of keeping pages in local memory and the replacement goal of minimizing disk access.

Global memory management is unique in two important respects. First, the management of local and remote memory, placement and replacement, are tightly integrated into a single policy that uses global information for every replacement decision, ensuring that each decision is globally optimal. Second, this policy is implemented at the lowest level of the system allowing remote memory to benefit all higher-level functions such as VM, file caching, mapped files, etc.

Chapter 3

FORMAL DESCRIPTION OF GLOBAL MEMORY MANAGEMENT

Mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.

— Bertrand Russell, 1901

This chapter describes formally the problem of global memory management. Its presents a model of the memories in a distributed system and the flow of data among them. This flow of data is controlled by a *memory management system*, which is directed by a policy whose goal is to maintain in memory the data that is *most valuable* to programs running in the system.

We will begin by describing the simpler case of local memory management. We will then look at the key differences between local and global memory management and extend the model to incorporate these differences. Finally, will see that the addition of even one level to the memory hierarchy significantly complicates the problem of finding an optimal offline algorithm. Our model is designed to provide some insight into finding an optimal algorithm, but attempting to define one is beyond the scope of this dissertation. We will, however, develop a means of evaluating a set of memory management decisions that will allow us to compare alternative approaches. The goal of this chapter is to provide the background needed to understand how to build a good online algorithm.

3.1 Local Memory Management

3.1.1 Local-Memory Model

We begin by describing the simpler case of local memory management; that is, managing the main memory of a workstation or other system to cache disk data for programs running on that node. The system consists of a program execution and a two-level memory hierarchy of main memory and disks. A program is a sequence of S read or write references to disk-resident data items: $d_1, d_2, \ldots, d_S$. Some of the data items are cached in memory. The memory, M, is a subset of the disk items; $|M| = memorysize$, a constant. Let M_s be the memory state immediately before d_s is referenced.

There is a cost function, R_s, associated with each program data reference, and two constant costs C_l, the cost of a local memory access, and C_d, the cost of a disk access. The values of these constants are system parameters. The cost function is defined as follows:

$$R_s = \begin{cases} C_l & \text{if } d_s \in M_s \\ C_d & \text{otherwise} \end{cases}$$

A key question is to define the *objective function* optimized by the memory management policy. We assume, for the moment, a simple policy of minimizing the average access time for all programs; that is to minimize the quantity $\sum_{s=1}^{S} R_s$.

The central task of workstation memory management is to pick a *replacement victim* each time a program accesses a non-resident item, assuming a demand fetch policy. At program step s, if $d_s \notin M_s$ then $v_s \in M_s$ is replaced. Each such access changes system state in the following way.

$$M_{s+1} \leftarrow \begin{cases} M_s & \text{if } d_s \in M_s \\ M_s - v_s + d_s & \text{otherwise} \end{cases}$$

3.1.2 Optimal Offline Algorithm for Local Memory Management

Belady showed that the optimal replacement strategy is to replace the resident page that will not be not accessed for the longest time in the future [8]. This *offline* algorithm requires knowledge of the future reference stream and as such is useful only as a benchmark for comparing practical *online* algorithms, which typically use a heuristic such as LRU to predict future program behavior (as discussed in Section 2.2.1).

3.2 Differences Between Local and Global Management

Before extending the model to describe global memory management, let us look at the important differences between local and global memory management. There are four key differences related to the fact that memory management must now deal with page *placement* as well as replacement.

- There are now at least three levels of data-access cost (i.e., local, remote, and disk).

- There is a choice between remote access or caching.
- Remote pages can be replicated or moved.
- The local replacement victim can be forwarded to replace a remote victim.

3.2.1 Three-Level Data-Access Cost Function

A major difference between local and global management is that there are now at least three levels of data-access cost, which must be reflected in the data-access cost function. In the earlier model, access cost was uniformly C_l for memory and C_d for disk. Global memory management complicates the access-cost function, because local and remote memory have different access cost (i.e., C_l and C_r respectively). Some networks, in fact, may have multiple costs for remote-memory access (i.e., if some processors are more costly to access than others).

3.2.2 Remote Access or Caching

Data in remote memory can be accessed remotely instead of caching it in local memory, an option not feasible for data on disk. Remote access might be preferred if a program needs only a small portion of a remote page, because on most networks a small remote access will complete well before an entire page can be transferred into local memory. Alternatively, caching a page locally may increase remote-access latency, but it has the advantage that subsequent accesses to the page will complete locally. This tradeoff provides the motivation for our use of subpage transfers discussed in Chapter 8.

3.2.3 Remote Pages Can Be Replicated or Moved

If a page is shared between processors, it might be desirable to keep copies of the page in multiple memories (i.e., to *replicate* the page). Replication allows each processor to access the page locally. There are, however, two costs to replication. First, it reduces the number of unique pages that can be stored in network memory. Second, keeping multiple copies of a page complicates memory coherency and may result in extra messages and other additional overhead when a copy is modified.

3.2.4 Local Victim Can Be Forwarded to Replace Remote Victim

The final difference between local and global memory management is the question of what happens with the victim page replaced from local memory. In a local-management system, the victim is dropped from memory and re-read from disk if accessed again. With global management, however, it is possible to *forward* the victim to a remote memory, thus allowing a subsequent access to be satisfied from remote memory instead of disk. Notice, however, that forwarding may require another replacement, this time at the receiving node. The one case in which this remote replacement is not needed is if the faulted page is *moved* to local memory from the memory of a remote node. In this case, the system can swap the victim and faulted page, leaving both in network memory.

3.2.5 Summary

To summarize, the expansion of the memory hierarchy to include remote memory requires that the system make four independent decisions to handle a page fault.

1. Should the referenced page be cached locally or accessed remotely?

2. Should a cached paged be *replicated* or *moved* into local memory?

3. What local page should be replaced to make room for the cached page?

4. Should the local replacement victim be forwarded and thus replace a remote page?

Global memory management is complicated by these four factors and by the fact that choosing a replacement victim is itself less straightforward than for local management. In a two-level memory hierarchy, victim choice is simplified by the fact that the access cost for all non-resident pages is the same. This fact means that only *next-access time* need be considered to choose the best victim, choosing the page not accessed for the longest time in the future. In a three-level hierarchy such as a workstation network, however, local- and remote-memory access costs differ, requiring the consideration of *access cost*, in addition to *next-access time*, when choosing the victim page. As a result, Belady's algorithm in not optimal for global memory management. The next section presents a model of global memory management that captures these complexities. In Section 3.3.3, we will outline the factors involved in devising an *optimal* optimal replacement algorithm for global memory.

3.3 Model of Global Memory Management

The system consists of multiple memories, multiple processing nodes, and an interconnection network. As before, a memory is represented by the set of data items that are currently stored in the memory. Let $\mathcal{M} = M^1, M^2, \ldots, M^N$ be the set of memories in the system where M^n is the *local* memory of node n.

In the local model, we represented program execution by a sequence of memory accesses; for global management, every node has such a sequence. We thus model system execution by the combined execution stream $d_1^{n_1}, d_2^{n_2}, \ldots, d_S^{n_S}$. A reference, d_s^n, is the data element at global step s referenced from node n. References are partially ordered such that for any pair of references, d_s^n and d_t^m, $(n = m) \Rightarrow (s < t \Leftrightarrow d_s^n$ precedes d_t^m in program order$)$.

The state of a memory at program step s is given by M_s^n.

3.3.1 The Memory Management Decision

The global management decision can be formalized by the tuple $\mathcal{D}_s^n = (a, r, v, \hat{v})$, which represents the four subdecisions made for each fault: a indicates whether the page is accessed remotely or cached locally, r indicates whether a locally cached page found in remote memory is copied or moved, v is the local replacement victim if cached locally, and $\hat{v}$ is the remote (globally best) replacement victim. $\mathcal{D}$ is defined as follows.

$$\mathcal{D}_s^n = (a, r, v, \hat{v}) : \begin{cases} a & \in & \{local, remote\} \\ r & \in & \{\phi, move, copy\} \\ v & = & \begin{cases} \phi & \text{if } a = remote \\ d \in M_s^n & \text{otherwise} \end{cases} \\ \hat{v} & = & \begin{cases} \phi & \text{if } a = local \text{ or } v \text{ is dropped} \\ (e \in M_s^m, m) & \text{if } v \text{ is forwarded to node } m\text{, replacing } e \end{cases} \end{cases}$$

For each element, $d_s^n \in M_s^n$ in the global execution stream, the system computes $\mathcal{D}_s^n$ and then transforms the state of the system as shown below. There are two case; either the faulted page is found in remote memory or it is not.

If the faulted page is found in remote memory of another node m, one of the following three decisions is taken. First, if the faulted page is moved from node m to node n, it replaces v on node n and v is optionally forwarded to node m to occupy the memory vacated by the faulted page (i.e., the pages are swapped). Second, if the faulted page is instead copied, it remains on node m and v is optionally forwarded to another node o where it replaces page e ($\hat{v} = (e, o)$). Finally, if the faulted page is accessed remotely, no state change occurs.

$$\begin{aligned}
&\text{if } \mathcal{D} = (local, move, v, \hat{v}) \\
&\qquad M^n_{s+1} \leftarrow M^n_s - v + d^n_s \\
&\qquad M^m_{s+1} \leftarrow M^m_s - d^n_s + v \quad \text{if } \hat{v} = (d^n_s, m) \\
&\text{if } \mathcal{D} = (local, copy, v, \hat{v}) \\
&\qquad M^n_{s+1} \leftarrow M^n_s - v + d^n_s \\
&\qquad M^m_{s+1} \leftarrow M^m_s \\
&\qquad M^o_{s+1} \leftarrow M^o_s - e + v \qquad \text{if } \hat{v} = (e, o) \\
&\text{if } \mathcal{D} = (remote, \phi, \phi, \phi) \\
&\qquad M^n_{s+1} \leftarrow M^n_s \\
&\qquad M^m_{s+1} \leftarrow M^m_s
\end{aligned}$$

Alternatively, if the faulted page is not in memory anywhere in the network, one of the following two decisions is taken. First, if the faulted page is cached locally, it replaces page v which is forwarded to node o replacing page e ($\hat{v} = (e, o)$). Second, if the faulted page is cached and accessed remotely, it is copied to remote node o where it replaces page e.

$$\begin{aligned}
&\text{if } \mathcal{D} = (local, \phi, v, \hat{v}) \\
&\qquad M^n_{s+1} \leftarrow M^n_s - v + d^n_s \\
&\qquad M^o_{s+1} \leftarrow M^o_s - e + v \qquad \text{if } \hat{v} = (e, o) \\
&\text{if } \mathcal{D} = (remote, \phi, \phi, \hat{v}) \\
&\qquad M^n_{s+1} \leftarrow M^n_s \\
&\qquad M^o_{s+1} \leftarrow M^o_s - e + d^n_s \quad \text{if } \hat{v} = (e, o)
\end{aligned}$$

3.3.2 Evaluating Management-Decision Alternatives

We compute execution cost as before by summing the cost for each access in the execution stream. R^n_s is the cost of accessing d^n_s in whatever memory it resides; we will use R without a superscript to refer to accesses independent of the node that makes them. If we consider a simple three-level hierarchy, R^n_s is computed as follows.

$$R^n_s = \begin{cases} C_l & \text{if } d^n_s \in M^n_s \\ C_r & \text{if } d^n_s \in M^m_s \wedge m \neq n \\ C_d & \text{otherwise} \end{cases}$$

If the goal is to maximize global throughput, then the objective function is to minimize the following expression.

$$\sum_{n=1}^{N} \sum_{s=1}^{S} R_s^n$$

Which is equivalent to the following.

$$\sum_{s=1}^{S} R_s$$

Maximizing global throughput, however, might favor the performance of one node over others; that is, it might be *unfair*. One objective that provides a degree of fairness would be to minimize the following function.

$$\max_{n=1}^{N} \sum_{s=1}^{S} R_s^n$$

Other goals are possible but for simplicity, we will focus on the first goal of maximizing global throughput.

3.3.3 Toward an Optimal Algorithm for Global Replacement

The optimal memory management policy for global memory is much more elusive than for local memory. It is complicated by several factors: (1) there are four independent decisions to make related to be placement and replacement, (2) even local replacement decisions are more complex because there are three possible costs for each access (i.e., local, remote, and disk), and (3) the execution streams of all nodes must be considered because the changes in the memory of one node can change the performance of others.

One approach to optimality has been taken by Leff et al. [48, 47]. They assume a static access probability for every data page and thereby reduce the problem to that of finding an optimal initial placement of these data pages in workstation memories. This assignment of data to memories does not change over time because the access probabilities are static; as a result, replacement is not needed. Formulated in this way, memory management can be regarded as a *File Allocation Problem* [25, 81], which Leff et al. show can be transformed to an equivalent transportation problem [36] and thus efficiently solved [10]. While useful from a theoretical point of view, the assumption of static access probabilities is highly unsatisfactory for understanding memory management in real systems. The fact that access patterns are not static makes replacement necessary and makes the problem of finding an optimal solution more complex.

Defining an optimal algorithm for the more useful case, which includes both placement and replacement, is beyond the scope of this dissertation. Instead let us take a brief look at one of the four management subdecisions — choosing a local replacement victim — to see why Belady is not optimal and to get some intuition about how an optimal offline or, more to the point, a good online algorithm should behave.

Choosing the Optimal Local Replacement Victim

For local management, Belady showed that it is optimal to replace the resident page that will not be accessed for the longest time in the future. The intuition behind this choice is that by always replacing such a page, the system is guaranteed to perform the minimum number of faults; and, because each fault reads a page from disk, the number of disk reads is therefore minimized.

A memory hierarchy that include remote memory, however, complicates the replacement choice in two ways. First, faults do not always come from disk, they may also come from remote memory. Total fault cost is thus not minimized by minimizing the total *number* of faults, because multiple low-cost faults (from remote memory) might be preferred to a single high-cost fault (from disk). Second, pages can be accessed either locally or by a fault on a remote node. Replacing a locally-accessed page creates a future fault when that page is accessed again; while, replacing a remotely-accessed page may increase the cost of an existing fault, because a remote node's fault will be forced to go to disk instead of remote memory (unless the page is replicated on another node). Both costs must be considered to find the best replacement victim.

The cost of a replacement decision can be measured by the increase in access cost due to the replacement. Each replacement may cause some local faults and may increase the cost of some remote faults. The first such fault occurs when the replaced page is next accessed, either locally or remotely. A local access increases in cost from C_l to C_r, if the page is found in remote memory, and to C_d, if not. A remote access increases from C_r to C_d, unless a copy of the page exists on another node. A local access also *causes* a second replacement, which in turn impacts another future fault and possibly causes even more replacements.

Let f_i^v be the global step of the ith fault affected by the replacement of v and let F^v be the total number of affected faults. Also define R'_s to be the access cost at step s assuming that v is replaced and R_s to be the cost if no replacement occurs at the decision point in question.

The total change in access cost due to the replacement of v is called the *replacement cost* of v and is given by X_v, below.

$$X_v = (R'_{f_1^v} - R_{f_1^v}) + (R'_{f_2^v} - R_{f_2^v}) + \cdots + (R'_{f_{F^v}^v} - R_{f_{F^v}^v})$$

The optimal replacement victim is the one that results in the smallest access-cost increase; that is, the minimum value of X_v. In a two-level memory system, where $R'_f = C_d$ and $R_f = C_l$ for all values of f, we have $X = F^v \times (C_d - C_l)$; that is, the choice depends only on the *number* of replacements. In a three-level memory, however, the value of R'_f can be either C_d or C_r and the value of R_f can be either C_r or C_l. The minimum cost increase is thus not necessarily the one that results in the fewest number of faults, F^v. Instead, picking the best replacement victim requires considering both the number of future faults and access-cost increase of each of these faults.

3.4 Summary

This chapter presents a model that illustrates the issues, costs, and tradeoffs related to managing a three-level (or more) memory hierarchy. In a two-level memory hierarchy, memory management involves deciding what page to replace when a program accesses a non-local page. In a three-level hierarchy, there are four decisions to be made: (1) should a page be cached or referenced remotely, (2) if cached, what page should be replaced, (3) should a referenced page be moved into local memory or replicated, and (4) should the replacement victim be dropped or forward to replace a page on another node. Determining the best decision involves a complex tradeoff between local and remote reference cost and the next reference time of each page in the system.

In the next chapter, we will describe our algorithm for global memory management.

Chapter 4

THE GLOBAL MEMORY MANAGEMENT ALGORITHM

From each according to his abilities, to each according to his needs.

— Karl Marx, 1875

With this chapter we begin our description of the *Global Memory System* (GMS) we have developed [29, 40]. This chapter describes our global memory management algorithm, which handles page faults using a combined placement-replacement policy that globally manages workstation-network memory. Chapter 5 describes how we maintain the global information used to make policy decisions and includes a comparison with other approaches. Chapter 6 describes our prototype implementation of GMS.

The GMS management algorithm responds to each page fault by moving the faulted page into local memory and, if necessary, replacing the globally least valuable page in the network; Section 4.1 describes the algorithm. Key to the algorithm is the choice of an objective function that defines what page is *least valuable*; issues surrounding this choice are discussed in Section 4.2. The chapter concludes with a detailed discussion of the three key aspects of our algorithm: (1) the division of memory into local and global regions, (2) the selection of the least valuable page, and (3) the handling of coherency and node failures.

4.1 The GMS Replacement Algorithm

As previously stated, our goal is to globally coordinate memory management. We assume that nodes trust each other but may crash at any time. All nodes run the same algorithm and attempt to make choices that are good in a global network sense, as well as for the local node.

We classify pages on a node P as being either *local* pages, which have been recently accessed on P, or *global* pages, which are stored in P's memory on behalf of other nodes. Pages may also be either *replicated* or *unique*. A replicated page is found in the local memory of multiple nodes and is not found in global memory. A unique page may be either local or global and is found in exactly one memory in the network. Note that every page

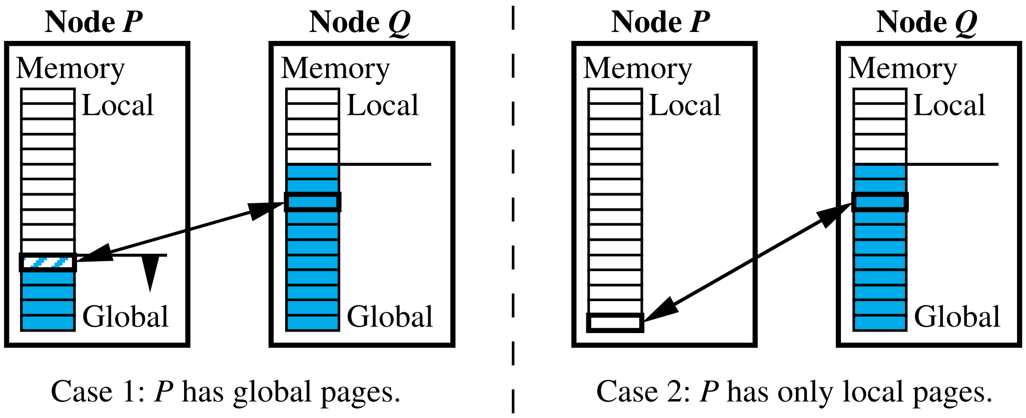


Figure 4.1: Global replacement with hit in the global memory.

in global memory is unique and that if a page is found in any local memory it will not be found in global memory.

The algorithm is driven by page faults (i.e., accesses to non-resident pages). Each fault triggers the global replacement algorithm on the faulting node. In response, the algorithm replaces a page on the faulting node, forwards it to another node, and transfers the faulted page into local memory; if the replaced page has been modified, it is written to disk in parallel with being forwarded to global memory. These actions may change the local/global memory balance on both the faulting node and on the node that receives the forwarded page. Page faults always result in a faulting-node replacement and sometimes result in a replacement on a remote node.

On a fault, Node P performs the following steps, which we describe in terms of four possible cases:

Case 1: The faulted page is in the global memory of another node, Q. We swap the desired page in Q's global memory with any global page in P's global memory. Once brought into P's memory, the faulted page becomes a local page, increasing the size of P's local memory by one page. Q's local/global memory balance is unchanged. This is depicted in Figure 4.1.

Case 2: The faulted page is in the global memory of node Q, but P's memory contains only local pages. Exchange the *least valuable local page* on P (writing it to disk if modified) with the faulted page on Q. The size of the global memory on Q and the local memory on P are unchanged.

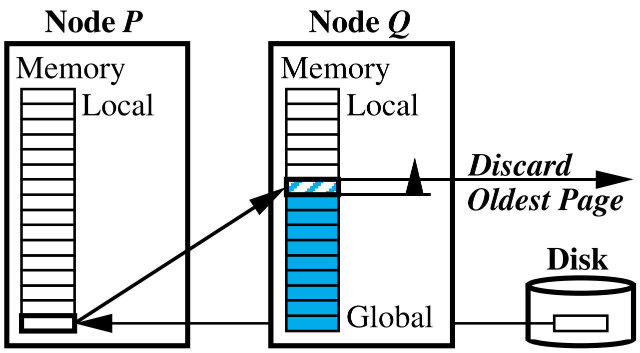


Figure 4.2: Global replacement with miss in the global memory (Case 3). The faulted page is read from disk and the oldest page in the network (local or global) is discarded.

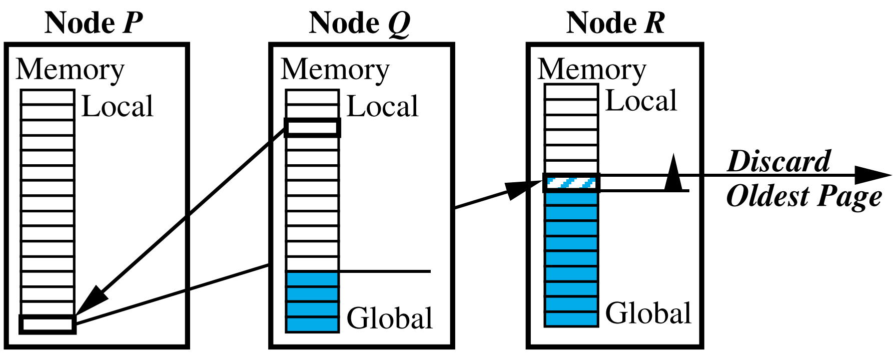


Figure 4.3: Global replacement with hit in remote local memory (Case 4). The faulted page is replicated and the oldest page in the network (local or global) is discarded.

Case 3: The page is only on disk. Read the faulted page into node P's memory, where it becomes a local page. Choose the *least valuable page in the network* (say, on node Q) for replacement and write it to disk if it is a modified local page. Send a global page on node P to node Q where it continues as a global page. If P has no global pages, choose P's least valuable local page instead. This is shown in Figure 4.2.

Case 4: The faulted page is in the local memory of another node Q. Copy that page into a frame on node P, leaving the original in local memory on Q. Choose the least valuable page in the network (say, on node R) for replacement and write it to disk if necessary. Send a global page on node P to node R where it becomes a global page (if P has no global pages, choose P's least valuable local page). This is shown in Figure 4.3.

The behavior of the replacement algorithm is fairly straightforward. To make room for the faulted page, a page must be replaced on the faulting node. This replacement victim is forwarded to another node where it becomes a global page. If the faulted page is found in global memory, the victim and faulted page are swapped and the local/global balance on the two nodes remains unchanged unless the faulting node replaces a global page, in which case the faulting node's local memory grows by one page (shown in Figure 4.1).

A page must be discarded from network memory if the faulted page is read either from disk (Figure 4.2) or from the local memory of another node (Figure 4.3). A discard is needed when reading from local memory because, in this case, the faulted page is *replicated* in order to avoid harming local performance on the remote node, and thus a swap is not possible. The algorithm discards the *globally least valuable* page in the network and the faulting node's replacement victim is forwarded to that node.

In every case, if the faulting node's replacement victim is either *replicated* at another node or is itself the *globally least valuable* page in the network, it is not forwarded to another node and is instead dropped from the network. Modified local pages are written to disk when they are forwarded to global memory on another node.

4.2 Objective Function

The algorithm description just given assumes the existence of four cost functions for comparing pages to choose the least valuable page on a node and in the network. Cost functions are needed for the following four decisions made by the algorithm.

1. Choose a node's least valuable local page.

2. Choose a node's least valuable global page.

3. Compare the least valuable local and global pages to choose the overall least valuable page in a memory.

4. Choose the least valuable page in the network from among the set of least valuable pages contributed by each node.

Before we can discuss how to make these decisions, we must first say something about what we mean by the *least valuable* page. This choice is guided by the replacement algorithm's *objective function*. The choice of an objective function involves several high-level policy tradeoffs which can be summarized by two, sometimes competing, concerns: *throughput* and *fairness*. Listed below are five possible objective functions that could be used alone or in combination.

- Maximize global throughput; or, in other words, minimize average read time for all programs in the network.

- Provide each program with a "fair share" of network memory resources.

- Maximize local throughput; or, in other words, minimize the average read time for the worst-performing program in the network.

- Maximize average speedup of all programs running in the network.

- Provide a lower bound on the slowdown of local programs caused by the use of memory for remote programs.

The choice of objective function is a matter of taste, which may vary from one user to another. For this reason, our algorithm is designed to accommodate a wide variety of possible objectives. The algorithm does not depend on any particular policy choice; instead, it simply assumes the existence of the four cost functions listed above.

4.3 Division of Main Memory into Local and Global Regions

The main difference between our algorithm and the formalisms introduced in the previous chapter is its division of memory into local and global regions. The *local* region stores pages in use by programs running on the local node and the *global* region stores pages on behalf of programs running on remote nodes.

The boundary between local and global memory on a node changes dynamically due to actions taken by the global replacement algorithm. In general, the local memory of a node grows when a fault occurs and the node has global pages; in this case, local memory grows by one page and global memory shrinks by one page (the replacement victim). Global memory on a node grows when a fault on another node results in a disk read or a replication and the first node has the least valuable page in the network in its local memory; the total global memory in the network grows when that fault occurs on a node with no global memory. The balance between local and global storage on a node thus depends on its dynamic workload and on the workload in the rest of the network. Over time, nodes that are actively computing and using memory will fill their memories with local pages and will begin using remote memory in the cluster; nodes that have been idle for some time and whose pages are old will begin to fill their memories with global pages.

This grouping of pages into local and global regions simplifies algorithm design by dividing memory management into two subproblems, one for local pages and the other for global pages. Such a division is natural and convenient, grouping pages based on access cost, management cost, and who benefits from the pages. Access cost is uniform within each group, local pages are accessed with one cost and global pages with another. Management cost can be low for local pages because local mechanisms are possible; global pages require more costly distributed management. Finally, pages exist in local memory to benefit the local node; global pages benefit only remote nodes. Our algorithm exploits this grouping in three key ways, discussed below.

4.3.1 The Replication Policy

The first important benefit of the local/global division is that we use it to determine whether a faulted page should be replicated or moved. Recall that we assume that network access cost is uniform throughout the network. This assumption implies that a global page can be accessed with equal cost regardless of which node stores the page. We are thus free to move global pages from node to node without impacting the performance of future accesses to those pages. This freedom permits the following replication strategy. When a *global* page is referenced, we *migrate* it to the faulting node's local memory, removing it from global memory on its previous home node. Subsequent access to the page is handled from the faulting node's local memory as cheaply as from the global memory of the original home node. Local pages, on the other hand, are always *replicated*. Replication is called for in this case because we assume that pages are in local memory only if they benefit local programs. Migrating a local page away from a node is thus unacceptable because doing so would harm local performance.

The assumption of uniform access cost is important to our decision not to replicate global pages. If remote access were not uniform, moving a global page might also increase its access cost from other nodes. To avoid this cost increase, replication of some global pages might be necessary.

4.3.2 The Tension Between Local Performance Loss and Global Gain

The local/global classification acts as a metaphor for the tension between local and remote needs; a tension that arises from competition for workstation resources between local and remote programs. Increasing the size of global memory to hold more of a program's pages requires decreasing the size of local memory on some other node, possibly slowing programs on that node. The goal, of course, is to do this only when the benefits outweigh the costs. Improving the performance of remote programs while sacrificing local performance on some nodes, for example, may result in better overall system performance and may therefore be desirable. Each workstation, however, typically has a primary user who expects priority use of the workstation's resources. The degree to which this user is willing to sacrifice local performance to improve global throughput is a matter of taste. The local/global classification makes it easy to consider *owner priority* and other fairness issues as part of the replacement policy. The algorithm can give priority to local programs by favoring the replacement of global pages over local pages. At the extreme, global pages could be favored to the extent that only an *idle* workstation would donate any of its memory to remote programs. Alternatively, some users may be willing to tolerate *moderate* degradation of local performance at certain times in order to gain the benefit of a larger global memory at other times; global memory will be larger the more aggressive the policy is about converting local memory into global memory.

4.3.3 Advantages of Separating Local and Global Management Mechanisms

The division of memory into local and global regions also simplifies algorithm design by allowing us to make policy decisions along the lines of the cost functions listed above, separating the management of local and global regions as much as possible. The management algorithm first chooses the least valuable page in each region and then in a separate process compares these two pages to find the least valuable page on the node and then in the network as a whole. All but the last step of this process can be completed locally. We discuss issues related to each of these decisions in the next section.

4.4 Selecting the Least Valuable Page

As mentioned above, the description of the algorithm assumes the existence of four cost functions: the first for choosing the least valuable local page on a node; the second for choosing the least valuable global page; the third for comparing these two pages to choose the overall least valuable page on a node; and the fourth for comparing the least valuable page on each node to find the least valuable page in the network. Taken together these functions define the objective of the replacement algorithm. To define the functions several difficult policy questions must be answered (e.g., how to compare local pages with global pages). In an implementation of the algorithm these policy choices will be made by heuristics, such as LRU, for estimating future access patterns or for comparing estimates of access time and cost.

In this section we will review the key issues and policy questions related to defining each of the cost functions. We describe the heuristics used in our GMS prototype implementation in Section 6.3.

4.4.1 Choosing the Least Valuable Local Page

To pick the least valuable local page on a node, the system must have a method for computing the *replacement cost* of every local page. This computation requires requires (1) a heuristic for predicting the time of next reference to each page, (2) knowledge of which local pages are replicated, and (3) a heuristic for comparing a unique and a replicated page. The first requirement is the same as for standard non-hierarchical (i.e., local) memory management; the Least Recently Used (LRU) heuristic is commonly used (see Section 2.2.1). The second requirement is easily achieved as we will see in Chapter 6. The third requirement, however, is more difficult.

The system should clearly favor replacing replicated pages over unique pages; the key question, however, is by how much. Replicated pages are better replacement victims because a subsequent access can read the replaced page from remote memory while replacing a unique page would require reading from disk. Given two pages with the same *next-access time*, one unique and one replicated, the replicated page should thus be replaced. The difficultly, however, is to decide which page to replace when the replicated page is next accessed *sooner* than the unique page. Making this decision requires a currency for comparing next-access time and access cost. For example, if remote memory is ten times faster than disk, the unique page should be replaced when doing so results in at least ten fewer future faults. It is difficult, however, to predict how many faults a particular replacement choice will cause. With LRU, for example, the oldest page is predicted to be the best victim; but, there is no reliable way of determining the effect of choosing, say, the twentieth oldest page instead.

4.4.2 *Choosing the Least Valuable Global Page*

Fortunately, choosing the least valuable global page on a node, or in the network, is straightforward. Every page in global memory has the same replacement cost (i.e., a replaced global page must always be read from disk). As a result, the least valuable global page is easily chosen by determining or predicting the page that is not accessed for the longest time in the future (e.g., the LRU page).

4.4.3 *Comparing the Least Valuable Local and Global Page*

Choosing between the least valuable local and global pages is difficult for two reasons. First, similar to the choice between unique and replicated pages, the choice involves multiple replacement costs (e.g., replicated pages can be re-read from remote memory but global pages must be re-read from disk). Second, there are several important factors that favor keeping local pages.

There are two cases to consider: either the least valuable local page is *unique* or it is *replicated*. If the local page is unique, the global page should be replaced because its replacement cost is significantly lower; that is $C_d - C_r \ll C_d - C_l$. It might at first appear that, for similar reasons, a replicated page should be replaced before a global page (i.e., because $C_d - C_r > C_r - C_l$). This comparison is less straightforward, however, because of other factors that favor keeping local pages longer than global pages, even if the local pages are replicated.

Choosing Between a Replicated Local Page and a Global Page

Three factors favor keeping replicated local pages longer than global pages.

1. Local pages are more likely to be accessed sooner.

2. Workstation-owner priority argues for favoring local pages.

3. Local slowdown due to replacing a replicated page is greater than remote speedup due to replacing a global page.

We will now consider each of these factors in detail.

Local Pages are More Likely to be Accessed Sooner

A useful heuristic for predicting next-access time is to assume that, other things being equal, local pages are more likely to be accessed than global pages, because global pages have already been replaced once from some local memory. Note that this heuristic might be a poor predictor, however, if some nodes have very small memories or some nodes run programs with significantly greater memory needs than other nodes. Local pages would tend to be evicted rapidly from both of these types of nodes, resulting in global pages that are just as valuable as the local pages from other nodes.

Workstation Owners Should Get Priority

We have already mentioned that "sociological" issues and fairness concerns might lead us to have an *objective function* other than simply maximizing system throughput. In particular, workstation owners expect to get priority use of their workstations. This priority can be incorporated into the replacement algorithm by favoring local pages. Replacing a replicated page might improve global throughput; but, it does so by slowing local performance.

If a Replicated Page is Replaced, Local Slowdown is Greater Than Remote Speedup

One way to measure the impact of a global management algorithm is to compute the speedup (or slowdown) of programs compared to running those programs locally (i.e., without global memory). This ratio is important because it allows us to compare the impact of global management on each workstation in the network; in particular, we can compare the slowdown of some nodes with the speedup of others in order to evaluate the fairness of the algorithm.

Consider the choice between replacing either a local-replicated page or a global page on a node where both pages are accessed at the same point in the future, the replicated page by a local program and the global page by a remote program. Replacing the global page favors the local program and slows the remote program, because its future access to the global page will be forced to come from disk, instead of remote memory. Similarly, replacing the

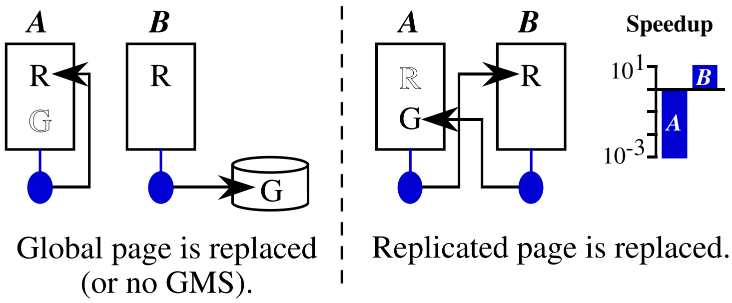


Figure 4.4: Example of the choice between replacing global page G (on left) or local-replicated page R (on right). Node A accesses R and node B accesses G. If R is replaced, A's slowdown exceeds B's speedup.

replicated page favors the remote program and slows the local program, because its future access to the replicated page will be forced to come from remote memory, instead of from local memory. A careful analysis of this choice shows that replacing the replicated page slows local performance more than keeping the global page speeds remote performance, even though the execution *time* difference favors replacing the replicated page. To see why, consider the following example.

Figure 4.4 shows the choice between replacing a replicated page (R) and a global page (G) on node A and shows the impact of this choice on the performance of nodes A and B. Initially, node A stores both R and G and runs a program that references R. Node B stores the other copy of R and references G. In a system without global memory, A's read cost is C_l and B's cost is C_d. In a system with global memory, if G is replaced, nodes A and B run the same as they do without global memory; this case is shown on the left. If R is replaced instead, A's access cost increases to C_r and B's cost decreases to C_r, because both now read their pages from remote memory; this case is shown on the right. Replacing R should thus be favored only if A's slowdown is less than B's speedup (i.e., if $(C_r : C_l) < (C_d : C_r)$). Recall from Section 2.1.2, however, that this is not the case; in fact, if R is replaced, the slowdown of A is roughly three orders-of-magnitude more than the speedup of B, as shown in the Figure (i.e., $C_r : C_l \gg C_d : C_r$). This analysis thus argues for replacing global pages before local-replicated pages, even though doing the opposite results in better global throughput (i.e., the access cost is $C_l + C_d$ if G is replaced, and is $C_r + C_r$ if R is replaced).

Summary

We have seen that the choice of the least valuable page depends on the replacement algorithm's objective function in subtle ways. If global throughput is the only objective and future access time is known or accurately predicted, the best choice is to replace first a local-replicated page, then a global page, and finally a local-unique page. There are, however, three key arguments for favoring the replacement of global pages over local-replicated pages: (1) local pages are more likely to be accessed sooner than global pages, (2) workstation ownership priority argues for favoring local pages, and (3) replacing a local page slows local performance more (as a ratio) than it speeds up remote performance.

4.4.4 Choosing the Least Valuable Page in the Network

Once we have in place a means of finding the least valuable page on every node, the task of finding the *least valuable page in the network* is relatively straightforward. We can use the same cost functions used on each node to compare pages from different nodes. The page with the lowest replacement cost is the globally best replacement victim.

4.5 Coherency and Node Failures

Node failures do not cause data loss in global memory, because all pages sent to global memory are *clean*. When a page is chosen for replacement, if that page is dirty, a copy of the page is written to disk in parallel with its transmission to global memory. We do not change the number of disk writes that occur; our system allows a disk write to complete as usual but promotes that page into the global cache so that a subsequent fetch does not require a disk read. If a node housing a requested remote page is down, the requesting node simply fetches the data from disk.

Likewise, since our algorithm deals with only clean pages, coherence semantics for shared pages are the responsibility of the higher-level software that creates sharing in the first place. For instance, in our system shared pages occur when nodes access a common NFS file (e.g., an executable). Thus, the coherence semantics seen by the users of our system are no stronger and no weaker than what NFS provides.

4.6 Summary

In this chapter, we described the GMS *global memory management* algorithm. The algorithm is driven by page faults and each fault either swaps the storage node of two pages or, if a new network page frame is needed, replaces the *globally least valuable* page in the network. Each workstation's memory is divided into two regions, one that stores local pages and the other that stores global pages. The boundary between local and global memory moves dynamically in response to workstation and network workloads; idle workstations tend to store mostly global pages and active workstations mostly local pages. The division of memory into local and global regions is useful in several ways: it forms the basis of our replication policy, which replicates local pages and migrates global pages; it simplifies the handling of tensions between local and remote demands on workstation resources; and it divides management into separate subpolicies for local and global regions. Separating management into these subpolicies allows replacement to follow a four step process, three of which can be completed locally. First, the least valuable local page is chosen, taking care to favor replacing replicated pages over unique pages. Second, the least valuable global page on a node is easily selected. Third, the least-valuable local and global pages are compared to choose the overall least valuable page on the node, favoring the replacement of global pages over local pages. Fourth, a distributed algorithm is used to find the globally least valuable page from among the least valuable pages on every node.

In the next chapter we will describe how GMS efficiently finds and replaces the globally least valuable page in the network. We will also review several alternative approaches.

Chapter 5

MAKING GLOBAL DECISIONS EFFICIENTLY

The theory of probabilities is at bottom nothing but common sense reduced to calculus.

— Pierre Simon de Laplace, 1820

The description of the GMS global memory management algorithm in the previous chapter pointed to a key issue, how to find the globally least valuable page in the network. Finding this globally least valuable page requires information from every node in the network about the state of its memory. It is obviously impossible, however, to maintain complete global information at every instant; doing so would require network-wide synchronous communication for processing each page fault. Clearly, for our algorithm to be useful, we must find an efficient way to maintain and use global information.

In this chapter, we will examine the question of how to efficiently use global information to make the critical memory-management decision of where a node sends the pages it replaces from its memory. This is the critical question for all remote-memory management approaches and is the main point on which these approaches differ. We will begin this chapter by describing the problem and alternative solutions in general terms. We will then describe how our algorithm uses global information. Following this, we will look at three alternative approaches and we will conclude with a discussion of possible extensions and simplifications to our algorithm.

5.1 Overview of Global Decision Making

In this section we describe the problem of finding the global oldest page as an instance of a more general problem we will term the *distributed-decision problem*. Specific instances of this problem vary according to four key characteristics, which we describe below. We also describe three solution strategies, which vary in effectiveness depending on the characteristics of particular problem instances. With this background in place, we show

how the characteristics of global memory management suggest a strategy along the lines that we follow in our algorithm, which is describe in the next section.

5.1.1 General Statement of Distributed-Decision Problem

The *distributed-decision problem* can be stated generally as follows. Each node executes an algorithm that periodically makes global decisions. Decisions are made in two parts. First, the node considers state information from every network node to select a target node. Second, the node takes an action than transforms its state and that of the target node. Importantly, a state change may invalidate pending target-node selections; that is, the completion of step two by a node may invalidate incomplete decisions on other nodes, because those decisions may be based on outdated state information.

For global memory management, a distributed decision is needed when a page fault requires the replacement of the network's globally least valuable page (Cases 3 and 4 in Section 4.1). First, the faulting node locates the node with the globally least valuable page by evaluating the state of every memory in the network. Second, the faulting node's replacement victim is forwarded to the selected node where it replaces the least valuable page. Note that if multiple nodes follow this procedure at the same time, they may all select the same least-valuable page for replacement. If this happens, then only one of these nodes will end up taking the correct action, because the first node to forward its page invalidates the selection made by all of the other nodes.

There are two difficult aspects of this problem. The first is that information is needed from every node in the network. Gathering this information can be expensive; if every node collects information from every other node, the total number of messages is $O(N^2)$. The second problem is that some form of global coordination may be needed to prevent one node from invalidating concurrent decisions made by other nodes. That is, since the action taken by one node invalidates decisions made by other nodes, it may be necessary to serialize the process so that each node finds the globally least-valuable page and replaces it before the next node makes its choice. Global serialization such as this would be expensive, requiring synchronous message exchanges and possible long delays for each local replacement.

Clearly we need a way to make the decision process less costly, either by collecting global information less frequently or by collection information from fewer nodes.

5.1.2 Key Characteristics of Decision Problems

The choice of solution strategy depends on the characteristics of the particular instance of the decision problem, including:

1. the frequency of decision making;

2. the volatility of state information over time;

3. the distribution of state information across the network; and

4. the overheads involved in computing the function, exchanging information, and making a mistake.

The first point is that if decisions are made frequently, the cost of each decision must be low. When decisions are made very frequently, it may be impossible to collect global information for each decision. Instead, it may be desirable to cache global information at each node and update the cached information less frequently.

Second, if state information is relatively stable over time, periodic sampling and caching of information should be effective. The more volatile the information is, the more frequently nodes must be sampled. If state changes in predictable ways, it might be possible to delay sampling by extrapolating between collections.

Third, the way that state information varies from node to node is a critical consideration. At one extreme, if state of every node is identical, it is only necessary to sample a single node to induce the state of the system. If, one the other hand, state information varies widely from node to node in a way that cannot be predicted, a large number of nodes must be sampled. Once again, if state varies in a predictable way, it might be possible to extrapolate state from a small sampling of nodes. More generally, it should be sufficient to sample a subset of the network to get a probabilistic answer that is within some acceptable bound of being accurate.

The final factor we must weigh in the design of a distributed-decision algorithm is what are the computational and communication overheads associated with the algorithm. The goal, of course, it to make the best possible decision with the lowest possible overhead. The higher the overhead, the more of an approximation we will be forced to accept. In our particular case, it is especially important to avoid overheads that slow performance beyond that seen when global memory is not used at all. For example, programs on a node that is actively using all its memory should not be slowed by global management overheads. One danger for such nodes is that mistakes made by an approximate algorithm might have a significant impact. If, for example, local replacement victims are mistakenly forwarded to busy nodes, performance on those nodes will suffer. The cost and frequency of such mistakes are important factors.

We can now see three possible ways to lower overhead. The first is to cache information locally in a way that we can tolerate it being out-of-date; the cost of collection can thus be amortized over multiple decisions. The second is to extrapolate from a sample collected from a subset of nodes; the cost of each collection is thus low. The third is to make local decisions blindly and correct mistakes at the target node. Approaches for global decision making might combine these techniques in various ways and to varying degrees. In the next section we will discuss three basic approaches.

5.1.3 Approaches to Efficient Global Decision Making

Solutions to the distributed-decision problem follow one of three possible strategies, listed below; hybrid combinations are also possible.

1. Collect precise information periodically but tolerate it being somewhat out of date.

2. Collect information from a few nodes with each fault and use this imprecise information to estimate the correct choice.

3. Make a blind local decision and correct mistakes at the target node if the blind choice is wrong.

Solutions of the first type work well when state information changes slowly over time or when changes can be predicted and incorporated into locally cached information. Solutions of the second type work well when state information can be estimated from a small sampling of nodes. Note that the frequency of decision making has an impact on the tradeoffs between the first two approaches. If decisions are frequent, the first method might be preferred because each decision is made locally, and therefore cheaply, using locally-cached information. If decisions are less frequent, the second method might be preferred because less frequent decisions permit us to have a somewhat higher per-decision cost. In addition, in the first method, cached information must be updated periodically to keep it from getting too far out of date. Frequent decisions may allow the cost of collecting state information to be amortized over a large number of decisions. The third strategy is to choose a target node blindly and have the target node detect and correct mistakes. This approach works well when mistake cost is low or when state information is distributed in such a way that blind choices are usually correct.

5.1.4 Choosing a Decision Strategy for Global Memory Management

Let us now describe global memory management from the perspective of the four decision-problem characteristics listed in Section 5.1.2. First, replacement decisions are frequent on nodes that are actively faulting; these nodes may incur a few hundred faults per second, each fault requiring a global replacement decision. Second, the volatility of state information over time depends on what global information is collected from each node. If nodes know only the age of another node's oldest page, for example, this information becomes out-of-date, and thus useless, after a single replacement on that node. Global information that summarizes the age of a large number of each node's pages, on the other hand, should be able to remain useful for several seconds because, although faulting rates are high, only a small percentage of total memory is replaced in each multi-second interval. Third, the distribution of memory ages across the network is highly variable. A network contains three types of nodes: idle nodes, which have mostly very old pages; heavily-loaded active nodes, which have mostly very young pages; and lightly-loaded active nodes, which use only a portion of their memories and thus have some young pages and some old pages. The distribution of these three types of nodes varies over time — depending on network workloads — from lightly-loaded, consisting of mostly idle nodes, to heavily-loaded, consisting of only a few idle nodes. Global management must thus be able to adapt readily to a wide variety of idleness distributions; heavily-loaded networks may be particularly challenging because all of the idle memory in the network is located on only a few nodes. Fourth, our primary concern regarding algorithm overheads is to minimize the impact on active nodes, even if this might mean a somewhat increased overhead for idle nodes. This factor argues for shifting collection responsibilities to idle nodes and for avoiding algorithm mistakes that forward replaced pages to active nodes.

To summarize, the strategy for managing global memory should have the following characteristics.

- Per-decision overhead should be low.
- Global information should summarize a large number of each node's pages.
- A wide variety of age distributions across the network should be tolerated.
- Collection overhead imposed on active nodes should be minimal.
- Mistakes that forward pages to active nodes should be avoided.

The GMS memory management algorithm was designed with these characteristics in mind. It follows the first solution approach outlined in Section 5.1.3: global age information is distributed to nodes periodically and is then used to make replacement decisions for the next five to thirty seconds, thus amortizing collection cost over a large number of replacements. The global information we use summarizes the age of all of the pages on every node, thus ensuring that it will be useful for a large number of replacement decisions and that idle pages will be easily found even when distributed across only a small number of nodes. Active-node overhead is minimized using a centralized collection approach in which an idle node collects local age information from all nodes, generates a global-age summary, and redistributes the global information back to the nodes. This centralized approach reduces the message overhead imposed on active nodes by increasing the overhead of a single idle node; active nodes receive global age information by a single message exchange with that node. Finally, our decision strategy takes great care to ensure that pages are forwarded to the right nodes, thus avoiding slowing an active node by erroneously forwarding pages to it.

The next two sections describe how the GMS algorithm solves the global-memory-management distributed-decision problem and the key issues for our solution. Following this, we will describe and evaluate three alternative solutions that explore different points in the decision-problem design space.

5.2 The GMS Algorithm for Maintaining Global Information

The *Global Memory System* (GMS) algorithm collects load information from all nodes periodically, every five to thirty seconds, and redistributes a global summary that nodes use to make their page forwarding decisions. This global summary is formulated to be valid until information is collected again (i.e., we use a type-one approach from Section 5.1.3). The objective of our algorithm is to provide a reasonable tradeoff between the accuracy of information that is available to nodes and the efficiency of distributing that information. The key issue is guaranteeing the validity of the node information and deciding when it must be updated.

5.2.1 Page Replacement Metric

Thus far we have discussed our algorithm by saying that it replaces the *least valuable page* in the network. Here we describe how we do this. This description is simplified if we make the replacement choice metric more concrete. We adapt the LRU (Least Recently Used) heuristic to global management. Each page is assigned an *age* that represents the time since the page was last referenced. Replicated pages are aged faster than unique pages and global pages faster than local pages (either replicated or unique), thus simplifying comparisons between unique, replicated, and global pages. We will provide more details about this use of page ages when we describe our prototype implementation in Chapter 6.

5.2.2 Algorithm Basics

Our algorithm divides time into epochs; global load information is collected at the beginning of each epoch and is used to make all replacement decisions during the epoch. Each epoch has a maximum duration, T, and a maximum number of network-wide replacements, M, that will be allowed in that epoch. The values of T and M vary from epoch to epoch, depending on the state of global memory and the workload. A new epoch is triggered when either (1) the duration of the epoch, T, has elapsed, (2) M global pages have been replaced, or (3) the age information is detected to be inaccurate. Currently, each epoch is on the order of 5–30 seconds. We will say more about how we determine epoch length in a moment.

At the start of each epoch, every node sends a summary of the ages of its local and global pages to a designated *coordinator node*. Using this information, the coordinator computes the *node weight array*, W, such that for each node i, out of the M oldest pages in the network, w_i reside in node i's memory at the beginning of the epoch. The coordinator also determines the minimum age, *MinAge*, that it estimates will be replaced from the cluster (i.e., sent to disk or discarded) in the new epoch. The coordinator sends the weights W and the value *MinAge* to all nodes in the cluster. In addition, the coordinator selects the node with the most idle pages (the largest w_i) to be the coordinator for the following epoch.

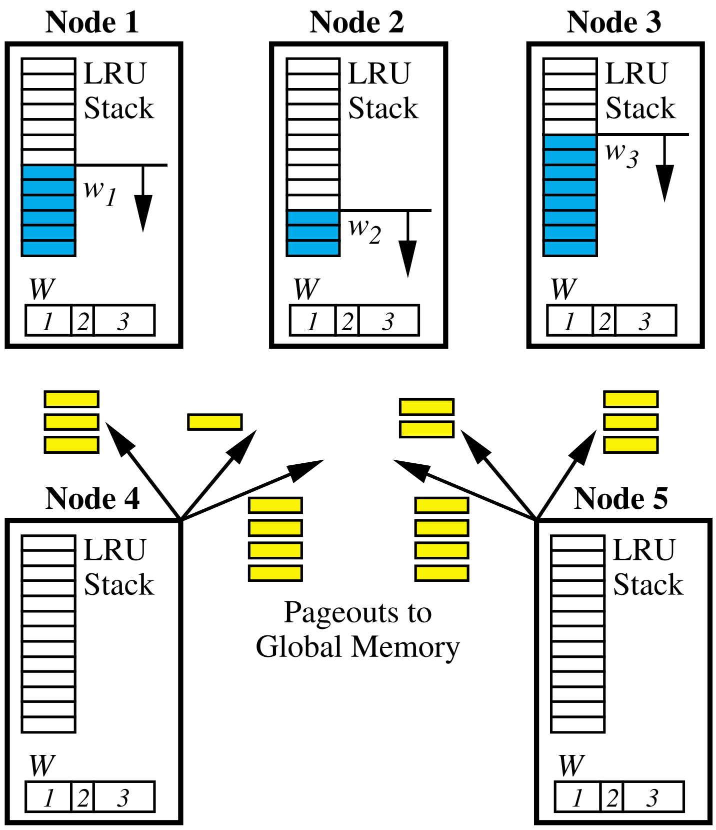


Figure 5.1: Sample operation of the GMS algorithm with three idle nodes and two active nodes.

During an epoch, when a node P must evict a page from its memory to fault in a page from disk or to replicate a remote-memory page (Cases 3 and 4 in Section 4.1), it first checks if the age of the evicted page is older than *MinAge*. If so, it simply discards the page (since this page is expected to be discarded sometime during this epoch). If not, P sends the page to node i, where the probability of choosing node i is proportional to w_i. In this case, the page discarded from P becomes a global page on node i, and the oldest page on i is discarded.

Figure 5.1 shows a sample operation of our algorithm in a network of five nodes: nodes 1–3 have old pages and nodes 4 and 5 are actively forwarding pages to global memory. Each node's pages are shown in descending LRU order (oldest pages at the bottom) and the M globally oldest pages are darkly shaded. Each node has a copy of W, which reflects the current epoch's distribution of old pages among the five nodes (i.e., node 3 has the most, node 1 is next, and node 2 has only a few). Nodes 4 and 5 are forwarding pages to nodes 1–3 according to this distribution (the lightly shaded pages).

Our algorithm is probabilistic: on average, during an epoch, the ith node receives w_i/M of the evictions in that epoch, replacing its oldest page for each one. This yields two useful properties. First, our algorithm approximates LRU in the sense that if M pages are discarded by global replacement during the epoch, they are the globally oldest M pages in the network. Second, it yields a simple way to determine statistically when M pages have been replaced; i.e., when the node with the largest w_i receives w_i pages, it declares an end to the epoch.

To reduce the divergence from strict LRU, it is thus important to keep the duration of the epoch T and the value of M appropriate for the current behavior of the system. The decision procedure for choosing these values considers (1) the distribution of global page ages, (2) the expected rate at which pages will be discarded from the network, and (3) the rate at which the distributed age information is expected to become inaccurate. The latter two rates are estimated from their values in preceding epochs. Roughly speaking, the more old pages there are in the network, the longer T should be (and the larger M and *MinAge* are); similarly, if the expected discard rate is low, T can be larger as well. When the number of old pages in the network is too small, indicating that all nodes are actively using their memory, *MinAge* is set to 0, so that pages are always discarded or written to disk rather than forwarded.

The remainder of this section is devoted to discussing the following algorithm details:

- the algorithm for computing a new epoch,
- scaling the algorithm to large networks,
- how the algorithm picks the target duration of the new epoch,
- the three factors that can signal the end of an epoch, and
- how the algorithm estimates local and global faulting rates.

5.2.3 Computing a New Epoch

This section provides additional details about the algorithm for computing the global node weights used in each epoch.

To begin a new epoch, the coordinator sends a message to every node requesting current age information. Nodes respond with an *age distribution array*. The array divides the pages on the node into 64 age categories; each element specifies the number of pages on the node that fall into that age category. In our prototype, global age is kept with a granularity of one minute and so element i of the array is the number of pages that are between i and $i+1$ minutes old (to simplify this discussion we will refer to these pages as being of age i).

The coordinator collects age arrays from all of the nodes and assembles an *aggregate age distributed array* that categorizes the ages of all the pages in the network. The aggregate age array is really two arrays, A_i and D_i^n that categorize the age of all of the pages in the network.

$$\begin{aligned} A_i &= \text{total number of network pages of age } i \\ D_i^n &= \text{number of pages of age } i \text{ on node } n \end{aligned}$$

Once A_i and D_i^n are computed, the coordinator follows the procedure outlined in Section 5.2.4 to determine T and M. The coordinator then uses T and D_i^n to compute *MinAge* and W, the node-weight array, and sends a message containing *MinAge* and W to every other node.

The work performed by each node, other than the coordinator node, is quite small. Nodes keep a running total of local and global page ages. As a result, the *local age distribution array* is "computed" by simply copying it from this running-total data structure; this procedure is fully explained in Section 6.3.

The coordinator performs $O(N)$ work to build the *aggregate age array* from all of the arrays sent by each node. Computing T and M involves iterating through the aggregate age array, $A_{0..63}$. Finally, computing the node-weight array is another $O(N)$ computation. Measurements of our prototype implementation are presented in Chapter 7. These measurements show that, for moderate values of N (say, less than 1000), these overheads are quite modest.

Epoch initiation thus requires $3 \times N$ messages: request, reply, and result. Request and reply messages are small but the result message contains the node-weight array which is $O(N)$ in size. For a 1000-node network, for example, the coordinator will handle 3000 messages totaling roughly 4-Kbytes, per epoch.

5.2.4 Scalability of Epoch Computation

In the algorithm as described so far, a central node collects information from every node, processes this information, and disseminates global information back to every node. This centralized computation limits the effective use of the algorithm to networks of a few hundred nodes. For larger networks, however, the algorithm can be naturally extended by subdividing the network into smaller groups of nodes with an instance of the algorithm running in each group. The only potential disadvantage of this subdivision is that a node in one group is not able to utilize memory from a different group. The groups, however, can easily contain hundreds of nodes and thus have considerable network memory available. The key issue then, is to ensure an even distribution of idle and active nodes among the groups. Such as distribution can be maintained by a hierarchical algorithm that communicates with the epoch coordinators in each group to monitor memory use and to move nodes between groups to maintain the desired balance.

5.2.5 Choosing the Target Epoch Duration

This section describes the process our algorithm follows to choose the new epoch's expected duration (T) and replacement-count (M).

We would like to choose the longest possible epoch that results in a good approximation of global LRU. Long epochs are preferred, because they result in lower overall overhead. A long epoch, however, may result in a poor LRU approximation; longer epochs include more pages in the epoch's *replacement group* (i.e., the group of pages targeted for replacement during the epoch and summarized by W). Recall that pages in the replacement group are replaced randomly during the epoch. Ideally, this randomness will be insignificant compared with the overall intra-epoch LRU behavior of the algorithm.

Two characteristics of the replacement group indicate whether an epoch will approximate LRU well: the age variance of pages in the group and the age of the group's youngest page. If the age variance is high, pages of largely different age will be replaced randomly during the epoch. If the youngest page in the epoch is fairly old this might be acceptable, because age differences are less important for predicting future access with older pages. If the youngest page is very young, however, a very low variance is important.

Our algorithm quantifies the quality of an epoch, Q_i, that includes pages of age $\geq i$, as follows.

$$\begin{aligned}
A_i^{sd} &= \text{std_dev}((A_i, i), (A_{i+1}, i+1), \ldots, (A_{63}, 63)) \\
Q_i &= \left(0.5\left(\frac{63 - 2 \times A_i^{sd}}{63}\right) + 0.5\left(\frac{i}{63}\right)\right)^2
\end{aligned}$$

The first term of Q_i represents the quality of the age distribution and the second the quality of the age of the youngest page; each are given equal weight (by multiplying them by 0.5). Quality ranges from 0 to 1; a quality rating of 1 represents a perfect epoch (i.e., a perfect LRU-approximation).

The procedure for selecting T for the new epoch considers the global age distribution, A, starting with A_{63}. Each step of this selection process considers a candidate replacement group, containing the pages in $A_i, A_{i+1}, \ldots, A_{63}$, for step $63 - i$. The candidate group is then evaluated as follows. We first compute Q_i for the group; this yields a number between 0 and 1. We then use Q_i to determine the duration of the shortest epoch we are willing to tolerate for this group, call this T_i.

$$T_i = \max(5, 30 \times Q_i)$$

The candidate replacement rate, $(A_i + A_{i+1} + \cdots + A_{63})/T_i$, is then compared with the predicted rate for the next epoch, R, which was computed using hysteresis as described in Section 5.2.7. If a group is sufficient to handle the predicted rate, the procedure terminates. Otherwise, the next-younger group of pages from A are added to the group and a new Q is computed. Each step results in a larger and typically lower-quality group; a larger group means longer epochs are *possible* and a lower-quality group means that shorter epochs are *tolerated*, thus ensuring that the algorithm does converge on an acceptable replacement group.

Once a good replacement group has been found, T and M are computed as follows. M is assigned to be the number of pages in the replacement group: $M = A_i + A_{i+1} + \cdots + A_{63}$, and $T = M/R$.

Intuition Behind Epoch Choice Strategy

It might appear that there is a bit of black magic involved in computing T_i and Q_i. The computation is a heuristic and as such its details are less important than the fact that it satisfies key characteristics needed to select an acceptable epoch; other functions that satisfy these characteristics might do as well (or better).

There are three key characteristics for the computation of T_i and Q_i. First, T_i has the correct value in the boundary cases of the best and worst epochs; the best epoch allows the maximum duration and the worst allows the minimum duration. Second, with each step of the selection algorithm, the size of the replacement group increases and T_i decreases, increasing the epoch's *allowable* replacement rate. Third, the quadratic order of Q provides the useful property that we can be aggressive about trying to get a long epoch for high-quality replacement groups, but back off from that aggressive stance as the quality rating drops. For low-quality groups, we are more aggressive about *shortening* epochs to avoid further degradation in the group's quality. The key observation is that an epoch of $Q = 0.5$, compared with $Q = 1$, is much worse than a epoch duration of 15 s, compared to 30 s.

5.2.6 Triggering a New Epoch

A new epoch is triggered by one of the following events:

- The epoch duration expires (after T seconds).
- The epoch coordinator estimates that M replacements have occurred.
- Any node sees more than the expected number of faults that convert global pages to local pages.

As mentioned above, the node with the largest replacement weight, w_i, triggers a new epoch when it receives w_i forwarded pages. If w_i is sufficiently large, this provides a good statistical estimate that M replacements have occurred throughout the network. The new epoch is actually triggered when slightly fewer than w_i pages are received so that the computation of the new epoch will overlap with the end of the current epoch.

We stated earlier that our algorithm has the LRU-like behavior that after M replacements have occurred, the M oldest pages in the network have been replaced. Notice, however, that what is actually guaranteed, so far, is that after M replacements have occurred, each node will have received w_i forwarded pages and that a new epoch will have been triggered. These actions result in the desired LRU behavior only if pages in the replacement group are consumed exclusively by forwarding through the weighted distribution. There are, however, two other ways that these pages are consumed.

First, a page fault on a node with global pages will convert the oldest page on the node into a young local page (i.e., the old page is replaced with the faulted page). If this old page in one of the M oldest in the network, we call this a *victim conversion*.

Second, an access to a page (either locally or remotely) resets the page's age. An access to one of the of M oldest pages is called a *access conversion*. Notice that it is possible for a fault to cause two conversions: a victim conversion on a local node and an access conversion on a remote node.

When either type of conversion occurs, one of the M oldest pages is consumed on a node i without being counted against w_i. Thus, when the epoch ends, each node will have replaced its $w_i + c_i$ oldest pages, where c_i is the number of conversions that occurred on that node in the current epoch; a total of $M + \sum_{i=0}^{N} c_n$ pages will have been replaced network wide. Clearly, this is not what we want to have happen. In particular, the c_i extra pages replaced on each node could end up being some of the *youngest* pages in the network. We safeguard against the problems caused by both access and victim conversions by allowing any node to terminate an epoch early if more than a threshold number of conversions occur on that node during the epoch, as described below.

Safeguarding Against Victim and Access Conversions

The algorithm safeguards against the effects of victim- and access-conversions in the following way. The epoch initiation procedure on each node uses hysteresis to predict the number of conversions that will occur on that node during the epoch. This number of the node's oldest pages are hidden from the global information provided to the coordinator. If the prediction is correct, the w_i oldest pages on the node are correctly replaced at the end of the epoch. If the rate is overestimated, the node will end up holding pages that are older than pages replaced on other nodes during the epoch. If the rate is underestimated, the node risks replacing very young pages, as discussed above. To avoid replacing more than w_i pages, each node monitors the actual conversion rate and when the predicted number have occurred, it triggers a new epoch by sending a message to the coordinator. As before, the new epoch is triggered when the number of conversions is slightly less than the predicted value so that epoch initiation will overlap with the end of the current epoch.

5.2.7 *Estimating Local and Global Faulting Rates*

As we have seen, key to our algorithm is its ability to estimate (1) the global replacement rate, R, used to compute T and M, and (2) the victim- and access-conversion rates discussed above. The algorithm uses hysteresis to estimate these rates from the actual rates measured in preceding epochs. It uses the measured values for the preceding two epochs and the value for the last two seconds of the last epoch. The predicted rate is computed by taking the highest of the three values if the rate is increasing or the mean if the rate is decreasing. We thus favor overestimation, a conservative approach as we will see in the next section.

5.2.8 *Summary*

In summary, using our algorithm each node estimates the number of local conversions predicted to occur in an epoch and hides this number of pages from other nodes. The remaining pages are summarized by an age array that is collected from each node by a centralized coordinator node. The coordinator combines the local age information into a global summary of the location of the oldest M pages in the network, where M is the number of pages predicted to be replaced in the next five to thirty second epoch. During the epoch, nodes forward pages according to the global-age distribution they share and each node monitors the number of local conversions that occur. An epoch is terminated when any node performs the predicted number of conversions or when M replacements have occurred network-wide. At the end of an epoch, the M globally oldest pages in the network will have been replaced by pages forwarded from faulting nodes.

5.3 Issues for the GMS Global-Information Algorithm

Our approach to global decision making is based on several assumptions regarding the actions of our algorithm and the system as a whole. If these assumptions hold, our algorithm provides a close approximation of global LRU replacement with very little overhead. In this section, we will examine how our assumptions can go astray and how our algorithm behaves under these circumstances. In most cases, the algorithm has been designed to handle these adverse situations gracefully. A key attribute of our algorithm, in fact, is that its behavior is stable and easy to analyze.

5.3.1 Underestimating Global-Replacement or Conversion Rates

We have already discussed how our algorithm triggers an early end to an epoch if either the global-replacement rate (R) or any node's victim- or access-conversion rate is underestimated. This early epoch ending, however, can be problematic in two ways. The first is that the resulting epoch may be shorter than the desired minimum (5 seconds in our prototype). While a single short epoch is not that alarming, a long sequence of very short epochs would increase the algorithm's average overhead to unacceptable levels. Such a sequence, however, is made unlikely by our prediction method, which favors overestimation. Our method ensures that a sequence of short epochs occurs only if each epoch has a significantly higher rate than preceding epochs, thus limiting the length of such a sequence. The second potential problem is that, because nodes continue to receive forwarded pages while the next epoch is being computed, some nodes might receive more forwarded pages than they should. The algorithm attempts to safeguard against these overruns by ending the epoch somewhat before the predicted number of replacements or conversions is actually observed. If the actual rate is significantly higher than the predicted rate, however, overruns may begin to occur before the epoch computation completes. When this occurs, overrun nodes have the choice either of dropping the extra forwarded pages they receive or of replacing their next locally-oldest pages; the result, in both cases, is the replacement of younger pages than planned. Once again, however, by favoring overestimation, the algorithm makes it unlikely that the predicted rate will remain significantly lower than the actual rate for long.

The key to avoiding epoch overruns then is to trigger a new epoch while there are still enough of the M oldest pages on every node to absorb pages forwarded while the new epoch is being computed. If epoch-computation time is a small fraction of typical epoch-duration time, it is feasible to reserve enough pages to handle even the worst-case replacement rate for this last fraction of each epoch. If epoch computation is too slow, however, the number of reserve pages needed will be too high for our early-triggering strategy to be effective. This fact effectively limits the number of nodes that can participate in an epoch because epoch-computation time is $O(N)$. Measurements presented in Section 7.2.3 show that networks of a few hundred nodes present no problems. In Section 5.2.4, we discussed how to extend this and other aspects of our algorithm to more than a few hundred nodes.

5.3.2 Overestimating Global-Replacement or Conversion Rates

As we have said, the negative effect of underestimating the replacement and conversion rates lead us to favor overestimating these values. Of course, we would like to predict these values precisely and overestimating them does have a negative effect as well. It forces us to tolerate less favorable epochs, because we end up needing to include more pages in the replacement group than is actually necessary.

5.3.3 Idle Node Transitioning to Busy Node

One common and important scenario for global memory management to handle is a node that makes a sudden transition from idle to busy. These transitions occur, for example, when a workstation user returns to her idle workstation to start using it again. If the workstation has been idle long enough, its main memory will consist almost exclusively of global pages and the node will probably tend to be weighted high in the current epoch's weighted distribution. It is important for our algorithm to handle this situation well. The goal is to rapidly move the global pages from the reactivating node to other nodes in the network and to rapidly adjust node weights so that the reactivating node stops receiving forwarded pages (assuming that global pages are completely replaced by the local pages accessed by the nascent user).

The following describes how GMS handles the idle to active node transition. First, as the user starts faulting in local pages, global pages are forwarded to other nodes and the local *victim conversion* count is incremented. Since the node was previously idle, the conversion threshold on the node will be low and thus the node will quickly reach the threshold and trigger a new epoch. As the new epoch is computed, user faulting continues to convert old global pages on the reactivating node into young local pages. When, as part of the new-epoch computation, the coordinator asks the reactivating node for its age information, the reactivating node is likely to still have some old global pages that are candidates for inclusion in M for the next epoch. Notice, however, that at this point, the reactivating node should be experiencing a high conversion rate and so a large number of its global pages will be hidden from the network (as described in Section 5.2.6 above). If the conversion rate is insufficient to hide all of the global pages, some of the reactivating node's pages may be included in M. If this happens, the reactivating node will continue to receive forwarded pages in the next epoch and the node will thus trigger yet another new epoch when its conversion threshold is reached for a second time. Clearly, if the node is in the process of converting all of its global memory to local memory, we would like to stop forwarding pages to the node immediately. It is difficult, however, to detect this situation accurately. The mechanism for doing so is the victim-conversion rate prediction discussed in Section 5.2.7. It is thus important for the predictor to adapt rapidly to an increase in the actual victim-conversion rate. It is for this reason that we consider the rate seen in the last two seconds when predicting the new rate. Experimental results presented in Section 7.4 show that our algorithm performs very well in these circumstances, adapting rapidly to changing idleness.

5.3.4 Few Old Pages in the Network

As the number of old (or idle) pages in the network decreases, we would like our algorithm to gracefully degrade to the point where system performance is comparable to a system with no global memory mechanisms in place. This useful property is a natural consequence of our algorithm. As the number of old pages in the network decreases, the *MinAge* epoch parameter gets younger and younger. As this happens, more and more local replacement victims are dropped instead of forwarded, because only victims *younger* than *MinAge* are forwarded. Finally, when every page in the network is younger than a local-age threshold (five minutes in our prototype), every forwarded page is guaranteed to be older than *MinAge* and is thus dropped (due to the way that local and global are aged as discussed in Section 6.3). Therefore, in a network with only a few old pages, overhead is low because few if any pages are forwarded and because epochs are thus long.

5.3.5 Nodes that Forward Pages Should not Receive Forwarded Pages

During an epoch, each node is selecting local replacement victims from its main memory and forwarding them to other nodes, using the epoch's weighted distribution. It seems possible that a node that is forwarding pages might also be receiving pages from other nodes. In light of the fact that global pages can be accessed from any node with equal cost, such page ping-ponging would be wasteful. Our algorithm uses the *MinAge* constraint and the victim conversion estimate to avoid this waste. If the victim conversion estimate is correct, every local victim chosen by a node with non-zero w_i will be at least as old as *MinAge* and will thus be dropped instead of forwarded.

5.3.6 Node Failures

As with any distributed algorithm, a key issue for our algorithm is how it deals with node and network failures. Several factors about the algorithm might appear to present challenges in this regard. The failure of the coordinator node might delay the initiation of a new epoch, because only that node is keeping track of when M replacements have occurred. Similarly, excessive conversions at a node might not trigger a new epoch because nodes trigger the epoch by sending a message to the coordinator. The failure of a node *during* epoch computation can cause various complications. If a node fails to respond to the coordinator's request for age information, the coordinator could be delayed indefinitely. If a node fails to receive the global information for the next epoch, it might continue to operate in the previous epoch in which a *different* node is the coordinator. Such a node will forward pages using an out-of-date node weighting and will send trigger messages to a node that is no longer the coordinator. A similar problem might occur while the coordinator is broadcasting global age information to all nodes. There will be an interval in which some nodes have received the information and have started acting in the new epoch and other nodes have not yet received it and remain in the old epoch. This time-skew can cause confusion if, for example, a node in the old epoch tries to trigger a new epoch.

The key strategy for avoiding problems caused by node and network failures is to implement a protocol that ensures that all nodes agree on the membership of the *node set* (i.e., the set of nodes that are considered active). A mechanism is needed to check for liveness of all nodes. Message timeouts are used to detect failed nodes and remove them from the node set. Failure of the current epoch's coordinator node is detected in this way, and if the coordinator fails, a new node is elected to fulfill its duties for the remainder of the epoch. Node failures that occur while an epoch is being computed are detected in the same way. If a failure is detected while the coordinator is collecting age information from the rest of the network, the failed node is removed from the node set and initiation continues with the smaller node set. The failure of a node to receive the global age information for the new epoch is detected by the normal liveness-detection mechanism. Note that removing a node from the node set invalidates the node weights stored at each node and thus triggers a new epoch. There are several other issues, not related to the global-information algorithm, that our prototype must deal with to properly handle node failures. Section 6.1.6 discusses these issues in detail.

Finally, time-skew problems are handled by making the official beginning of a new epoch an atomic operation. A new epoch begins when the coordinator node increments the monotonically-increasing *epoch number* associated with each epoch. This number is included with the global information sent to each node and with all other messages associated with our global-information algorithm. Messages from stale epochs are easily detected because any such message is directed to the coordinator for the stale epoch which has a higher epoch number recorded locally.

5.4 Comparison with Other Approaches

In this section we will compare the GMS global-information algorithm with two competing algorithms for managing network memory that have been proposed by others — *N-Chance Forwarding* and *Hints* — and a third algorithm used for workstation-network load balancing, a similar distributed-decision problem — *Random Probe*.

5.4.1 N-Chance Forwarding

N-Chance Forwarding was proposed by Dahlin et al. as an algorithm for *cooperative client caching* in the xFS distributed filesystem [22, 3]; we discussed cooperative caching in Section 2.2.3. Key to cooperative caching is that client-cache managers coordinate their activities with the goal of (1) keeping active-node pages in network memory longer than would be possible with only a smaller server cache (i.e., use remote client caches as an extended server cache) and (2) minimizing the amount of duplication among memories in the network. Both of these goals were also explored by Franklin et al. in their work on client-server object databases [33]. In [22], Dahlin et al. studied several cooperative-cache management algorithms using the simulations of the Sprite filesystem traces [7]. Their simulations showed that N-Chance Forwarding performed best of the alternatives they studied.

Algorithm Description

Each client uses LRU to independently manage its local cache. When a client replaces a *local victim* from its cache, it checks whether the victim is unique or replicated. If unique, the page is forwarded to another node that is chosen *randomly*. The receiving node makes room for the page by discarding its oldest replicated page; if all of its pages are unique, the forwarded page is forwarded again. If retained, the forwarded page is made the youngest page on the node's LRU list and the page's *recirculation count* is set to N (they found that $N = 2$ or 3 worked well). If the page is forwarded again before being accessed, its recirculation count is decremented; pages with a recirculation count of zero are dropped instead of being forwarded.

Discussion

N-Chance is a *type-3* algorithm according to the taxonomy presented in Section 5.1.3 (i.e., blind decisions corrected at the target node). This algorithm has the advantage that global information is unnecessary for deciding where to forward a local victim. The algorithm relies on a series of random choices to find a node that has sufficient idle memory to store the page. In making a random choice of target node, the algorithm sometimes forwards pages to nodes that are actively using their client cache and thus have no room for the forwarded page. In this case, the forwarded page will rapidly descend the node's LRU list until it becomes the oldest page and is forwarded again. If N is sufficiently high, the page will eventually end up on a node that is not actively using all of its cache and will thus remain there for some time.

The key problem with N-Chance Forwarding is that random target selection works well only when most nodes are idle or when old pages age uniformly distributed throughout the network. In the Sprite workloads used to evaluate N-Chance, between 66% and 90% of the nodes were idle in each 10-minute period [7]. As a result, the simulations showed that N-Chance's performance was nearly optimal. We shall see in a moment that Sarkar and Hartman used these same traces to evaluate their *Hints* algorithm. Both studies lack any analysis of how their algorithms would perform on workloads with different characteristics. In particular, blind algorithms such as N-Chance will make frequent mistakes when idleness is rare or skewed.

Unfortunately, the performance cost of mistakes in N-Chance is high (as it is for global memory management in general). Performance is impacted in two ways. First, the blind approach will forward pages to non-idle nodes. In N-Chance, these pages displace valuable local-replicated pages on those nodes, resulting in subsequent faults to retrieve the displaced pages from remote memory when the pages are accessed again. Programs on non-idle nodes are thus slowed by the overheads of receiving forwarded pages, of handling page faults when displaced pages are accessed, and of forwarding to other nodes the pages it mistakenly receives. Second, if a page is forwarded to N non-idle nodes in a row, never finding the idle memory that exists in the system, the page is dropped from the network; the original program will thus have to read the page from disk instead of remote memory even though idle memory existed to hold the page — it was just hard to find.

Local Replicated vs. Global Pages in N-Chance

The first problem with N-Chance, the cost to non-idle nodes, is due in part to the way the algorithm compares the "value" of local-unique, local-replicated, and global pages. Importantly, N-Chance does not distinguish between local and global pages. Forwarded pages are handled the same as local pages on a node. In particular, a forwarded page becomes the youngest page on the recipient node and will thus displace a local replicated page on that node (recall that a forwarded page will not replace a unique page). It is interesting to compare this to the GMS approach. In GMS terms, the forwarded page would be called a *global* page. As such, it would be less valuable than a comparably aged local-replicated page for reasons discussed in Section 4.4. As a consequence, a forwarded page would not displace local pages, as is done by N-Chance, unless those pages are somewhat *older* than the forwarded page itself (where its age is the time it was last accessed). Changing N-Chance to incorporate this philosophy would substantially reduce the cost of mistaken forwardings to non-idle nodes. For such a node, it is unlikely that the forwarded page would be young enough to replace any local replicated page, a key source of the overhead in N-Chance.

5.4.2 Cooperative Caching Using Hints

Sarkar and Hartman have recently proposed a new algorithm for managing cooperative filesystem caches [60]. This algorithm seeks to improve on N-Chance by using some global information to make forwarding decisions, while retaining N-Chance's simplicity and low-cost decision making. Sarkar and Hartman use simulations of the same Sprite workloads used in the N-Chance studies to compare their algorithm to N-Chance and GMS. These simulations show that, for these workloads, the hints algorithm performed as well as GMS with slightly lower overhead and that both perform better than N-Chance on some of the traces.

Algorithm Description

As with N-Chance, each node maintains its client cache independently using LRU. In addition nodes maintain an array with a *hint* of the age of the oldest page on each node. Local replacement victims are forwarded to the node that has the oldest entry in the forwarding node's hint array. The hint arrays of the forwarding and receiving nodes are updated each time a page is forwarded. The forwarding node piggy-backs the age of its new oldest page with the forwarded page. The receiving node acknowledges the forwarding message with a message that includes the current age of its oldest page.

A centralized cache, called the *discard cache*, is also maintained by the system. When a node receives a forwarded page, it replaces its oldest page to make room for the forwarded page. If this displaced page is younger than any entry in the node's hint array, it is forwarded to the discard cache. The discard cache mitigates the effect of replacement errors caused by the use of inaccurate hints. Any displaced page that a node thinks might not be the oldest page in the network is stored in the centralized discard cache, which then affords a low-cost global comparison for discarding the globally oldest of all such pages.

The Sarkar-Hartman algorithm also uses hints for block coherence and location; whereas GMS (and N-Chance) use a distributed-directory approach. These differences are discussed in Sections 6.4 and 6.5, when we present our prototype implementation.

Discussion

Hints has lower overhead than GMS, because it eliminates the computation and distribution of global age information needed for each epoch. For the Sprite traces, at least, Hints results in the same average read time as GMS. Both outperform N-Chance Forwarding for some of the traces because the random forwarding results in poor utilization of idle memory. If this feature holds for other workloads as well, Hints appears to have the additional benefit of simplicity and scalability. Recall that the key impediment to scalability in GMS is the centralized portion of epoch computation. The key question is: do the simulation results presented in [60] generalize to other workloads. It is well known, however, that Unix filesystem accesses are idiosyncratic and not representative of VM or database workloads.

Note first that the Hints algorithm assumes a reply message for each forwarded page. In old-style networks, an acknowledgment was commonly needed to ensure message delivery and so hints could be exchanged without adding any extra messages. Cluster networks and many modern LANs (e.g., Digital AN2), however, provide reliable delivery in hardware and thus eliminate the need for acknowledgment messages. If an acknowledgment is not needed, then the exchange of hints requires an *extra* message for every forwarded page. As a result, the Hints algorithm would generate more total messages than GMS when epochs summarize more then $3 \times N$ pages, as is common. Nevertheless, the important question remains: does Hints provides average read time comparable to GMS for a wide range of important workloads. If it does, it might be preferred over GMS due to its simplicity even if they both have the same overhead.

Hints is similar to GMS in that it uses global information to make forwarding decisions. There are three key problems, however, with the global information that Hints uses (i.e., the age of the oldest page on each node): (1) the hints are *arbitrarily* out of date, (2) each hint is only a page deep, and (3) obsolete information is not invalidated.

The first problem is that a node only updates its knowledge of another node's state when the two nodes exchange a forwarded page. Consider a node, A, that is actively using its local cache and therefore has only young pages. Assume that every other node starts with an accurate hint, which indicates that all of A's pages are young. Other nodes that are actively using their own caches will correctly avoid forwarding pages to A. Over time, the hints for A that these nodes hold will grow stale. Ironically, idle nodes will continue to have an up-to-date hint for A, as long as A is forwarding pages to those nodes. There are two possible ways that the hints algorithm could handle stale hints. First, as hints become more out-of-date, it could make remote nodes appear to have older and older pages. In this case, each node will eventually choose to forward a page to A even though A really has no old pages. The number of these erroneous forwardings that an active node receives is $O(N)$, where N is the number of active nodes in the network, possibly limiting the algorithm's scalability. Second, if out-of-date hints don't make pages appear older, there would be no mechanism for an active node to discover when A becomes idle and its pages actually do start aging. Clearly, when A's pages become among the oldest in the network, other nodes must be able to discover this fact. A third alternative, not suggested by Sarkar and Hartman, is to alert the network of an active-to-inactive transition using a broadcast message sent to every node in the network.

The second problem is that, because the hint contains information for only a single page, the information can only be used to direct a single forwarded page. A subsequent forwarding must wait for the acknowledgment message to be returned from the target node to determine if the target has more old pages or if the rest of its pages are relatively young.

In the first case, the target might be selected again to receive the next forwarded page, while in the second case another node is likely to be chosen. Thus, a round-trip network delay is required between each local replacement on a node. Sarkar and Hartman mention that this problem could be handled by maintaining at each node the age of the oldest X pages, instead of only the single oldest page. It would seem that this would allow a node to make X forwarding decisions before it receives updated information from the the first target node of the forwarding sequence. This is not the case, however, because several other nodes might have information about the same X pages and if they are forwarding pages concurrently with the first node, the target node might end up receiving more than X pages before any of the forwarders receive updated information.

The third problem is that forwarding a page from node A to node B invalidates the hint stored at a third node C, which is not updated by the forwarding (i.e., A and B get updated hints but C does not and thus now has a stale hint for B). At some point in the future, when C uses its out-of-date hint for B, it will send a page to B that replaces a younger page on B than C intended. In general, every time a page is forwarded to B, a hint on some other node is invalidated and the hint at the forwarding node is updated.

When B receives pages from all nodes with equal probability, the overall effect is to keep the hints for B fairly up to date. If, however, B receives a flurry of forwardings from a single node (or small number of nodes), those forwardings will invalidate the hints for B held on a large number of other nodes. This invalidation will later cause a large number of pages to be mistakenly forwarded to B, when the out-of-date hints are used. Similarly, any replacement on B due to local activity will invalidate another node's hint without updating any node. Note that in both of these cases, the number of mistaken forwardings that B will receive following an invalidation burst is bound by the number of nodes in the network, because each node that uses an invalidated hint will receive a new hint in return. When the number of nodes is large, however, the resulting overhead could be significant, once again bringing into question how well the Hints algorithm will scale to a large network.

5.4.3 Workstation Load Balancing Using the Random-Probe Algorithm

A well studied form of the distributed-decision problem we have been discussing is load balancing for workstation networks. The goal of load balancing is to use process migration to evenly distribute network load across all of the machines in the network. Typical balancing algorithms work by choosing to start newly created processes on the least loaded node in the network. Each time a process is created, the creator node must decide which node in the network is least loaded; it then sends the new process to that node. This load balancing decision thus appears quite similar to the distributed-decision problem for global memory management, in which the goal is to find the node with the least valuable page in the network and forward the local replacement victim to that node.

Algorithm Description

Eager et al. have shown that a type-two algorithm (from Section 5.1.3) called *Random Probe* provides the best solution for workstation load balancing [28]. Using their algorithm, each time a process is created, the system decides whether to create it locally or on a lightly-loaded remote node. To make this decision, a *probe* message is sent to a small number of randomly chosen nodes. Each node responds with information about its current load. The requesting node compares the remote loads and its own load, and creates the new process on the node with the lowest load. Eager et al. showed that the random-probe algorithm finds a sufficiently lightly loaded node with far fewer messages than a more coordinated algorithm would require. It is typically sufficient to probe as few as three other nodes; a minimum of seven total messages are thus needed, including one to migrate the new process.

Discussion

A key difference between load balancing and global memory management is the frequency of decision making; that is, page faults occur much more frequently than process creations. Conservatively, an active node might incur a fault rate of 200 faults/s. In this case, a probe-style algorithm results in a per-node message rate of 1,400 messages/s (seven for each fault). The type-one GMS algorithm results in a message rate of 203 messages/s on all but the epoch coordinator and $((200 + 3N)/t)$ messages/s on the coordinator for epochs of t seconds. GMS overhead is thus significantly lower for all but the coordinator node. Even the coordinator's overhead is lower for small networks; for a typically-shortest 5-second epoch, for example, probe uses more message than GMS for networks of 2000 nodes or fewer. Furthermore, the coordinator is typically an idle node and so a higher overhead is quite reasonable, particularly if it reduces overhead for active nodes, as it does in GMS. Of course, there may be ways to modify the probe algorithm to reduce the number of messages needed, by remembering load information from one fault to another. An algorithm of this form, combining features of type-1 and type-2 algorithms, is interesting and will be discussed further in Section 5.5.2.

5.4.4 Summary and Discussion

The goal of the GMS algorithm we have described in this and the preceding chapter is to have each page replacement decision in the network reflect global priorities. The algorithm closely approximates global LRU in a way that is efficient to implement; as we will see in Chapter 7. One of the algorithm's key features is that, because it approximates LRU, it is stable under a variety of workloads and it is thus easy to understand and analyze. The two other global memory management algorithms that have been proposed have taken a different approach. Both N-Change forwarding and Hints are based on heuristics that have been tuned through analysis of the Sprite filesystem traces. They both perform well for these traces and this simulation is offered as the argument for their effectiveness. It is unclear, however, if the heuristics perform in a predictable way for other types of workloads (e.g., VM and databases) or even other filesystem workloads.

In our discussion of the three algorithms in this chapter we have analyzed each without using empirical arguments because we wish to understand their comparative strengths and weaknesses for the widest possible rage of workloads. The key issue of this chapter and the focus of our analysis is the question of how these algorithms make globally informed replacement decisions.

Random algorithms such as N-Chance have the appeal that no global information is needed to make the decisions. A page forwarding consists of the selection of a series of random nodes and a set of *local* replacement decisions at each node. It has the disadvantage, however, that to find a node with idle memory may require several forwardings and each mistaken forwarding to a non-idle node slows programs on that node by replacing local pages with forwarded pages. In other words, the algorithm suffers because random decisions are too blind. In Chapter 7 will will provide empirical data collected from prototype implementations of GMS and N-Chance Forwarding that validates our analysis of these two algorithms.

The Hint algorithm proposed by Sarkar and Hartman performs better than N-Chance Forwarding because it, like GMS, uses global information to pick the node to receive forwarded pages. Unlike GMS, however, Hints piggy-backs global-information messages on the page forwarding and forwarding-acknowledgment messages. The goal is to achieve the benefits of GMS's global decision making with the lowest possible overhead. As we have seen, there are several potential problems. First, forwarding-acknowledgment messages are not needed in many modern cluster and some modern local-area networks. Thus, the message-overhead savings compared to GMS is small or non-existent. Hints removes the need for a centralized coordinator and thus may scale to larger networks than GMS. The scalability of Hints is limited, however, by two factors. First, an active node will not exchange hints with other active nodes. As a result, the hints at those nodes will age to the point where the nodes appear to have old pages, at which point a page will be forwarded to the active node; the number of these erroneous forwardings each active node receives is $O(N)$. Second, a flurry of forwardings to a target node from a small number of nodes will invalidate the hints held at other nodes for the target. These invalidated hints will eventually be used to erroneously forward a page to the target node. Again, the number of erroneous forwarding that nodes receive is on the order of the number of nodes in the network. Finally, unlike GMS, it is difficult to predict how the Hints algorithm will behave on a wide variety of workloads and under various network conditions.

The GMS algorithm was designed to ease analysis of the algorithm. As a result, the global information stored at each node reflects system state in a predictable way. This predictability comes at the cost of a centralized computation that combines information from all nodes in the network. The thesis of Dahlin et al. and Sarkar and Hartman is that this precision, and the resulting overhead, is unnecessary for good performance. Sarkar and Hartman present convincing evidence that their algorithm will work well for Unix-like filesystem workloads but it is unclear, at best, whether either N-Chance or Hints will perform well for other workloads. The argument for a simpler algorithm, however, has merit. In the next section we will briefly outline ways in which the GMS algorithm could be simplified and its overhead lowered.

5.5 Simplifying the GMS Algorithm

There are number of ways that the GMS Algorithm might be changed that would lower its overhead, simplify its operation, or improve its scalability while maintaining its good performance characteristics (i.e., low average read time for applications).

5.5.1 Involve Fewer Nodes in each Epoch

First, we could reduce the number of nodes that participate in each epoch. One way of doing this is suggested by the Hints algorithm. An interesting characteristic of the Hints proposal is that active nodes exchange age information with each other infrequently. In fact, the only information exchange really needed in the system is to inform active nodes about inactive nodes. This fact suggests a possible improvement to the GMS algorithm that would reduce its overhead. Each node could be designated *active*, *inactive*, and *unsure*. The coordinator would collect age information from only inactive and unsure nodes and distribute it to only active and unsure nodes. The number of messages is still $O(N)$ but it might now closer to N than $3 \times N$ (i.e., there would be two messages for each inactive/unsure node and one message for each active/unsure node).

5.5.2 GMS vs Probe

Another approach might be to attempt to dispose of the centralized epoch computation altogether. One way of doing this is to use a *probe* technique in which each node collects state information from a randomly chosen set of other nodes. Note that the Hints algorithm is a bit like this but it collects information only from nodes that exchange forwarded pages. A probe technique could improve on Hints by providing control over how out-of-date global information is allowed to become and thus eliminating the erroneous forwardings caused by stale hints.

A key question is what sort of global information is collected. We saw above that simply collecting the age of the oldest page on a node is unsatisfactory because it is useful for only a single forwarding decision; that is, it is not resilient to being out-of-date. We could instead use the local age arrays computed by each node in the GMS algorithm, or we could alternatively use simpler information such as a quartile ranking of page ages on a node. Each node would cache this information for a random subset of nodes and would update the information using periodic random probes and using information piggy-backed on forwarding (and other) messages as is done in the Hints algorithm. We would tolerate age information being somewhat out-of-date, but each node's information would have a *best-if-used-by* time and stale information would not be used to make forwarding decisions. A node would make forwarding decisions using its locally cached age information by computing a GMS-style node weighting, which it then uses to direct page forwarding as in GMS.

One advantage of this GMS-Probe algorithm compared with GMS-Epoch is that the expiration and collection of information is spread evenly over time. Using epochs, all global information becomes invalid at once. This all-at-once property has three potential disadvantages. First, it might require shorter epochs to achieve the same quality of information as we have with GMS-Probe. Second, we have already discussed several complications of the epoch approach related to detecting early expiration of the epoch. Third, handling the overhead in bursts may be less efficient and be more disruptive to the performance of the system. For example, GMS-Epoch may result in far fewer messages than GMS-Probe but, in GMS-Epoch, all of those messages come at once while GMS-Probe spreads them evenly over time.

The key disadvantage of GMS-Probe is that it requires significantly more messages than GMS-Epoch to collect information for a given number of nodes. That is, information collection is $O(N^2)$ in GMS-Probe, and is $O(N)$ in GMS-Epoch, due to its use of a central coordinator node. Two factors could make this difference less important. First, it might be possible that good forwarding decisions can be made with information from a small subset of nodes. Observe, however, that this sample must at least be representative of the idleness distribution of the network. Second, a probe could exchange information from more than just the two nodes involved. Instead, a probe message could include all of the age information currently cached by the node that sends (or replies to) the probe. Doing so would reduce the number of probes needed to collect information about a particular number of nodes. The total number of bytes exchanged, however, would not be reduced. In GMS-Epoch, nodes receive only a summary of global age information, instead of a list of information from each node as done in GMS-Probe. It would be difficult for GMS-Probe to exchange summary information because, as described, each node has information for a potentially different subset of nodes.

5.6 Summary

In this chapter we described the general problem of making global replacement decisions efficiently, we presented the details of our algorithm for dealing with these issues, we described the strengths and weaknesses of three alternative approaches, and we discussed possible changes to our algorithm.

Our algorithm uses an idle node to collect age information from every node, to combine this information into a global-age summary, and to disseminate the global information back to the nodes. The age summary computed is a weighted distribution in which each node is assigned a weight based on how many of the network's oldest pages are stored on that node. Nodes then probabilistically replace the globally oldest network pages by forwarding locally-replaced pages to nodes according to the global distribution. The process is repeated every five to thirty seconds when the global-age summary becomes out-of-date.

A key advantage of our approach is that it closely approximates global-LRU, which makes the algorithm easy to analyze and ensures its stability for a wide range of workloads. Another important feature of the algorithm is that it minimizes the overhead imposed on active nodes. It does this in two ways. First, active nodes receive global information using a single message exchange with the centralized node; less centralized approaches would require active nodes to send more messages to collect information from a sufficient number of other nodes. Second, our use of global information ensures that pages are not forwarded to active nodes. While our centralized approach has advantages, it limits the scalability of our algorithm to a few hundred nodes. For large networks, the network can be subdivided into groups of nodes that each run an instance of our algorithm. A higher-level algorithm would monitor the global distribution of idle and active nodes and would move nodes between groups to maintain a balance.

We examined three alternative approaches in detail: N-Chance Forwarding, Cooperative Caching using Hints, and Random-Probe for workstation load balancing. N-Chance makes forwarding decisions blindly and relies on the target node to correct its mistakes. The cost of mistakes is high, however, and mistakes are common in networks of scarce or skewed idleness. The Hints algorithm improves on N-Chance by caching limited global information at each node and using this information to choose forwarding targets. These global hints, however, are inferior to GMS in ways that may result in poor performance and more message overhead for certain workloads and network configurations. Finally, the Random-Probe technique for workstation load balancing may have applicability for global memory management. The main problem is that using Probe, active nodes receive information from only one other node in each message exchange, while, with GMS, nodes receive information about the entire network in a single message exchange.

The GMS algorithm has several useful properties and key advantages over alternatives that have been proposed. Of concern, however, is the overhead imposed on the centralized epoch coordinator node. This overhead is low for networks of a few hundred nodes and we choose an idle node to be the coordinator so that no active node suffers from this overhead. Nevertheless, the centralized approach has some disadvantages. We discussed two modifications to our algorithm, one that reduces the message overhead of the coordinator approach and the other that eliminates the coordinator entirely.

We will describe our prototype implementation of GMS in the next Chapter.

Chapter 6

THE GMS PROTOTYPE IMPLEMENTATION

For the things we have to learn before we can do them, we learn by doing them.

— Aristotle

This chapter describes our prototype of the Global Memory System (GMS), which is implemented in the Digital Unix workstation operating system.

The GMS prototype changes the standard Digital Unix memory management system in four main ways. First, GMS manages a dynamically-sized portion of main memory for storing global pages. Second, page faults and filesystem cache misses are handled by checking global memory for the desired page; only pages not found in any workstation's memory are read from disk. Third, pages replaced from local memory are forwarded to remote nodes, where they are stored in the global memory managed by GMS. Fourth, the algorithm described in Chapter 5 maintains global age information, which is used to direct forwarded pages to nodes with the oldest pages in the network.

We begin with a detailed description of the structure of our prototype. Following this is a discussion of the three key implementation issues for global memory management: (1) how to maintain accurate age information for every page in memory, (2) how to determine which local pages are replicated on other nodes, and (3) how to locate pages in global memory.

6.1 Structure of the GMS Subsystem

6.1.1 Integration into Digital Unix

The GMS prototype is tightly integrated with the Digital Unix memory management system. A simplified representation of this system is depicted in Figure 6.1. The boxes represent functional components and the arrows show some of the control relationships. The two key components of the basic Digital Unix memory system are (1) the VM system, which supports *anonymous pages* devoted to process stacks and heaps, and (2) the Unified

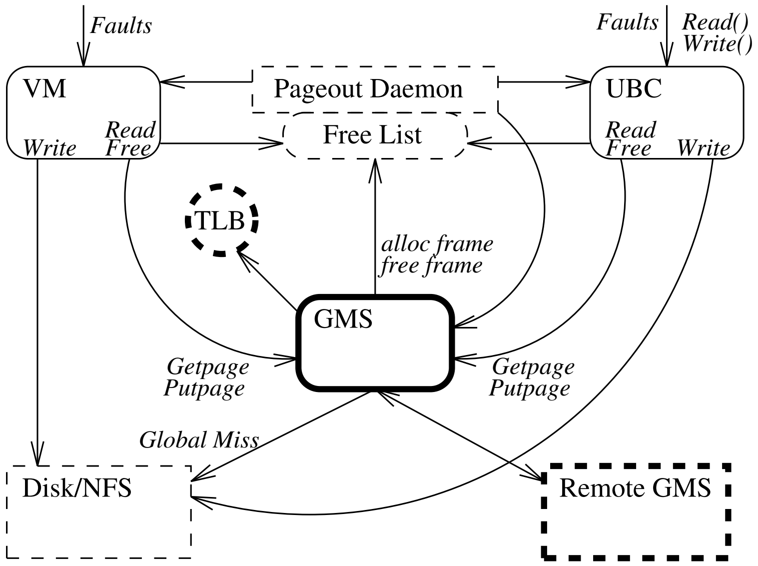


Figure 6.1: Structure of modified Digital Unix memory management system.

Buffer Cache (UBC), which caches file pages. The UBC contains pages from both mapped files and files accessed through normal read/write calls and is dynamically-sized. Page allocation is handled using a *free list* and Pageout Daemon. The VM and UBC subsystems grow by allocating physical pages, as they are needed, from the free list and the Pageout Daemon shrinks the subsystems by reclaiming pages when the free list becomes too small. At the same level as VM and UBC, we have added the GMS module, which holds global pages housed on the node.

We modified the kernel to insert calls to GMS at each point where pages were either added to or removed from the UBC. Similarly, we inserted calls into the VM swapping code to keep track of additions and deletions to the list of anonymous pages. Inserting these calls into the UBC and VM modules allows us to track the collection of pages on the node. It also allows us to forward disk reads to GMS. Disk writes occur exactly as in the unmodified Digital Unix system. The Pageout Daemon was modified to make replacement decisions using the GMS-maintained age information, reclaiming pages from VM, UBC, or GMS, depending on which subsystem holds the node's oldest pages.

6.1.2 Basic Data Structures

Our prototype manages pages in the cluster, which are ultimately backed by secondary storage: either a local disk or an NFS server's disk. Each page of data must be uniquely identified, so that we can track the physical location of that page (or the multiple locations for a shared page). We uniquely identify a page in terms of the file blocks that back it. In Digital Unix, entire pages, rather than disk blocks, are transferred between the disk driver and the rest of the kernel. Thus, to identify the contents of a page, it is sufficient to know the IP address of the node backing that page, the disk partition on that node, the inode number, and the page's offset within the inode. We use a 128-bit cluster-wide unique identifier (*UID*) to record this information. We ensure that the page identified by each UID is in one of four states: (1) cached locally on a single node (*local-unique*), (2) cached locally on multiple nodes (*local-replicated*), (3) cached on a single node on behalf of another node (*global*), or (4) not cached at all.

GMS maintains three principal data structures, keyed by UID.

1. The *page-frame-directory* (PFD) is a *per-node* structure that contains a record for each page (local or global) that is present on the node. A successful UID lookup in the PFD yields information about the physical page frame containing the data, LRU statistics about the frame, and whether the page is local-unique, local-replicated or global. An unsuccessful lookup implies that the particular page is not present on this node.

2. The *global-cache-directory* (GCD) is a *network-wide* data structure that is used to locate the IP address of a node that has a particular page cached. There is one entry in the GCD for every network-memory resident page, listing the nodes that cache the page and whether the page is local or global. For performance reasons, the GCD is organized as a hash table, with each node storing only a portion of the table (i.e., the GCD is distributed).

3. The *page-ownership-directory* (POD) maps the UID for a sharable page to the node storing the GCD entry for that page. For non-sharable pages, the GCD entry is always stored on the node that is using the page. The POD is a small hash table replicated on every node and is changed only when machines are added or deleted from the cluster.

Table 6.1: The GMS-subsystem interface.

Operation
gms_getpage(UID, page_frame, page_offset, byte_count)
gms_putpage(UID, page_frame, group)
gms_discard(UID)
gms_syncpage(UID)
gms_addpage(UID, page_frame)
gms_discarded(UID)
gms_group_discard(UID, group)
gms_pageout(page_count) => (ubc_page_count, vm_page_count)

The *page-ownership-directory* provides an extra level of indirection that enables us to handle the addition or deletion of nodes from the network without changing the GCD hash function. A central server running on one of the workstations is responsible for updating the indirection table and propagating it to the rest of the network. This updating must occur whenever there is a reconfiguration. Parts of the global-cache-directory database are redistributed on reconfiguration as well. However, none of these operations are critical to the correct operation of the network. In the worst case, during the redistribution process, some requests may fail to find pages in global memory and will be forced to access them on disk; if so, the stale global pages will be discarded when the redistribution completes, thus avoiding any coherence problem.

6.1.3 The GMS Interface

GMS exports the interface shown in Table 6.1 to the rest of the operating system. The VM and UBC subsystems retrieve pages from global memory by calling *gms_getpage* and add pages to global memory by calling *gms_putpage*. The subsystems inform GMS of a newly created local page using *gms_addpage* and of a deleted local page using *gms_discarded*. To delete a page from network memory, they call *gms_discard*; *gms_syncpage* is called to delete all but the locally-stored copy of a page and is used to maintain coherence of shared pages. When pages are added to global memory they are tagged with a globally-unique group number; for VM the owning process ID is used and for file pages the file's device and inode number are used. When a process terminates or a file is deleted, the system

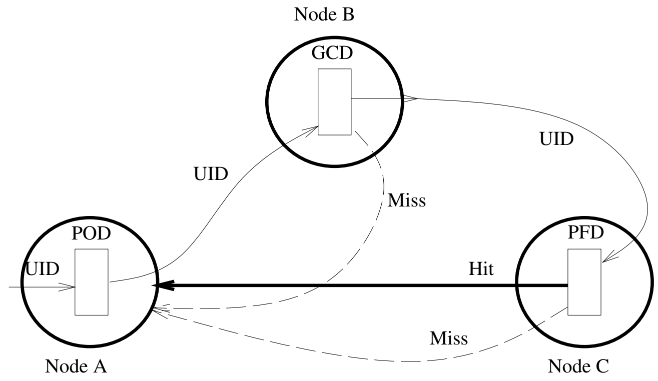


Figure 6.2: Locating a page.

calls *gms_group_discard* for the corresponding group, which deletes all global pages associated with that group. The Pageout Daemon calls *gms_pageout* to request that *page_count* pages be returned to the free list. To respond to this request, GMS examines the page-age information it maintains for all local and global pages and identifies the *page_count* oldest pages on the node. The portion of these pages that are managed by GMS are freed. The number managed by UBC and VM is returned to the Pageout Daemon, which then reclaims the pages by calling the subsystems appropriately.

In the next few sections, we will examine the most important GMS operations in more detail.

6.1.4 Locating a Page in Remote Memory

Following a page fault, before reading the page from disk, the system attempts to find the page in remote memory, by calling *gms_getpage*. GMS locates a page using a two-step lookup, as shown in Figure 6.2. First, the requesting node (e.g., node A in Figure 6.2) uses the UID to hash into the POD, producing an IP address of the node (node B) implementing the appropriate region of the global-cache-directory. The requester sends a message containing the UID to the GMS on that node, requesting the page identified by the UID. The GMS on the node does a lookup in the global-cache-directory, which results in either a miss or a hit. In case of a miss, a message is sent to the original node which then reads the page from disk (or from the NFS server). In case of a hit, the GCD node forwards the request to the node that contains the page-frame-directory for the UID (node C), which sends the page to the requesting node. Figure 6.2 shows three nodes involved in the lookup.

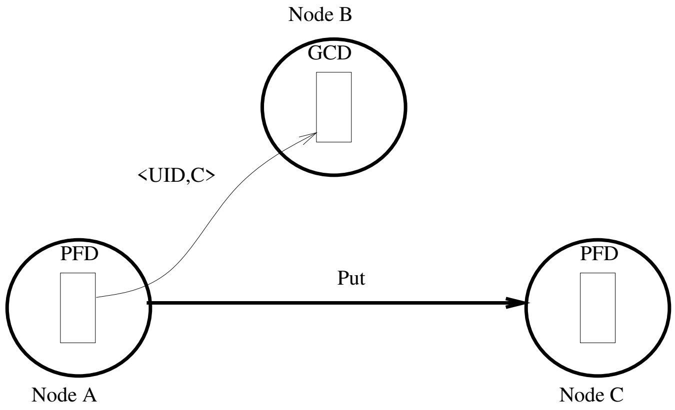


Figure 6.3: Executing a putpage operation.

When a faulted page is not shared, nodes A and B are identical, thereby decreasing the number of network messages needed. This distribution of the page management information is somewhat similar to the handling of distributed locks in VAXclusters [44].

As a side-effect of a successful lookup, data structures on each of the three nodes are updated to reflect the changed state of the system. The global-cache-directory is updated with the new location of the page and the page-frame-directories on the requester and the remote node are updated suitably. If the remote node was caching the page on behalf of another node (i.e., it was a global page), then the remote node deletes its entry from the page-frame-directory. (Recall from Section 4.1 that there needs to be only one copy of a global page in the system, because the cost of remotely accessing a global page is independent of which node it is on.) The only other possibility is that the requested page is a local page, in which case the remote node marks it as a local-replicated page. In either case, the requesting node adds an entry.

6.1.5 Sending a Page into Remote Memory

Pages are sent into remote memory by calls to *gms_putpage*. When a node performs a putpage operation, apart from sending the page to the target node, it also updates the global-cache-directory at the node that is responsible for the affected page. The target node and the sending node each update their page-frame-directory structures. If the evicted page is a local-replicated page, the sending node simply discards the page. If the page is a local-unique or global page, the sending node deletes the page and the target node marks the page as *global*.

The target node for a putpage operation is selected by executing the algorithm described in Section 5.2. Figure 6.3 depicts the major steps involved in a putpage operation.

6.1.6 Addition and Deletion of Nodes

When a node is added to the network, it checks in with a designated master node, which then notifies all the existing members of the new addition. The master node distributes new copies of the page-ownership-directory to each node, including the newly added one. Each node distributes the appropriate portions of the global-cache-directory to the new node. In the current implementation, the master node is pre-determined and represents a single point of failure that can prevent addition of new nodes. It is straightforward to extend our implementation to deal with master node failure through an election process to select a new master, as is done in other systems [44, 62].

Node deletions are straightforward as well. The master node checks periodically for the liveness of the other nodes. When it detects a crashed node, it redistributes the page-ownership-directory. As with addition, global-cache-directories are also redistributed at that time.

6.1.7 Inter-node Communication

Between nodes we use simple non-blocking communication. In our current implementation, we assume that the network is reliable and we marshal and unmarshal to and from IP datagrams directly. This is justified primarily by the increased reliability of modern local area networks such as AN2 that have flow control to eliminate cell loss due to congestion [4]. To date, we have not noticed a dropped packet in any of our experiments. However, our implementation can be readily changed to use any other message-passing package or transport protocol.

6.2 Basic Operation of the GMS Global Management Algorithm

Section 4.1 described the basic operation of the algorithm in terms of a series of page swaps between nodes (see Cases 1–4 in Section 4.1). However, in the actual implementation, the swaps are performed in a somewhat different fashion.

On a fault, the faulting node allocates a frame from its free list and executes a remote *getpage* operation for the missing page. The inter-node interaction for a getpage operation is described in Section 6.1.4 and shown in Figure 6.2. The getpage operation represents one half of the swap; the second half is executed at a later time in the context of the Pageout Daemon. As getpage operations are executed on a node, the free list shrinks, eventually causing the Pageout Daemon to wakeup. The Pageout Daemon, in concert with the GMS, the UBC, and the VM managers (see Figure 6.1), attempts to evict the oldest pages on the node. These evictions are handled by the GMS module, which has age information about all these pages. Some of the pages are pushed to other idle nodes (using a *putpage* operation); others are discarded because they are older than the *MinAge* value for the current epoch. The putpage operations that go to other nodes may eventually cause older pages on those nodes to be forwarded or discarded. The putpage operation is described in Section 6.1.5 and shown in Figure 6.3.

6.3 Collecting and Using Age Information

An important part of our replacement algorithm is the periodic collection of *age arrays* from every node, as described in Section 5.2.3. These arrays are collected every five to thirty seconds and are aggregated to compute a set of global weights. The global weights are then used by active nodes to send replaced pages to nodes with the network's globally oldest pages. The age arrays contains information for every page — local, global, and free — on a node.

Unfortunately, it is difficult to track the ages of some pages in most workstation systems such as Digital Unix. For pages belonging to files accessed via explicit read/write requests, these calls can provide the required age information. Similarly, the age of global pages is recorded when the page is placed in global memory. Access to *mapped* pages (i.e., VM and mapped-file pages), however, is invisible to the operating system. As a result, workstations typically use replacement algorithms that provide little useful page information about these pages on nodes that are not heavily faulting — exactly the case in which we wish to know the ages of pages.

We have implemented two alternative techniques for collecting age information about mapped pages. In the first, we maintain a last-access-time array for each physical frame using a modified TLB-miss handler and periodic flushes of the TLB. In the second, we maintain age statistics using a modification to the standard Digital-Unix replacement algorithm that ages pages even during low-faulting periods. We discuss both alternatives below and compare their performance in Section 7.2. Before we do, however, it is important to understand precisely what type of age information we require and how it is used.

6.3.1 Assembling a Node's Age Array

Age information is needed by our global-management algorithm in order to assemble the *age arrays* on each node at the beginning of every epoch. These arrays summarize the age of the pages on a node, indicating the number of pages in each age category from 0 to 63 minutes old, in one minute increments.

Nodes hold four types of pages, local-unique, local-replicated, global, and free, which each have a different replacement priority. In Section 4.4, we concluded that we should favor replacing global pages over local pages and replicated pages over unique pages. This goal is achieved in our prototype by weighting the age of each page by its type, so that the aggregate age array reflects the desired priorities. Before describing how we do this, we will first discuss the handling of free pages, which were not described earlier, because they are an artifact of our prototype implementation and not central to the algorithm.

Accounting for "Free" Pages

At first, one might be tempted to consider pages currently on the *free list* to be *infinitely* old. Free pages, it may seem, should be favored over other page types, because storage for forwarded pages can be allocated from free memory without replacing other pages (as long as the free list is sufficiently large). This observation, however, misses a crucial point: the size of the free list is not an indication of *idleness* of its pages. To see why, notice that when a program terminates, it returns all of its VM pages to the free list. If free pages were treated as being infinitely old, such a node would suddenly appear to have a large number of old pages and would thus start receiving forwarded pages. Active nodes, in fact, typically start and stop programs frequently, and thus the free list typically fluctuates between a large and small number of pages. Free pages should thus appear old only when they are likely to remain free for some time. We therefore maintain ages for free pages just as we do for the other page types, favoring replacing free pages over local pages and slightly favoring replacing global pages over free pages.

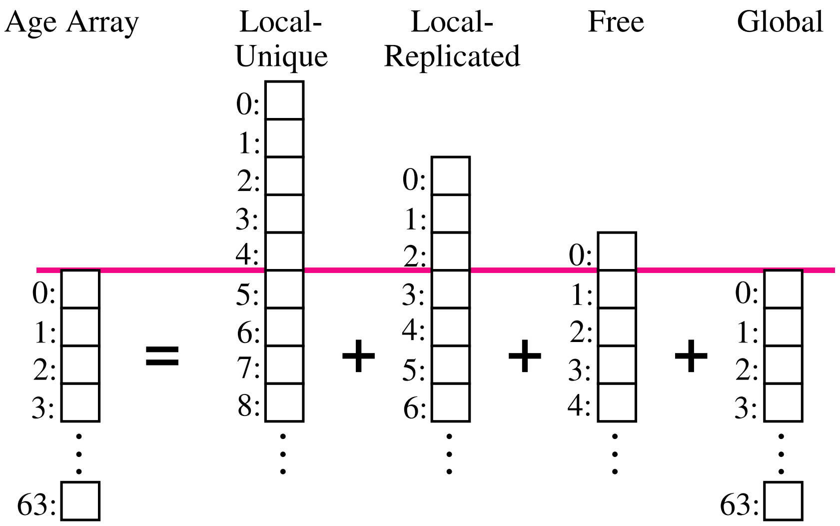


Figure 6.4: Consolidation of page ages according to type to form the *age array*.

Consolidating Page Types into the Age Array

We now describe how we consolidate the four different types of pages to form the age array, in a way that reflects the desired replacement priority. As shown in Figure 6.4, each node actually maintains four separate age arrays, one for each type of page. The aggregate age array is formed as the normalized sum of these four individual arrays, by adjusting the age of pages from different groups according to desired replacement priority (i.e., favor replacing global pages, then free, local-replicated, and finally local-unique pages). The diagram shows the normalization scheme used by our prototype: it adds four minutes to the age of each free page, five minutes to the age of each global page, and two minutes to the age of each local-replicated page. The age array includes only pages whose normalized age is older than five minutes, because forwarded pages, which are always *global* pages, will replace only pages older than themselves (i.e., older than five minutes).

Notice that our choice of normalizing scheme is only one of many that are possible. To be more aggressive about replacing local pages with global pages, for example, global pages could be weighted by fewer than five minutes; local performance could be further protected by increasing the global-page weight to ten or twenty minutes.

6.3.2 Key Characteristics of Age Information

Two characteristics of the *age array* are critical to the design of an algorithm for collecting age information for *mapped* local pages.

1. Age information is summarized with 1-minute granularity and young local pages are not reported.

2. Age information is *collected* more frequently (sometimes every 5 seconds) and the transition of a page from old to young must be reflected immediately.

These two characteristics will guide the design of our two alternative age-collection algorithms described in the next two sections. The first characteristic suggests that we do not need precise age information for local pages, particularly those that are young. The second characteristic indicates, however, that the algorithm must react quickly when an old page is accessed or replaced. This quick reaction is necessary in order to properly trigger the end of an epoch (as discussed in Section 5.2.6) and to ensure that a node does not report to the epoch coordinator old pages that it no longer holds. In summary, pages slowly get older until they are either accessed or replaced; an age-collection algorithm can be somewhat lazy about reporting page aging, but must be aggressive about reporting the sudden end of this aging process.

We will now discuss two alternative approaches for collecting local age information. We have implemented both in our prototype system and report on their performance in Section 7.2.

6.3.3 Alternative 1: Using TLB Misses to Determine Access Times

In order to collect age information mapped pages, we modified the Digital Alpha TLB handler, which is implemented in PALcode [67]. Once a minute, we flush the TLB of all entries.[1] Subsequently, when the TLB handler performs a virtual-to-physical translation on a TLB miss, it sets a bit for that physical frame. A kernel thread samples the per-frame bit once a minute in order to maintain LRU statistics for all physical page frames. The

---

[1] The Alpha does not identify which TLB entry is *replaced* when a new translation is added; it is thus impossible to know which pages are currently in the TLB without periodically clearing it. If the Alpha did provide this support, we could record a page's age when it was removed from the TLB and thus avoid the once-per-minute TLB flush.

node's age arrays are computed from the physical-frame LRU statistics. At the beginning of each epoch, the algorithm reads the LRU information and page type for each physical page frame, interprets this information to index the appropriate age-array element, and increments the element.

This algorithm is well suited to the key local-information characteristics outlined above. First, as much as possible, collection overhead is deferred to the once-per-minute scan of the physical frame table; the TLB-miss handler performs the minimal work of indexing the physical-frame table and setting a single reference bit for the accessed page. Second, an access to an old page is recorded immediately, because it causes a TLB miss, which updates the LRU statistics used to build the age array.

Unfortunately, there is a tension between the goal of providing immediate notice of access to an old page and the necessity of minimizing the impact our algorithm has on TLB-miss overhead. This tension results in a fairly high cost for building the *age array* at the beginning of each epoch. This overhead is a function of the size of physical memory; while per-page overhead is small, less than 0.3 μs per page on a 225-Mhz Digital Alpha (see Section 7.2), the total cost is high on systems with large memories, almost 10 ms for a 250-Mbyte memory and 8K pages. This high per-epoch overhead lead us to consider an alternative design, described next.

6.3.4 Alternative 2: Using a "Time-Driven" Clock to Determine Access Times

The goal of this second alternative is to reduce the overhead of building the *age array*, required from every node at the beginning of each epoch. This reduction is achieved by shifting work from the once-per-epoch task of building the array to the on-going task of collecting age information. The four page-type age arrays are updated directly as pages are allocated, mapped, accessed, replaced, and freed. As a result, the per-node epoch computation overhead is minimized; the age array is computed simply by adding the four pre-computed age arrays together, instead of by scanning the physical-frame table, as was necessary in the previous algorithm. This approach, however, increases collection overhead to the point where it is no longer feasible to use the TLB-miss handler to collect age information for mapped pages; low TLB-miss overhead is critical to good system performance. Instead, we devised a new algorithm, called *Time-Driven Clock*, that maintains age information for all pages at all times, even for mapped pages during low-faulting periods. Before describing our algorithm, we review its basis, the standard

Clock-like algorithm found in most systems, including Digital Unix.

Standard Two-Handed Clock and FIFO-With-Second-Chance

Two-Handed Clock [6, 79] maintains a single bit of information for each page, the *referenced* bit, set by the system when a page is accessed. Physical pages are logically organized into a circular list that is swept by two “hands” as replacement victims are needed. The first hand clears the reference bit for pages it passes over. The second hand checks the reference bit of pages it passes and replaces any page whose reference bit is still clear. That is, any page that is not referenced between the time the first and second hands pass over it is deemed to be *old* and is therefore replaced.

FIFO-with-Second-Chance [77, 26] is a common variant of Two-Handed Clock, implemented by most systems including Digital Unix. The system maintains three FIFO queues of pages: active, inactive, and free. Only active pages are directly accessible to applications. The system enforces a minimum size for each list by moving pages to the head of a list from the tail of the preceding list. Faulted pages are allocated from the free list and pages are moved from the inactive list to the free list and from the active list to the inactive list. Pages on the inactive list remain mapped by the programs that own them but the TLB entries for those pages are modified to cause an operating-system trap when they are accessed. Such an access sets the page’s *reference bit* and moves the page to the active list.

For clarity, we will describe our algorithm as a modification to Two-Handed Clock, although, in fact, it is implemented as a modification to Digital Unix’s FIFO-with-Second-Chance replacement algorithm.

Using Clock to Maintain the Age Array

A simple modification to Clock maintains the desired age information when clearing page reference bits. Each time a reference bit is cleared, the algorithm updates the page’s last-access time, decrementing the age bucket that counts its previous age and incrementing the bucket for its new age; updates also occur when file pages are accessed by read/write filesystem operations and when pages are returned to the free list. Using this procedure, the algorithm maintains a good lower bound on the age of every page except mapped pages whose reference bit is currently set; access to these pages is invisible to the operating system.

The lack of age information for pages whose reference bit is currently set is the key problem with using Clock for global memory management, because reference bits are only cleared when page faults advance the clock hands. This strategy makes sense for local memory management, because a node's old pages are needed only to make room for newly faulted pages on that node. For global memory management, however, a page fault *anywhere* in the network requires age information from *every node* in order to replace the globally oldest page. LaRowe and Ellis proposed a solution for NUMA multiprocessors in which the Clock algorithms running on every node are synchronized; a fault on any node advances the clock on every node [41]. We take a similar approach: instead of using a globally synchronized algorithm, however, each node advances its clock independently and clock hands advance even when nodes are not faulting.

Time-Driven Clock

Time-Driven Clock is similar to standard Two-Handed Clock except that the algorithm advances clock hands even on idle nodes. The key challenge for the algorithm is to detect idle pages as aggressively as possible without imposing significant overhead on active nodes. If the algorithm is too aggressive about advancing the clock, the number of reference bit clearings (i.e., *deactivations*) and settings (i.e., *reactivations*) on the node will increase, perhaps dramatically. Reactivations are particularly costly because they turn normal memory accesses into operating-system traps to set page reference bits, to update the age array, and to reset TLB entries to permit further access. The Time-Driven Clock algorithm addresses this challenge by slowly advancing the clock hands of a non-faulting node, while monitoring the node's reactivation rate; the clock hands are gradually accelerated as long as the reactivation rate remains below an acceptable threshold.

During non-faulting periods, the time-driven clock is advanced by D page deactivations per minute; the value of D is set according to the following process. Once a minute, a kernel thread checks the number of reactivations, R, that occurred during the preceding minute. If the number of reactivations is less than T_{low}, D is doubled; if it is between T_{low} and T_{high}, D is instead increased by only $T_{high} - R$. If, however, the number of reactivations is greater than T_{high}, D is reduced by $R - T_{high}$. Each time the standard pageout thread runs, at least four times a second in Digital Unix, it checks the reactivation rate of the preceding five-second interval. If the number of reactivations exceeds T_{burst} in any such interval, D is set to zero, stopping the time-driven advance of the page-aging clock until the next time the kernel thread runs.

The key features of this algorithm are the following. We slowly increase the deactivation rate as long as reactivation overhead remains suitable low; the deactivation rate is increased more rapidly when overhead is very low. If the reactivation overhead ever bursts too high, we stop deactivating pages immediately. The deactivation rate is increased at most once a minute in order to allow sufficient time for the effects of the current rate to be reflected in the reactivation count before attempting to increase the deactivation rate again. Increasing the deactivation rate aggressively when reactivation overhead is very small allows the algorithm to rapidly find idle memory when it exists, while limiting overhead when most of the remaining pages are non-idle.

As reported in Section 7.2, the deactivation and subsequent reactivation of a page adds a total of 14.4 μs of overhead, in our prototype. As a consequence, our prototype uses the following settings of the algorithm's threshold parameters: $T_{high} = 400$ limits average overhead to 0.01%; $T_{low} = 40$ aggressively increases deactivations when average reactivation overhead is below 0.001%; and $T_{burst} = 70$ limits peek overhead to 0.1%.

Summary

In this section we saw that the standard approaches for handling local replacement, Two-Handed Clock and FIFO-with-Second-Chance, fail to meet the page-age needs of global memory management. In particular, global management differs from local management in that age information must be collected from every node, even from nodes that are not themselves faulting. This information is needed because a fault anywhere in the system triggers a replacement decision that must consider the age of all network pages in order to replace the globally oldest page among them.

Section 7.2 reports on the performance of our Time-Driven Clock algorithm (and our TLB-based approach as well). Our experiments show that using Time-Driven Clock, we are able to rapidly deactivate idle pages, even on non-faulting nodes, without imposing significant overhead due to this aggressive page deactivation. The algorithm directly updates the age array when pages are deactivated and it ages pages once a minute by simply shifting page counts within the age array. This approach results in minimal collection overhead. Importantly, per-epoch overhead is trivially small because the age array is maintained on the fly as age information is collected.

6.4 Maintaining Information About What Pages are Replicated

An important feature of the locally-stored page-frame-directory (PFD) is that it records the state of every cached page, including whether the page is unique or replicated. Having local access to this information is important for two reasons. First, as discussed in Section 4.4, the replacement algorithm should favor replacing replicated pages over unique pages. Second, when a replicated page is replaced, it should be discarded from local memory without forwarding it to another node. There is, however, a potential disadvantage to maintaining replication information locally in the PFD. Doing so requires an extra message from GCD to PFD when a page transitions from replicated to unique; all other PFD state changes are handled as a by-product of other GMS operations and thus require no additional messages.

Li and Hudak [50] studied a similar problem for distributed shared memory systems, in which replication information is needed by the system's coherence protocol. That problem differs from ours, however, because coherence protocols are naturally distributed, and thus having *local* access to replication information was not an issue.

We will now consider four alternative approaches that can potentially reduce the number of extra messages needed for the maintenance of replication information.

6.4.1 Alternative 1: No Local State

First, state information could be eliminated from the PFD, thus making the extra replica-to-unique transition message unnecessary. This choice, however, would require the algorithm to treat unique and replicated pages equally when choosing a replacement victim and would thus result in the unnecessary forwarding of replicated pages to remote memory. The alternative would be to perform a synchronous message exchange with the GCD for every replacement decision, resulting in significantly more messages than the GMS approach.

6.4.2 Alternative 2: Cached Local State with Bulk Updates

Second, half way between the full-state approach taken in GMS and the no-state approach just mentioned is a compromise in which nodes cache replication information for their oldest pages and periodically inquire of the GCD to update this cached information. The xFS filesystem uses this approach [3]. The advantage of this scheme over GMS is that it replaces individual PFD-update messages, needed when pages transition from replicated to unique, with messages that provide bulk information for groups of pages. The disadvantage is that these bulk messages are large and, unless they are frequent, the cached replication information risks being out-of-date.

6.4.3 Alternatives 3 and 4: Master-Slave

The final two alternative solutions are based on a *master-slave* heuristic. The first copy of a page is designated the *master* and is assumed to be unique. Subsequent, *slave*, copies are assumed to be replicated and are thus discarded when replaced. The problem with this approach, however, is that the master copy is always assumed to be unique even when it is actually replicated. This fact complicates replacement, making it difficult to favor replacing replicated pages, as we would like. Furthermore, when a master copy is replaced, the system can either (1) contact the GCD to find out if the page is in fact unique, significantly increasing message overhead; or (2) obliviously forward the page to remote memory as if it were unique, forwarding replicated pages unnecessarily. We now look at both variants.

Master-Slave with Validated Master

In the first master-slave scheme, the system determines if the master is really unique before making its forwarding decision. The replacement of a master page thus requires either two or three additional messages, compared to the one extra message needed by GMS. A synchronous exchange of two messages is needed to determine the page's true status and a third message is needed when the master is replicated in order to upgrade one of the remaining slaves to be the new master.

In most cases, this master-slave approach results in significantly more messages than the GMS approach. When the number of copies of a page transitions from two to one and then from one to zero, GMS requires only one message, while master-slave requires five if the master is replaced first and two if the slave is replaced first. In addition, it is somewhat more likely that the master will be replaced first because, other things being equal, the first copy accessed is likely to be the first copy replaced. The only case in which master-slave performs better, is when the number of copies oscillates between one and two; each transition from two to one requires a message in GMS while no messages are required by master-slave.

Master-Slave with Oblivious Master

In the second master-slave scheme, the master copy is replaced as if it were unique, even if it is not, thus avoiding the message overhead of the previous scheme. Sarkar and Hartman proposed this approach for filesystem cooperative-cache management [60]. The master copy is always forwarded to global memory (or to a remote cooperative cache) and slave copies are never forwarded. This approach has the advantage that it requires no additional messages to maintain replication information. This variant at first appears to provide an advantage when pages oscillate between two and one copies, the worst case situation for GMS. Notice, however, that when pages do oscillate, the master copy is replaced and forwarded to global memory roughly half the time. When this happens, overhead is significantly increased and memory is used inefficiently, because global memory is allocated for a page when a copy still exists in local memory elsewhere. The one case in which this master-slave variant will perform better than GMS is when the number of copies goes from two to one to zero (but not two to one to two). In this case, both master-slave and GMS forward the master copy into global memory. Master-slave forwards the master when transitioning either from two to one or from one to zero copies. GMS forwards the master when transitioning from one to zero copies. Master-slave, however, avoids the extra message GMS requires when the number of copies goes from two to one.

6.4.4 Summary

In summary, there are clear advantages to maintaining replication information locally; doing so allows us to favor replacing replicated pages over unique pages and replicated pages can be discarded with no additional message overhead. The choice of technique for maintaining replication information depends on network and workload characteristics. If it is significantly more efficient to send a few large messages than many small messages, the local-cache-bulk-update method may be preferred. If it is common for the number of copies of a page to oscillate between two and one, the first master-slave variant may be preferred. If it is common for the number of copies to fluctuate between two and zero, the second master-slave variant may be preferred. Both master-slave approaches significantly increase overhead for other access patterns, however: the first when the copy-count transitions from two to zero, requiring multiple messages to get the master's true status and to nominate a new master; and the second when the copy-count goes from two to one and then back to two, sending replicated masters into global memory unnecessarily. The advantage of the

GMS approach is that it works well for a broad range of access patterns, requiring a single small message only when the number of copies transitions from two to one and ensuring that pages are only forwarded into global memory when absolutely necessary.

6.5 Alternative Techniques for Finding Sharable Pages in Remote Memory

We saw in Section 6.1.4 that GMS locates pages using the global-cache-directory (GCD), a distributed directory that records the location of every memory-resident page in the network. This section reviews the motivation behind our page location strategy and examines two alternatives.

The GCD stores two types of pages: *non-sharable*, such as VM pages and pages from local files, and *sharable*, such as pages from NFS files. The two types of pages are located in different ways. Finding non-sharable pages is considerable easier than finding sharable pages, because the current location of non-sharable pages can be stored locally on the node that accesses those pages; in GMS, non-sharable pages are recorded in a locally allocated region of the GCD. Sharable pages, however, can be accessed from any node; we therefore use a globally-accessible directory to find these pages (i.e., the global portion of the GCD). A global directory such as the GCD has the potential disadvantage that finding a sharable page requires three network messages, instead of the two needed when location information is stored locally. In addition, directory-update messages are required when pages change locations.

A problem similar is studied in [50] for distributed shared memory systems, in which pages are located either using a centralized directory, by broadcasting location messages to every node in the network, or by following forwarding pointers to locate the node storing a page's location information. The GCD is similar to centralized approach, except that the GCD is *logically* centralized and physically distributed. The other two approaches are not practical for our purposes.

We now examine two alternative techniques that can be used to reduce page-location message overhead. The first is to manage the directory entries for a page on the node most likely to access those pages. The second is to cache location information on the nodes most likely to access that information. We will examine both techniques in the remainder of this section.

6.5.1 Co-Locating Directory Management and Page Access

The key issue for maintaining a global page location directory is how a node determines the node that stores the directory entry for a page. In GMS, we use the page-ownership-directory for this purpose, dividing the UID namespace for sharable pages evenly, but arbitrarily, among all the nodes in the network. If a page tends to be accessed with higher probability from a particular node, however, location messages can be reduced or eliminated by storing the page's directory entry on that node. A mechanism must then be provided for nodes to determine the identity of a page's designated directory node.

It would be simple, for example, to modify the GCD to manage all of a file's pages on one of the nodes that currently has the file opened. When a file is opened on only one or a small number of nodes, this optimization would substantially reduce the number of location messages. Other nodes could easily learn the identity of a file's GCD node when they open the file. Files that are opened by a large number of nodes could revert to the current POD-based system in order to evenly distribute location overhead instead of burdening a single node with all of the requests for the file. The GCD node of a page or file can be changed at any time as long as all nodes agree on the identity of the new GCD node. In fact, a mechanism for adjusting GCD responsibilities, by redistributing the POD, already exists in the GMS prototype in order to handle the addition and deletion of nodes.

6.5.2 Local Caching of Location Information

A second alternative is to cache location information for certain pages locally on one or more nodes so that accesses on those nodes can locate the pages with no remote communication. The key issues are to decide what to cache and how to update caches when pages move.

Sarkar and Hartman [60] suggest a hint-based variant for filesystem cooperative caching. When a node opens a file it receives a set of hints for blocks of the file listing nodes that might be caching them. These hints are retrieved from the last node to have opened the file. Recall that they designate the first cached copy of a block to be its master. Nodes that cache master blocks record them in their local hints for the block's file. When a node replaces a master block, it updates the block's location in its local hints. When a client faults on a block, the client checks its hints and, if the block is present, requests the block from the hinted node. Invalid hints are deleted when an attempt to use them fails.

The advantage of this approach is that no messages are required to maintain location information. The key disadvantage, however, is that nodes may be unaware of many cached file pages and will thus read them from disk instead of from remote memory. The problem is that a node is only aware of remote pages that were accessed by another node while that node was the last-opener of the file. For example, if node A and then B open the same file in that order, node A will be unaware of any of B's pages and node B will only be aware of pages accessed by A before B opened the file. As Sarkar and Hartman show in their simulations, this approach actually works well for Unix-like file access represented by the Sprite traces, because these files tend to be small and accessed in their entirety; that is, in terms of our example, A would be finished with the file before B opened it and, because A read the entire file, B would be able to read every page from remote memory using the hints supplied by A. It is unlikely however, that this algorithm will perform at all well for workloads with different access patterns, because many of a file's pages that are cached in memory will be unknown to other nodes and thus nodes will unnecessarily read those pages from disk.

6.6 Summary

This chapter described our GMS prototype implemented as an extension to Digital Unix running on DEC Alpha workstations. GMS is fully integrated with the Digital Unix memory management system, acting as a third memory subsystem along-side the VM system and the UBC file cache. At the core of the implementation are two data structures: the page-frame-directory (PFD) is a per-node data structure that records key information for pages housed on a node; and the global-cache-directory (GCD) is a distributed data structure that records location information for every page currently resident in network memory. Directory entries for non-sharable pages, such as VM and local file pages, are stored in the GCD of the node that accesses them. The entries for sharable pages are stored on the page's designated owner node, which is found by hashing in the page-ownership-directory (POD). The POD is small table replicated on every node, which provides a level of indirection so that GCD entries can move from node to node as nodes come and go from the network.

Key to our global management algorithm is its ability to collect age information for every page in the network. Unfortunately, techniques currently used for local memory management (e.g., Two-Handed Clock), do not provide useful age information for mapped pages (i.e., VM and mapped-file pages) on nodes that are not heavily faulting; exactly the case where we most need this information. The reason for this apparent deficiency is that local management only needs to locate old pages when page faults on the node require the replacement of a page; if the node is not faulting, no page aging needs to occur. A significant difference with global memory management is that it requires age information from *every* node when a fault occurs on *any* node. We described two alternative technique we have implemented for collecting age information for mapped pages on non-faulting nodes. The first uses a modified TLB-miss handler and periodic TLB flushes to record page-access information. The second, the *Time-Driven Clock* algorithm, modifies the standard Clock replacement algorithm to age pages even when a node is not faulting. This time-driven aging starts slowly and accelerates as long as it does not induce significant reactivation overhead. Time-Driven Clock has the advantage that the per-epoch overhead of building the node's age array is trivially small.

This chapter also explored alternative techniques for locating sharable pages and for recording replication information. In both cases, the GMS approaches sometimes require moderately more messages for certain access patterns but their behavior is significantly more stable across a broad range of workloads than are the alternatives.

The next chapter reports on the performance of the GMS prototype.

Chapter 7

PERFORMANCE RESULTS

O mighty Caesar! dost thou lie so low?
Are all thy conquests, glories, spoils,
Skrunk to this little measure?

— *William Shakespeare, Julius Caesar, 1599*

This section provides performance measurements of our implementation of global memory management from several perspectives. First, we present microbenchmarks that give a detailed breakdown of the time for fundamental global memory management operations. Second, we look at the speedup obtained by memory-intensive applications when using our system. Third, we examine the effectiveness of our algorithm under different global loading parameters. All experiments were carried out using DEC Alpha workstations running Digital Unix V3.2, connected by a 155 Mb/s DEC AN2 ATM network. The page size on the Alpha and the unit of transfer for our measurements is 8 Kbytes.

7.1 Microbenchmarks

Here we report simple measurements to evaluate the underlying performance of our current implementation. The experiments were done using eight 225-MHz DEC 3000-700 workstations in the environment above, each with a local disk and connection to one or more NFS file servers that export shared files, including program binaries. All of these measurements were taken on otherwise idle machines.

Table 7.1 itemizes the cost of a getpage operation. The getpage cost depends on whether the page is sharable or non-sharable and whether there is a hit or miss in the global cache. As described in Section 6.1, if the page is non-sharable, the GCD (global-cache-directory) node is the same as the requesting node.

The rows provide the following information. *Request Generation* is the time for the requesting node to generate a getpage request, including access to the page-ownership-directory to contact the GCD node. *Reply Receipt* is the requester time spent to process the

Table 7.1: Performance of the *Getpage* operation (μs).

Operation	**Latency in μs**			
	Non-Sharable Page		**Sharable Page**	
	Miss	Hit	Miss	Hit
Request Generation	7	61	65	65
Reply Receipt	-	156	5	150
GCD Processing	8	8	59	61
Network HW&SW	-	1135	211	1241
Target Processing	-	80	-	81
Total	15	1440	340	1558

reply from the remote node; e.g., for a hit, this includes the cost of copying data from the network buffer into a free page and freeing the network buffer. *GCD Processing* accounts for the total time spent in the GCD lookup operation as well as forwarding the request to the PFD node. *Network HW&SW* is the total time spent in the network hardware and the operating system's network protocol. The bulk of this time is spent in hardware; when sending large packets, the total latency introduced by the sending and receiving controllers is comparable to the transmission time on the fiber. *Target Processing* refers to the time on the remote node to do a lookup in the page-frame-directory and reply to the requesting node. *Total* is the sum of the rows and represents the latency seen by the initiating node before the getpage operation completes.[1] To put these numbers in perspective, on identical hardware, the cost of a simple user-to-user UDP packet exchange for requesting and receiving an 8 Kbyte page is about 1640 μs.

The *Total* line for the first column shows the cost of an unsuccessful attempt to locate a non-sharable page in the cluster. In this case, the requested page is not in the global cache and a disk access is required. This cost thus represents the overhead we add to an Digital-Unix disk access. The Digital-Unix disk access time varies between 3600 μs and 14300 μs, so this represents an overhead of only 0.4–0.1% on a miss.

The getpage operation is also summarized in Figure 7.1, which shows a detailed timeline

[1] We could reduce the getpage hit latency 200 μs by eliminating a copy and short circuiting the network-packet delivery path, but have not yet included these changes.

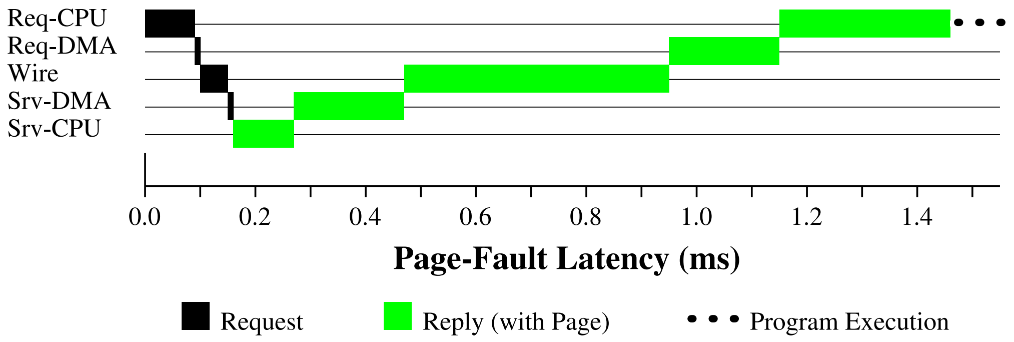


Figure 7.1: Remote page fetch timeline for fault on requesting for a page stored on a serving node.

Table 7.2: Performance of the *Putpage* operation (μs).

Operation	Latency in μs: Non-Sharable Page	Latency in μs: Sharable Page
Request Generation	58	102
GCD Processing	7	12
Network HW&SW	989	989
Target Processing	178	181
Sender Latency	65	102

for the processing of a remote-memory page fault (in this case for a non-sharable page). Times for the request message are shaded darkly and times for the reply message, which includes the requested page, are shaded lightly. *CPU* and *DMA* overheads are shown for the two nodes involved: the faulting node (*Req*) and the storing node (*Srv*) (*DMA* is the transfer between host memory and the network interface). *Wire*, the time the message spends in the network, is also shown.

Table 7.2 shows the costs for a putpage operation. As shown in Figure 6.3, a putpage updates the global-cache-directory as well as the page-frame-directory structures on the source and target nodes. In the case of sharable pages, the sending node might need to initiate two network transmissions—one to the GCD and another to the PFD; this time is reflected in *Request Generation*. In a putpage operation, the sending node does not wait for a reply from the target before returning from a putpage call; therefore, in the bottom row we show *Sender Latency* rather than the total. For sharable pages, this latency is the

Table 7.3: Average access times for non-sharable pages (ms).

Access Type	**Access Latency in ms**	
	GMS	**No GMS**
Sequential Access	1.9	3.6
Random Access	1.9	14.3

same as *Request Generation*; for non-sharable pages, the latency also includes the *GCD processing*, because the GCD is on the same node.

Putpage requests are typically executed by a node under memory pressure to free frames. An important metric is therefore the latency for a frame to be freed; this consists of the sum of the *Sender Latency* and the time for the network controller to transmit a buffer, so that the operating system can add it to the free list. For 8 Kbyte pages, the combined transmission and operating system overhead is about 300 μs.

Table 7.3 compares the average data-read time seen by programs for non-sharable pages with and without GMS. For this experiment, we ran a synthetic program on a machine with 64 Mbytes of memory. The program repeatedly accesses a large number of anonymous (i.e., non-sharable) pages, in excess of the total physical memory. In steady state for this experiment, every access requires a putpage to free a page and a getpage to fetch the faulted page. The average read time thus reflects the overhead of both operations.

The first row of Table 7.3 shows the average performance of sequential reads to non-sharable pages. The numbers shown with no GMS reflect the average disk access time; the difference between the sequential and random access times indicates the substantial benefit Digital Unix gains from prefetching and clustering disk blocks for sequential reads. Nevertheless, using GMS reduces the average sequential read time by 41% for non-sharable pages. For non-sequential accesses, GMS shows a nearly a 7-fold performance improvement. Here the native Digital-Unix system is unable to exploit clustering to amortize the cost of disk seeks and rotational delays.

Table 7.4 shows the data-access times for NFS files that can be potentially sharable. In this experiment, a client machine with 64 Mbytes of memory tries to access a large NFS file that will not fit into its main memory, although there is sufficient memory in the cluster to hold all of its pages. There are four cases to consider.

In the first case, shown in the first column of Table 7.4, we assume that all NFS pages

Table 7.4: Average access times for sharable pages (ms).

Access Type	Access Latency in ms			
	GMS Unique	GMS Replicated	NFS Miss	NFS Hit
Sequential Access	1.9	1.5	4.8	1.9
Random Access	1.9	1.5	16.7	1.9

accessed by the client will be put into global memory. This happens in practice when a *single* NFS client accesses the file from a server. For the most part, the behavior of the system is similar to the experiment described above: there will be a putpage and a getpage for each access. In this case, the pages will be fetched from global memory on idle machines.

The second case is a variation of the first, where two clients are accessing the same NFS file. One client has ample memory to store the entire file while the other client does not. Because of the memory pressure, the second client will do a series of putpage and getpage operations. The putpage operations in this case are for sharable pages, for which copies already exist in the file buffer cache of the other client (i.e., they are *replicated*). Such a putpage operation causes the page to be dropped; there is no network transmission. The average access cost in this case is therefore the cost of a getpage.

The next two cases examine the cost of a read access when there is no GMS. In the first case, we constrain the NFS file server so that it does not have enough buffer cache for the entire file. A client read access will thus result in an NFS request to the server, which will require a disk access. In the final case, the NFS server has enough memory so that it can satisfy client requests without accessing the disk. Here, the cost of an access is simply the overhead of the NFS call and reply between the client and the server. Notice that an NFS server-cache requires the same time as a GMS hit for a *unique* page. This reflects the additional cost of the putpage operation performed by GMS when a page is discarded from the client cache. In NFS, discarded pages are dropped as they are in GMS for *replicated* pages, in which case GMS is 0.4 ms faster than NFS.

7.2 Bookkeeping Overheads

This section describes the cost of performing the essential GMS bookkeeping operations, which include the overheads of maintaining age statistics for local pages and of collecting and propagating global page age information. We show the costs for both local-age algorithms we implemented: *TLB-based*, and *Time-Driven Clock*; these algorithms are described in Section 6.3.

7.2.1 Using the TLB to Maintain Local Age Information

On the 225-MHz processor, our modified TLB handler introduces a latency of about 60 cycles (an additional 18 cycles over the standard handler) on the TLB fill path. In addition, since TLB entries are flushed every minute, with a 44-entry TLB, we introduce a negligible overhead of 2640 (60×44) cycles per minute. In practice, we have seen no slowdown in the execution time of programs with the modified TLB handler.

7.2.2 Using Time-Driven Clock to Maintain Local Age Information

The key overhead of the Time-Driven Clock algorithm (described in Section 6.3.4) is the overhead imposed on an active node by the aggressive deactivation of pages. In order to detect the presence of idle memory on a node that is incurring no page faults, our algorithm deactivates pages at a rate determined by the observed reactivation rate. The rate of deactivations increases as long as the reactivation rate remains below a certain threshold. The correct value of this threshold is determined by the overhead of an erroneous deactivation, that is the sum of the costs of deactivating and reactivating a page. On the Alpha, deactivation cost is 4.5 μs. The overhead for accessing a deactivated page is 9.9 μs; this includes the time for (1) the TLB-fault and resulting operating system trap, (2) the update of the page's age and reference information, (3) the reactivation of the page by updating its page-table entry and loading a new translation for the page in the TLB, and (4) the resumption of the accessing program.

7.2.3 Collecting and Propagating Global Age Information

Collecting and propagating the age information consists of multiple steps: (1) the coordinator triggers a new epoch by sending out a request to each node asking for summary

Table 7.5: Normalized overhead for age information for 2-second epoch (N is this number of nodes, L the number of local pages on a node, and G the number of global pages on a node).

	Operation	**CPU Cost** μ**s**	**Network Traffic bytes/s**
(1)	Coordinator Request	$78 \times N$	$10 \times N$
(2a)	Gather Summary (TLB)	$0.29 \times L + 0.54 \times G$	$62 \times N$
(2b)	Gather Summary (Clock)	19.8	$62 \times N$
(3)	Distribute Weights	$45 \times N$	$(43 + 0.8 \times N) \times n$

age information, (2) each node gathers the summary information and returns it to the coordinator, and (3) the coordinator receives the information, calculates weights and epoch parameters, and distributes the data back to each node.

The three rows of Table 7.5 represent the CPU cost and the network traffic induced by each of these operations. For steps one and three, the table shows the CPU overhead on the coordinator node and the network traffic it generates as a function of the number of nodes, N. Two lines are listed for step two: the first, (2a), when the TLB-based algorithm is used; and the second, (2b), when Time-Driven Clock is used. The CPU cost in (2a) is a function of the number of pages each node must scan: 0.29 μs per local page and 0.54 μs for each global page scanned; thus the overhead on a node with 64-Mbytes (8192 pages) of local memory and 2000 global pages, for example, is 3.5 ms. Step (2b) shows a significantly lower overhead than (2a) because summary information is maintained on the fly using this approach. We display network traffic as a rate in bytes per second by assuming a worst-case triggering interval of 5 seconds (a 5-second epoch would be extremely short). Given this short epoch length and a 100-node network, CPU overhead is less than 0.3% on the coordinator node and less than 0.08% on other nodes using the TLB-based algorithm and substantially less with Time-Driven Clock, while the impact on network bandwidth is minimal.

7.3 Execution Time Improvement

This section examines the performance gains seen by several applications with the global memory system. These applications are memory and file I/O intensive, so under normal circumstances, performance suffers due to disk accesses if the machine has insufficient memory for application needs. In these situations we would expect global memory management to improve performance, assuming that enough idle memory exists in the network. The applications we measured were the following:

Boeing CAD is a simulation of a CAD application used in the design of Boeing aircraft, based on a set of page-level access traces gathered at Boeing. During a four-hour period, eight engineers performed various operations accessing a shared 500-Mbyte database. We simulated this activity by replaying one of these traces.

VLSI Router is a VLSI routing program developed at DEC WRL for microprocessor layout. The program is memory intensive and can cause significant paging activity on small-memory machines.

Compile and Link is a partial compile and link of the Digital-Unix kernel. By far, the most time is spent in file I/O for compiler and linker access to temporary, source, and object files.

OO7 is an object-oriented database benchmark that builds a parts-assembly database in virtual memory and then performs several traversals of this database [13]. The benchmark is designed to synthesize the characteristics of MCAD design data and has been used to evaluate the performance of commercial and research object databases.

Render is a graphics rendering program that displays a computer-generated scene from a pre-computed 178-Mbyte database [16]. In our experiment, we measured the elapsed time for a sequence of operations that move the viewpoint progressively closer to the scene without changing the viewpoint angle.

Web Query Server is a server that handles queries against the full text of Digital's internal World-Wide-Web pages (and some popular external Web pages). We measured its performance for processing a script containing 150 typical user queries.

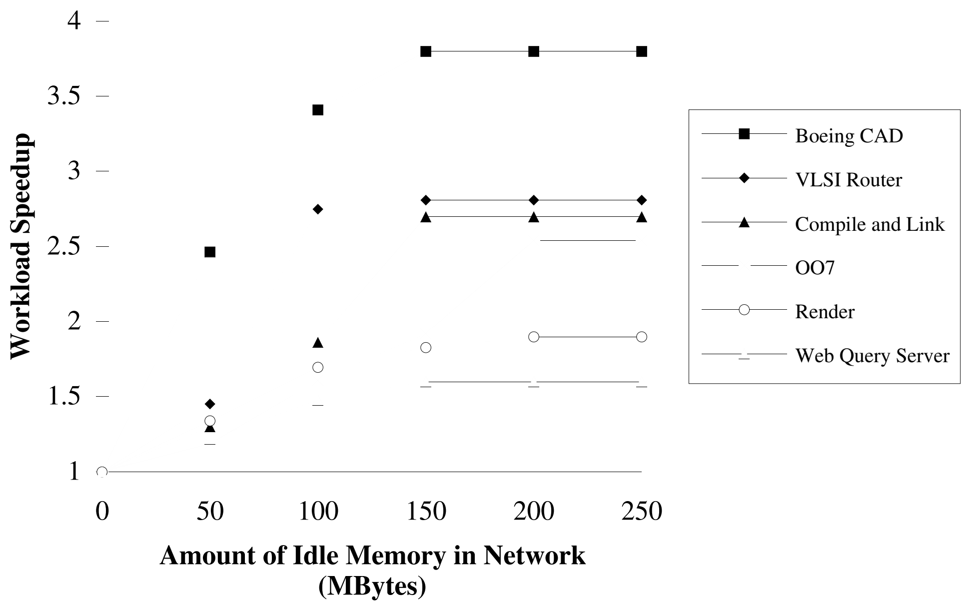


Figure 7.2: Workload speedup with GMS.

To provide a best-case estimate of the performance impact of global memory management, we measured the speedup of our applications relative to a native Digital-Unix system. Nine nodes were used for these measurements: eight 225-MHz DEC 3000 Model 700 machines rated at 163 SPECint92 and one 233-MHz DEC AlphaStation 400 4/233 rated at 157 SPECint92. The AlphaStation had 64 Mbytes of memory and ran each application in turn. The other eight machines housed an amount of idle memory that was equally divided among them. We varied the total amount of idle network memory to see the impact of free memory size. Figure 7.2 shows the speedup of each of the applications as a function of the amount of idle network memory.

As Figure 7.2 shows, global memory management has a beneficial impact on all the applications. With zero idle memory, application performance with and without GMS is comparable. This is in agreement with our microbenchmarks that indicate GMS overheads are only 0.4–0.1% when there is no idle memory. Even when idle memory is insufficient to meet the application's demands, our system provides relatively good performance. Beyond about 200 Mbytes of free memory in the cluster, the performance of these applications does not show any appreciable change, but at that point, we see speedups of from 1.5 to 3.5, depending on the application. These speedups are significant and demonstrate the potential of using remote memory to reduce the disk bottleneck.

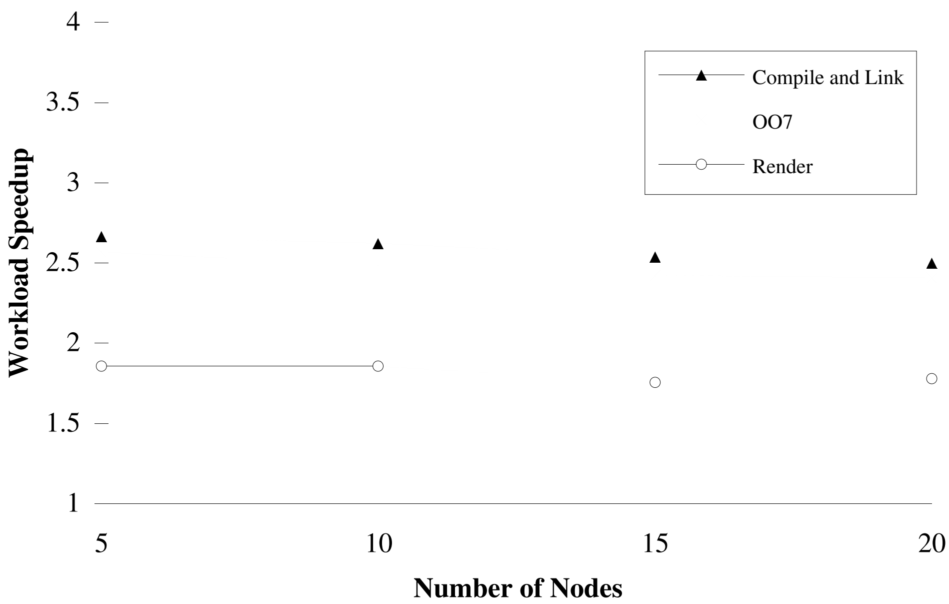


Figure 7.3: Workload speedup as we vary the number of nodes. Two fifths of the nodes are idle and the remainder run a mix of three workloads.

Figure 7.2 shows the speedup of each application when running alone with sufficient global memory. To demonstrate that those benefits remain when multiple applications run simultaneously, competing for memory in a larger network, we ran another experiment. Here we varied the number of nodes from five to twenty; in each group of five workstations, two were idle and each of the remaining three ran a different workload (OO7, Compile and Link, or Render). The idle machines had sufficient memory to meet the needs of the workloads. Thus, when running with twenty nodes, eight were idle and each of the three workloads was running on four different nodes. The results of this experiment, shown in Figure 7.3, demonstrate that the speedup remains nearly constant as the number of nodes is increased.

7.4 Responsiveness to Load Changes

Our algorithm is based on the distribution and use of memory load information. An obvious question, then, is the extent to which our system can cope with rapid changes in the distribution of idle pages in the network. To measure this, we ran a controlled experiment, again using nine nodes. In this case, the 233-MHz AlphaStation ran a memory-intensive application (we chose OO7 for this and several other tests, because its relatively short

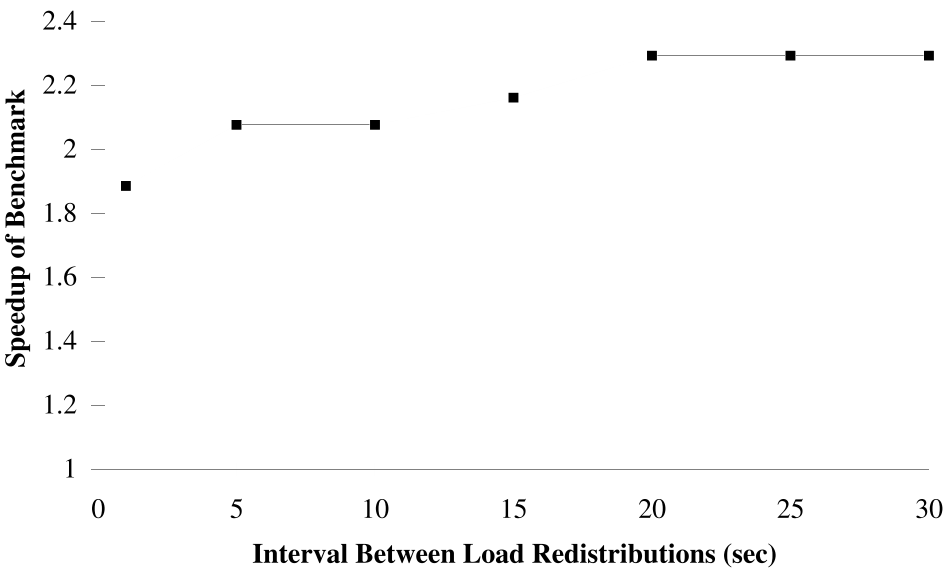


Figure 7.4: Effect of varying the frequency with which nodes change from idle to non-idle on the performance of the OO7 benchmark.

running time made the evaluation more practical); the other eight nodes were divided into two sets: those with idle memory and those without idle memory. The total idle memory in the network was fixed at 150% of what is needed by OO7 for optimal performance, and 4 of the 8 nodes had most of the idle memory. For the experiment, every X seconds we caused an idle node to switch with a non-idle node.

Figure 7.4 shows the speedup of the OO7 benchmark (again, relative to a non-GMS system) as a function of the frequency with which the load changes occurred. As the graph indicates, our system is successful at achieving speedup even in the face of frequent and rapid changes. At a maximum change rate of 1 second, we still see a speedup of 1.9 over the non-global system, despite the fact that the change requires a swap of 70MB of data between the non-idle and idle nodes, which adds significant stress to the system. At lower change rates of 20 to 30 seconds, we see only a small (4%) impact on the speedup.

7.5 The Effect of Idle Memory Distribution

Another key question about our algorithm is the extent to which it is affected by the distribution of idle pages in the network. To measure this impact, we constructed an experiment to vary the number of nodes in which idle pages are distributed. Again, we measured the speedup of the OO7 benchmark on one node, while we controlled the distribution of idle pages on 8 other nodes. We chose three points in the spectrum, arranging in each case that $X\%$ of the nodes contained $(100 - X)\%$ of the idle memory. The cases we consider are (1) 25% of the nodes had 75% of the free memory, (2) 37.5% of the nodes housed 62.5% of the free memory, and (3) 50% of the nodes held 50% of the free memory. Case (1) is the most skewed, in that most of the free memory is housed on a small number of nodes, while in case (3) the idle memory is uniformly distributed.

For comparison, we show results for our algorithm and for N-chance forwarding, the best algorithm defined by Dahlin et al.[23]. It is interesting to compare with N-chance, because it differs from our algorithm in significant ways, as described in Section 5.4.1. To the best of our knowledge, there is no existing implementation of the N-chance algorithm described by Dahlin et al. [23]. Consequently, we implemented N-chance in Digital Unix. We made a few minor modifications to the original algorithm. Our implementation is as follows.

Unique pages are forwarded by nodes with an initial recirculation count of N = 2. When a unique page arrives at a node, the victim page to be evicted is picked in the following order. If there are sufficient pages on the free list such that allocating one will not trigger page reclamation (by the pageout daemon), we allocate a free page for the forwarded page and no page is evicted. Otherwise, we choose in turn: the oldest replicated page in the UBC, the oldest recirculating page, or a very old unique. If this process locates a victim, it is discarded and replaced with the forwarded page. If no victim is available, the recirculation count of the forwarded page is decremented and it is either forwarded or, if the recirculation count has reached zero, dropped.

The modifications improve the original algorithm while preserving its key features: (1) a random choice of a target node is made without using global knowledge, and (2) unique pages are kept in the network at the expense of replicated pages, even if they might be needed in the near future.

Figure 7.5 shows the effect of varying idleness in the network. Here, for each point of the idleness-distribution graph we show four bars—three for N-chance and one for GMS. The captions on the N-chance bars indicate, for each case, the total amount of idle memory that was distributed in the network. For example, for Idle = Needed, the network contained exactly the number of idle pages needed by the application, while for Idle = 2 $\times$ Needed the network contained twice the idle pages needed by the application. The GMS bar is

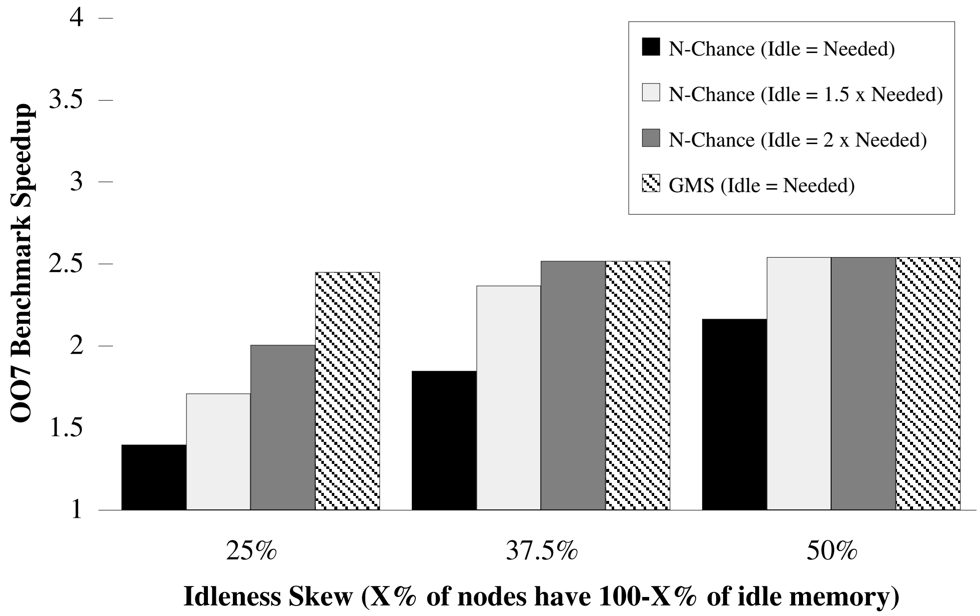


Figure 7.5: Effect of varying distribution of idleness on performance of OO7 benchmark.

measured with Idle = Needed. From the figure, we see that for degrees of idleness that are more skewed, it is difficult for N-chance to find the least loaded nodes. For example, in the case of 25% of the nodes holding 75% of the pages, GMS achieves a substantial speedup relative to N-chance in all cases, even if the N-chance algorithm is given *twice* the idle memory in the network. As we move along the graph to 37.5%, where 3 of the 8 nodes hold 63% of the idle pages, N-chance is equivalent to GMS only when given twice the idle memory. Obviously, when there is plenty of free memory in the cluster, the exact distribution of idle pages becomes secondary and the performance of N-chance is comparable to that of our algorithm, as long as it has more pages than are needed.

The effectiveness of the GMS system and its load information is in fact shown by two things: first, it is superior to N-chance with non-uniform idleness distributions, even given fewer idle pages; second, and perhaps more important, the performance of GMS, as show by the graph, is more-or-less *independent* of the skew. In other words, GMS is successful at finding and using the idle pages in the network, even though it had no excess idle pages to rely on.

The differences between these algorithms are mostly due to the random nature of the N-chance algorithm. Since N-chance chooses a target node at random, it works best if idle memory is uniformly distributed, which will not always be the case.

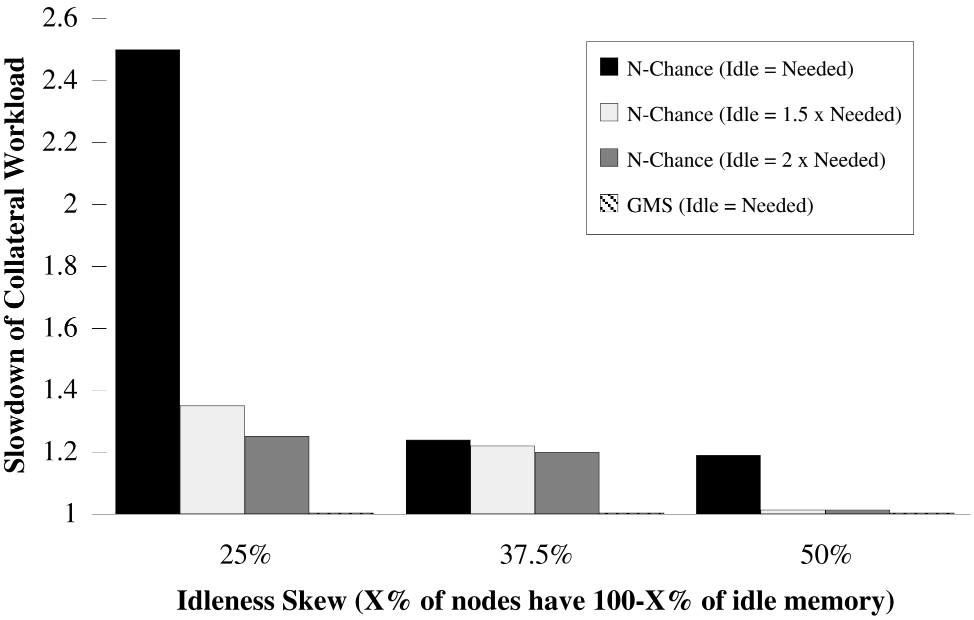


Figure 7.6: Effect of varying distribution of idleness on the performance of a program actively accessing local memory, half of which consists of shared data that is replicated on other nodes.

7.6 Interference with Non-Idle Nodes

A possible negative effect of global memory management, mentioned above, is that the global management may negatively impact applications running on non-idle nodes in the cluster. A heavily paging node can interfere with other non-idle nodes by adding management overhead, by replacing needed pages, or by causing network traffic.

As one test of this effect, we ran an experiment as before with 9 nodes. One node ran the OO7 benchmark, generating global memory traffic. On the remaining 8 nodes, we distributed idle pages as in the previous section, creating different skews of the idle pages. However, in this case, all the *non-idle* nodes ran a copy of a synthetic program that loops through various pages of its local memory; half of the pages accessed by this program are shared among the various running instances, while half are private to each instance.

Figure 7.6 shows the average slowdown of the synthetic programs for both GMS and the three N-chance tests given various distributions of idle memory, as before. The slowdown gives the performance decrease relative to the average performance when OO7 was *not* running. For all the distributions tested, GMS causes virtually no slowdown of the synthetic

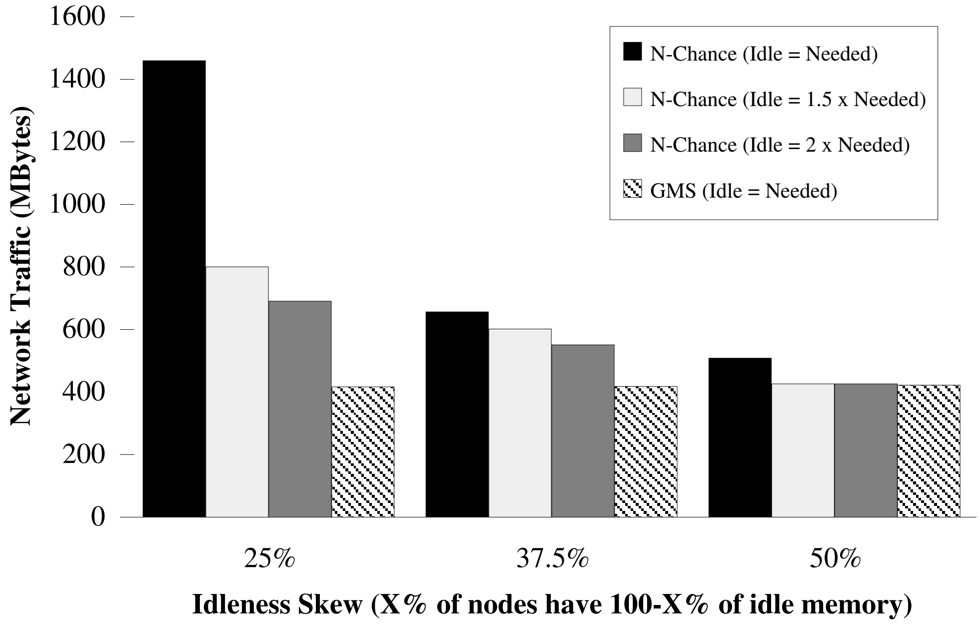


Figure 7.7: Effect of varying distribution of idleness on network activity.

program when OO7 is generating global memory traffic. In contrast, N-chance causes a slowdown of as much as 2.5, depending on the amount of idle memory and its distribution. For 3 idle nodes (37.5%), N-chance causes a slowdown of 1.2, even when twice the needed idle memory is available. When the idle memory is uniformly distributed, N-chance again does as well as GMS, as long as there is additional idle memory in the cluster.

Figure 7.7 shows, for the same experiment, the network traffic in megabytes measured during the test, which took 100–200 seconds. Here we see the network impact of the two algorithms for the various idle page distributions. For the more skewed distribution of 25%, GMS generates less than 1/3 of the network traffic of N-chance, given the same number of idle pages in the network. At twice the number of idle pages, N-chance still generates over 50% more traffic than GMS. Not until the idle memory is uniformly distributed does the N-chance network traffic equal the GMS traffic.

The reasons for these differences are two-fold. First, N-chance chooses nodes at random, as previously described, so remote pages from OO7 are sent to the non-idle nodes. Second, N-chance attempts to keep unique pages in the network by displacing replicated pages, even if they have been recently referenced. This second effect is evident in this test; that is, the active, shared, local pages on the non-idle nodes are displaced by remote pages sent (randomly) to those nodes by the N-chance algorithm. When the program on the non-idle

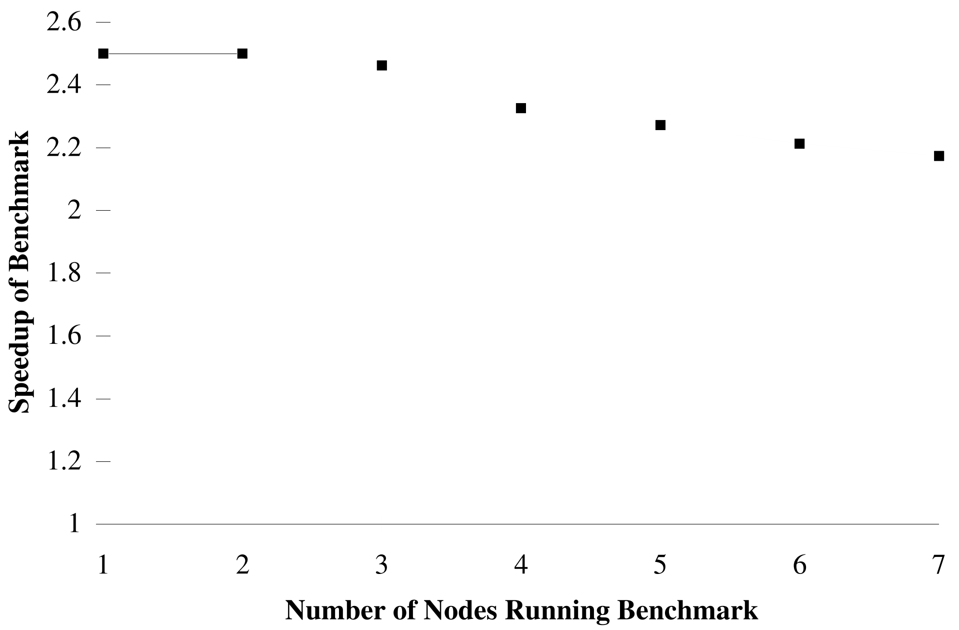


Figure 7.8: Performance impact of a single idle node serving the remote memory needs for multiple client nodes.

node references one of these displaced pages, a fault occurs and the page is retrieved from one of the other nodes or from disk. To make room for the faulted page, the node must discard its LRU page by forwarding it or dropping it if it is a replicated page. This additional paging increases network traffic and reduces the performance of all programs running in the cluster.

7.7 CPU Load on Idle Nodes

The experiment in the previous section demonstrates the success of GMS in avoiding nodes without idle memory. However, even nodes with substantial amounts of idle memory may be "non-idle," i.e., may be running programs executing in a small subset of local memory. A global management system such as ours does place CPU overhead on such a node in order to use its idle memory, through the getpage and putpage requests that it must respond to.

To evaluate the extent of this impact under heavy load, we performed an experiment in which only one node had idle memory. We then monitored the CPU load on that node as we increased the number of clients using its memory for their global storage. Again, we

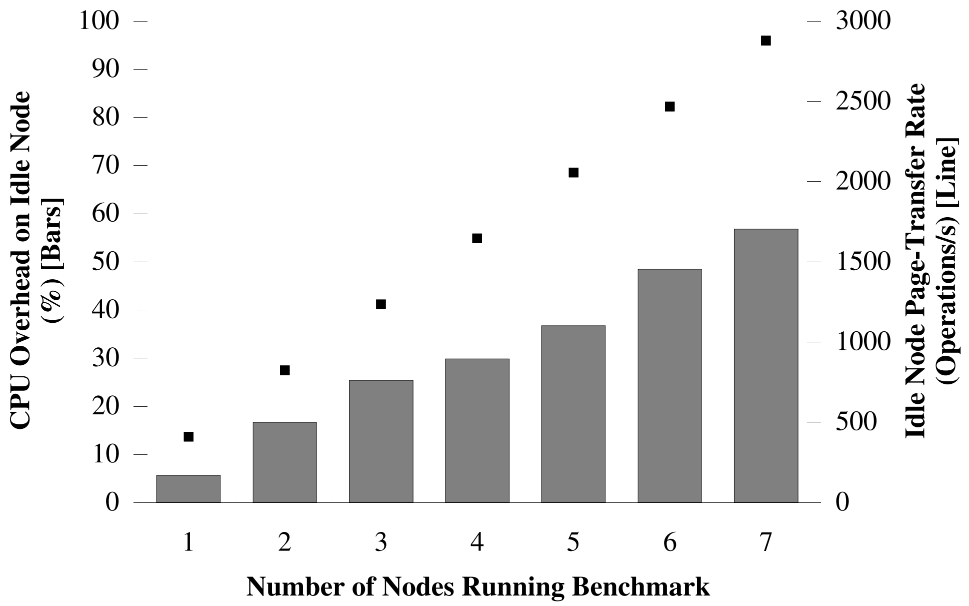


Figure 7.9: Impact of multiple clients on CPU performance of an idle node.

used OO7, increasing the number of OO7 client nodes from one to seven, ensuring in all cases that the idle node had sufficient memory to handle all of their needs.

Figure 7.8 shows that when seven copies of OO7 were simultaneously using the remote memory of the idle node, the average speedup achieved by GMS was only moderately lowered. That is, the applications using the idle node's memory did not seriously degrade in performance as a result of their sharing a single global memory provider.

On the other hand, Figure 7.9 shows the result of that workload on the idle node itself. The bar graph shows the CPU overhead experienced by the idle node as a percentage of total CPU cycles. As well, Figure 7.9 plots the rate of page-transfer (getpage and putpage) operations at the idle node during that execution. From this data, we see that when seven nodes were running OO7 simultaneously, the idle node received an average of 2880 page-transfer operations per second, which required 56% of the processor's CPU cycles. This translates to an average per-operation overhead of 194 μs, consistent with our micro-benchmark measurements.

7.8 Summary

This chapter presented performance results measured from our GMS prototype. A page fault from ATM-connected remote memory takes 1.5 ms (1.9 ms when the fault requires replacement of a local-unique page); this compares with 14.3 ms for a fault from disk (3.6 ms for sequential disk access). Memory- and IO-intensive programs we measured see speedups between 1.5 and 3.5.

We reported on several experiments that show the performance of our system under various stressful workload conditions. We showed that application speedup remains constant as the number of nodes (both actively paging and idle) in the network is increased. We demonstrated that our algorithm is responsive to rapid changes in node idleness; when nodes change from idle to active at a rate of several times per minute, the speedup remains within 10% of the maximum we measured. We also demonstrated that our algorithm does well in networks with low or skewed idleness. We compared GMS to our implementation of N-Chance Forwarding and showed that the blind choices made by N-Chance result in ineffective use of idle memory, because idle nodes are hard to find blindly, and in slowdown of active nodes, because pages are erroneously forwarded to these nodes. These experiments clearly demonstrate the importance of globally informed replacement decisions, as in GMS. Finally, we showed the CPU overhead GMS imposes on a storing node as the page rate to that node increases.

The performance results presented in this chapter demonstrate the ability of global memory management to reducing page fault latency. In the next chapter, we will examine the use of subpage transfers as a technique for further improving page fault performance.

Chapter 8

GLOBAL MEMORY AND PAGE SIZE

"Now! Now!" cried the Queen. "Faster! Faster!"

— Lewis Carroll, Alice's Adventures in Wonderland, 1865

Because pages are the fundamental transfer and access units in remote memory systems, page size is a key factor in remote memory performance. On current processors, page size has been driven upwards by two factors. First, magnetic disks have high access latency, therefore large pages and multi-page transfers are needed to amortize the cost. Second, the combination of relatively small TLBs and large physical memories on current machines causes performance degradation due to insufficient TLB coverage. TLB coverage is increased by large page sizes or superpage mechanisms [73, 72, 58]; e.g., the DEC Alpha supports page sizes from 8KB to 1MB, the SUN UltraSPARC supports page sizes from 8KB to 4MB, and the MIPS R10000 supports page sizes from 4KB to 16MB.

Unfortunately, the latency characteristics of high-speed networks are at odds with this trend. Recent research has succeeded in substantially minimizing the latency caused by operating system software for network transfers [80, 75], and newer controllers reduce latency even further [30]. Therefore, the total latency for remote memory transfers is dictated to an increasing extent by the *size* of the transfer, rather than by the controller and software overhead. For remote memory access, then, *smaller* transfers may ultimately be needed to reduce latency and improve program performance.

This chapter examines the use of *subpages* as the transfer units in remote memory systems. Subpages are hardware- or software-supported power-of-two subunits of full pages; for example, an 8K page might be managed as 8 1K subpages, 16 512-byte subpages, etc. We present several alternative techniques for using subpages for remote memory access. Using trace-driven simulation, we show how and why subpages reduce remote memory latency and improve performance of programs executing in a global memory environment. To ensure that our simulation parameters are realistic, we implemented a prototype software-supported subpage as part of our Digital-Alpha GMS prototype. Our implementation is able to satisfy a page fault on a 1K subpage stored in remote memory in

0.52 ms, compared with 1.5 ms for a full page. (By comparison, an average local disk access takes 4 to 14 ms on the same system, depending on the nature of the access – sequential or random.) Our results demonstrate the value of using subpages to reduce transfer size and latency in a high-speed network. For example, we show up to 1.8-fold speedup of memory-intensive applications executing with 1K subpages, when compared to the same applications using full 8K pages in the global memory system. Those same applications using 1K subpages execute up to 4 times faster than they would using the disk for backing store.

The remainder of the chapter is organized as follows. Section 8.1 presents further motivation for the use of subpages in global memory systems and introduces several alternative techniques for subpage management. Section 8.2 describes our simulation methodology and our subpage prototype implementation. In Section 8.3 we present performance results for several subpage management schemes. We also examine application memory access behavior and show why that behavior lends itself to the subpage models defined in Section 8.1. Finally, Section 8.4 concludes and summarizes our results.

8.1 Motivation

As mentioned above, the characteristics of modern network performance motivate a change in the transfer size for distributed memory systems. We see this graphically in Figure 8.1, which plots the transfer latency of a disk subsystem, a heavily-loaded 10 Mb/s Ethernet, a lightly-loaded Ethernet, and an ATM network as a function of page size on a DEC Alpha workstation. The graph demonstrates four key points. First, the disk subsystem exhibits high latency even for a "zero-length" page, as expected. Second, the networks have much lower initial overhead; therefore the linear increase in the transfer time with page size accounts for a more substantial fraction of the total latency. Third, even for an ATM-speed network, we can reduce latency substantially with smaller packets; this is difficult to see in the figure, because the scale is distorted by the high disk access time. The final interesting point shown by the graph is that even Ethernet, while much worse than disk for transferring large pages, would still have better latency than disk for very small pages.

How important are these measurements to a system using global page caching? For GMS, the complete latency for faulting an 8K Alpha page into memory from a remote node, including software overhead and management messages, is about 1.6 ms. Of that time, 1.03 ms is due to network and controller time for sending the page from the source node to

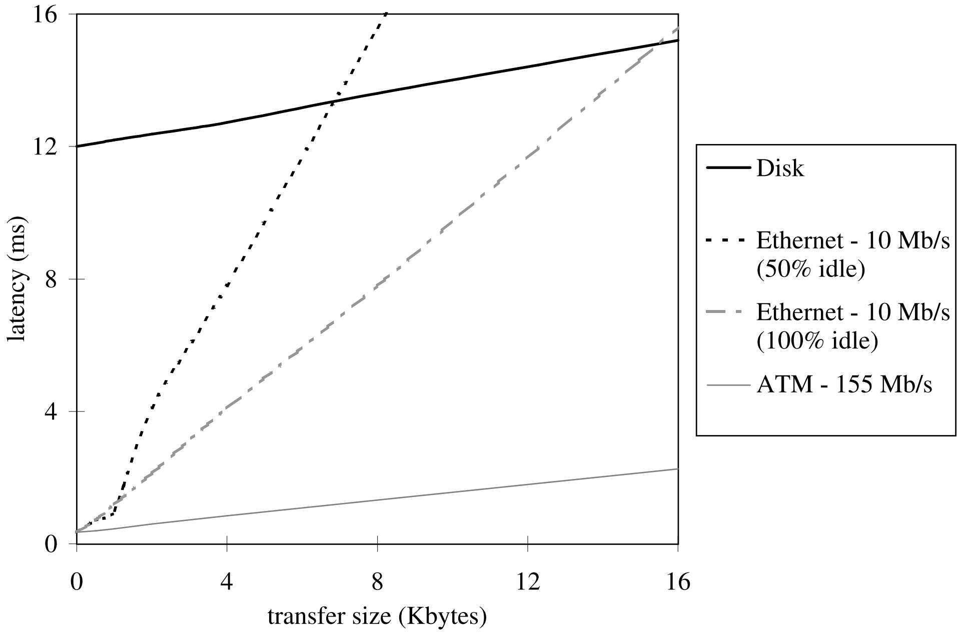


Figure 8.1: Latency vs. page size for disks and networks.

the destination. If the latency for a remote page fault is to be greatly reduced, then reducing this on-the-wire time is crucial. Moreover, in our experiments, the speedups using global memory management were close to the maximum achievable, given the ratio of disk access to remote memory access time. Therefore, reducing remote memory latency is important for obtaining further performance gains.

8.1.1 Using Subpages for Global Memory

In global memory systems such as GMS, pages are the fundamental units of access detection and transfer; because the virtual memory system is the only hardware mechanism available to trap access to non-resident storage, we must move data in integral page-sized units.

A *subpage* is a power-of-two subunit of a full page. Conceptually, a system supporting subpages would permit any (incomplete) subset of the subpages to be present within a full page, and would detect and trap attempted access to any non-resident subpage. In this way, when a fault occurs, the operating system could (a) transfer only the subpage containing the word accessed, and (b) restart the program immediately on receipt of that subpage. Because subpages are much smaller than full pages, the subpage transfer completes quickly and the program suffers a smaller latency. Whether this benefits the program in the long run depends on a number of issues, which we discuss shortly.

We have defined several possible techniques for handling subpage transfers:

1. *Lazy Subpage Fetch.* Transfer the faulted subpage. Bring the remaining subpages only on demand. This is equivalent in many respects to simply reducing the page size.

2. *Eager Fullpage Fetch.* Transfer the faulted subpage and restart the program. Asynchronously, transfer the remainder of the *full* page as a large transfer.

3. *Subpage Pipelining.* Transfer the faulted subpage and restart the program. Asynchronously, send the remainder of the subpages on the page. Because we are sending the remainder of the page in *subpage* units, the transfers can be *sequenced* to ship the most likely to be accessed subpages first.

Lazy subpage fetch is likely to perform poorly if the program eventually touches many subpages of the faulted page; i.e., fetching all of the subpages, one at a time, will be much worse than faulting the full page. On the other hand, if the page is read-only and the program touches only one or two subpages, the approach might be beneficial. Lazy subpage fetch is similar in many ways to a system with small pages; we compare the two approaches at the end of this section.

Eager fullpage fetch reduces the initial fault latency by transferring only a single subpage; it then *overlaps* the I/O of the remainder of the page with the execution of the program. Note that this approach could benefit execution in two ways. Assume that a program faults a subpage i on a non-resident page, P. First, if the time between that fault on i and the access to another subpage on P is greater than the transfer time for the rest of the page, we will have transferred the full page yet suffered the latency of only the single subpage, i. Second, if the program faults subpage j on *another* page, Q, before accessing another subpage on P, then the transfer of P can be overlapped with the transfer of j. In the first case computing is overlapped with I/O, while in the second two I/O transfers are overlapped.

The third scheme, subpage pipelining, is somewhat similar to eager fullpage fetch. However, in this case, the faster arrival of additional subpages has the potential to further reduce the time the program waits for the remote data, in cases where the time between the fault and the next subpage access on P is less than the full-page transfer time. As we will see, the first subpages to arrive following arrival of the faulted subpage take relatively little additional latency. This scheme is similar in some ways to the pipelining of cache data into cache lines over small buses [18].

It is interesting to compare our subpage schemes with an architecture that uses small pages (i.e., the size of one of our subpages). A major disadvantage of the small page scheme, relative to subpages, is the reduced TLB coverage and therefore higher TLB miss rate that small pages would incur. Furthermore, previous work [45] has shown that although smaller transfers offer the potential for increased locality, this advantage is outweighed by the increased overhead of the multiple requests required. We performed experiments to confirm that this is true for our environment as well. Therefore, we consider neither small pages nor lazy subpage fetch in this chapter.

8.1.2 Discussion

From the presentation above, we saw that there are two ways to benefit from subpages: computational overlap and I/O overlap. It is not obvious *a priori* whether this overlap can, in fact, be achieved. In particular, for eager fullpage fetch, the interval between arrival of the subpage and arrival of the follow-on full page is long, for example, on the order of 72,000 memory accesses for a 1K subpage. To completely overlap computation with that interval would require that no accesses occur to the incomplete parts of the page while the I/O is in progress, which is unlikely. To overlap I/O, on the other hand, requires that the program fault another page before it accesses a non-resident subpage on the page; yet for acceptable program performance, the average time between page faults must be much larger than the time to fetch a page.

This issue of overlap causes a tension between different subpage sizes. For small subpages, the subpage fault will complete quickly, leaving more time for overlap. On the other hand, larger subpages are likely to reduce the probability of access to incomplete parts of the page. In addition, with subpage pipelining, larger subpages reduce the number of network interrupts.

The quantitative performance gain depends on two types of factors: technological factors, such as the software overhead of message transfers and the ratio between network speed and memory speed, and behavioral factors, such as the access pattern of an executing program. In the following sections we examine these issues through simulation and measurement. Our goal is to answer the following questions:

- To what extent can eager fullpage fetch improve performance, if at all? How does this vary across applications?

- What is the optimal subpage size to use? We have described tensions in both directions.

- How much of the benefit is due to overlapped I/O vs. overlapped computation? To what extent is this benefit affected by the value of the fixed overheads?

- How much additional benefit can be obtained by pipelining?

8.2 Experimental Methodology

To study the effect of subpage transfers on remote memory operations, we implemented both a trace-driven simulator and a software prototype. Our simulator provides a detailed analysis of the policies and mechanisms we wish to examine and allows us to experiment with a range of system parameters. For our prototype, we constructed a software implementation of subpages in GMS running on the DEC Alpha 250 266-MHz workstation. We used the prototype to provide measured software management and network overheads for the simulator, and to validate simulated results where possible. The prototype and simulator are described below.

8.2.1 System Prototype

Subpage protection is implemented in software by modifying the PALcode [68] on the DEC Alpha 250. Our system keeps 32 subpage valid bits for each page, one bit for each 256-byte block; the valid bits indicate which subpages are currently valid (subpages are a multiple of 256 bytes). When a page fault occurs, we (1) allocate a page of physical memory, (2) transfer the first *subpage* from remote memory on another node into its proper position on that page, (3) set the corresponding subpage valid bits for that page, (4) turn *off* read/write access to the page, and (5) continue program execution. When the program attempts to access data on a page that is incomplete (i.e., not all of its subpages are loaded), the hardware traps to PALcode to handle the access protection violation. PALcode then checks the valid bits to see whether the attempted access is to a subpage that is loaded; if so, then the PALcode emulates the read or write access. This emulation slows execution, but

Table 8.1: Performance of PALcode load/store emulation.

Operation	Performance	
	Cycles	**Time**
fast load	52	195 ns
slow load	95	361 ns
fast store	64	241 ns
slow store	102	383 ns
null PAL call	15	56 ns
L1 cache hit	3	11 ns
L2 cache hit	8	30 ns
L2 miss	84	315 ns

only on incomplete pages; as soon as the remainder of the data for a page arrives, read/write access to the page is re-enabled.[1]

Prototype Evaluation

Table 8.1 shows the performance of emulated read/write operations on our prototype, as well as several other measurements for comparison. The PALcode caches the subpage valid bits for each emulated operation; a "fast" load or store occurs when an emulated operation is to the same page as the previous emulated operation. As the table shows, a fast load is 6.5 times slower than an L2 cache hit, and 1.6 times faster than an L2 miss (assuming the loaded data is in cache). Prototype measurements and simulation results indicate that emulation slowed execution by less than 1% for the workloads we examined.

For the simulation results reported in this chapter, we assume TLB-based hardware support, which has no overhead associated with accessing resident subpages. This simply consists of an additional valid bit in the TLB per subpage.[2]

[1] An alternative technique would be that used by the Wisconsin Wind Tunnel project [57, 56] to implement additional state bits for SVM on the CM5; they used ECC bits to cause faults, however, this would still require emulating writes, and the Alpha 250 has imprecise exceptions on data parity errors, making use of parity difficult or impossible. Similar techniques have been used for trace production as well [78].

[2] The IBM 801 used a similar scheme to manage transactions on units of less than a page – in their case, for each 128-byte line [17].

Table 8.2: Page-fault latencies for *Eager-Fullpage Fetch* from remote memory. Latencies are arrival times of subpage and rest of page. Improvement potential is given as a percentage of fullpage latency.

Subpage Size (bytes)	**Latency (ms)**		**Improvement Potential**	
	Subpage	**Rest of Page**	**Overlapped Execution**	**Sender Pipelining**
256	0.45	1.49	50%	0%
512	0.47	1.46	47%	1%
1024	0.52	1.38	40%	7%
2048	0.66	1.25	23%	16%
4096	0.94	1.23	1%	17%
fullpage	-	1.48	-	-

To gain a detailed understanding of performance, we instrumented our prototype to log crucial events. We extracted median latencies for these events from logs produced by running a memory-intensive program on our instrumented kernel configured for various subpage alternatives. These values were then used to calibrate the simulator (as described in Section 8.2.2) and are summarized in the rest of this section.

Table 8.2 shows the page-fault latencies for various subpage sizes. *Subpage* latency is the time until the program resumes execution after a page fault. This latency includes two components: (1) a fixed cost of about 0.27 milliseconds for handling the fault, locating the page in the network, sending a request message to the node storing the page, processing the request message, and resuming the faulted program when the subpages arrives, and (2) a variable cost for transferring the subpage over the network and copying it into the faulting node's memory. *Rest of Page* latency is the time from the page fault until the entire page has been transferred. This latency includes the time to send the rest of the page over the network and the time to copy it into memory. The latency reduction of subpages comes at a cost of increased overhead. Subpages increase faulting-node CPU overhead by 0.08 ms to 0.48 ms and increase sending-node overhead by 0.05 ms to 0.16 ms.

Table 8.2 also shows that there are two ways that subpages can improve page-fault latency. The first, *Overlapped Execution*, is the amount of time between subpage and rest-of-page arrival that the faulted program can potentially run (this is less than the difference

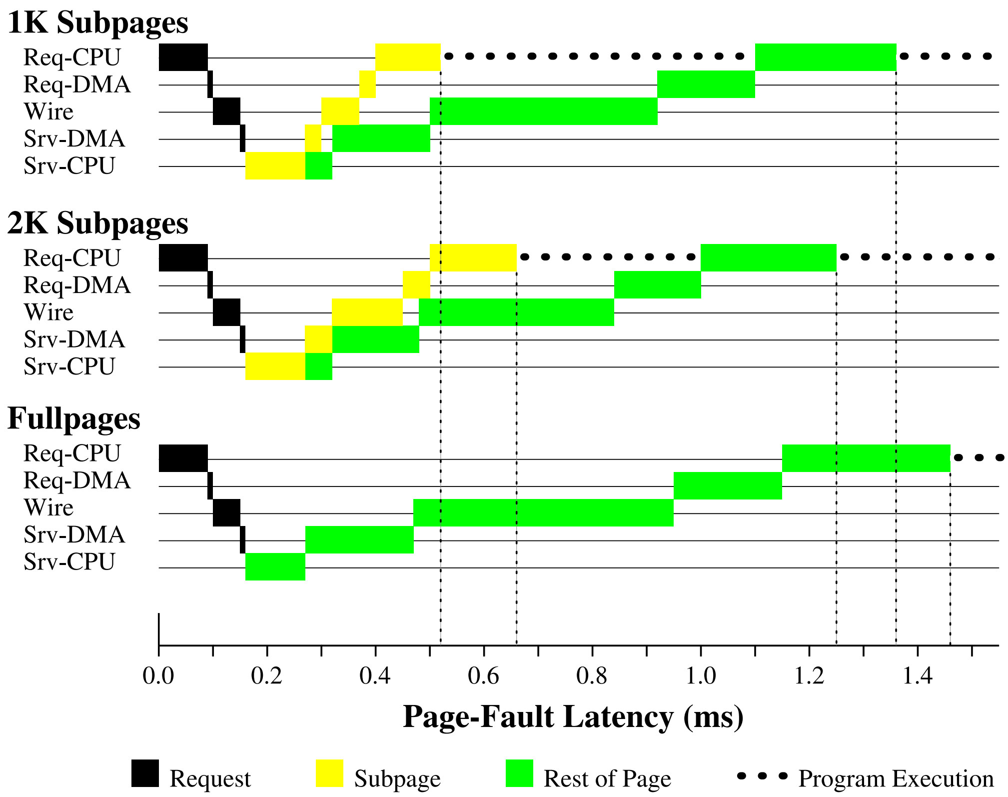


Figure 8.2: Remote page fetch timeline for fault on requesting for a page stored on a serving node.

in the two latencies by the CPU overhead of receiving the rest of the page). The second, *Sender Pipelining*, is the improvement achieved by better pipelining of the rest-of-page transfer by the sending node.

Understanding the exact operation of data transfers in a complex system, particularly where pipelining and overlap may occur in both hardware and software, is often challenging. Figure 8.2 shows a detailed timeline containing the crucial components involved in a remote page fetch operation for full 8K pages, and 2K and 1K subpages using *eager fullpage fetch* on our hardware platform.

Each timeline contains 5 components that contribute to a remote paging operation from a requesting node to a remote node serving the faulted page:

1. *Req-CPU*: Computation on the requester (the faulting node), either to process global memory management code (the thick bar) or execute the application code (the dotted line).

2. *Req-DMA*: Time required by the requester's network controller to copy data between host memory and the controller.

3. *Wire*: Time required for transmission of data on the network interconnect.

4. *Srv-DMA*: Time required on the server storing the page for the controller to copy data between host memory and the controller.

5. *Srv-CPU*: Execution on the server to process the request.

The shading on the bars indicates the work causing the particular component. For example, looking at the timeline for fullpage transfers, we see 4 black bars at the beginning of the transfer; these are for handling the fault on the requesting side and transmitting a control message to the server to request the remote page. The remaining bars are dark gray; these show the work required to transmit the page from the remote server to the requesting CPU, including the two DMAs and the wire time for the 8K page. In this case, the DMA on the requester completes at about 1.15 ms after the fault, and the application continues execution at 1.48 ms after the fault, as indicated by the dotted line.

For comparison, let's examine the timeline for a fault with 2K subpages. Again, the first 4 black bars show the sending of the request message. The light gray bars that follow are the return of the 2K subpage. Note that the DMA times and wire time for the 2K subpage are shorter than for the fullpage, so the application continues after .66 ms. The dark gray bars on this graph show the pipelining of the remainder of the page (in *eager fullpage fetch*) following the initial subpage transfer. Notice the extent to which components of this pipelined 6K transfer overlap other components of the initial 2K transfer. The second transfer arrives, interrupting the requesting CPU at approximately 1 ms. The application sees a window of .35 ms of execution before the second segment arrives, and then continues at 1.25 ms following the interrupt. In this case, then, not only did the application restart in half the time of the fullpage case, but the arrival of the *entire* transfer completed sooner, despite the fact that the same number of total bytes were sent. This occurs because of the overlap of parts of the DMA and wire transmission of the two transfers.

Finally, the top timeline shows similar detail for 1K subpages. It is surprising that the 1K case is slightly worse than the 2K case for completion of the total paging operation, finishing at close to 1.4 ms. This occurs because the DMA and wire transmission of the first 1K subpage is "too small" for optimal overlap; i.e., it completes too soon, leaving less

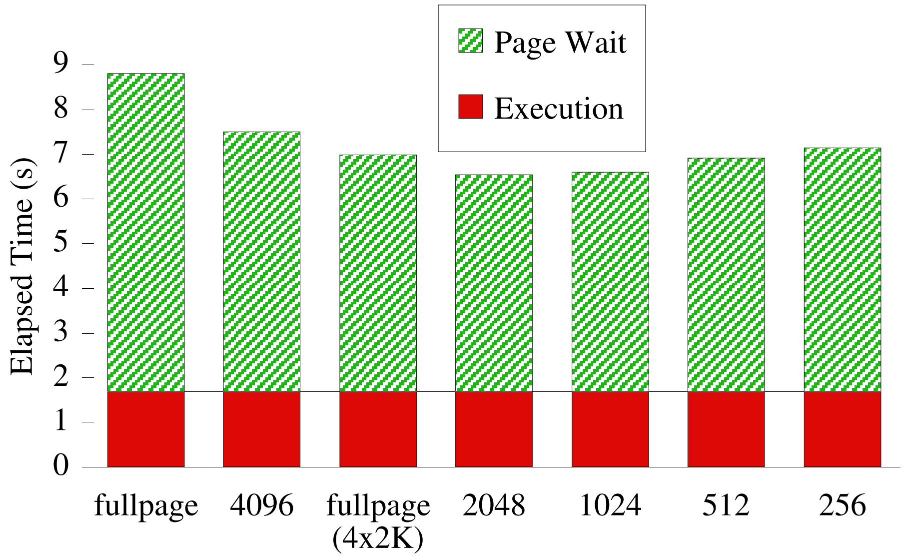


Figure 8.3: Subpage performance of Render application running on the prototype.

chance for the server DMA of the remainder of the page to be fully overlapped. This leaves a “space” on the wire between the first subpage transfer and the remaining transfer that is larger than for the 2K case.

Figure 8.3 shows the performance of the Render application (see Section 8.3) measured using the prototype. The graph shows elapsed time of Render for the standard GMS fullpage implementation, for subpages of various sizes, and for a modified (4x2K) fullpage transfer that sends a fullpage as four 2K messages. The speedup seen in the (4x2K) fullpage case is due to better pipelining of DMA and wire time; the difference between this bar and the bar for 2K subpages shows the additional benefit of subpages. The remainder of this section provides a detailed analysis using our trace-driven simulator, which was calibrated using prototype measurements.

8.2.2 Trace-Driven Simulator

Our simulator models the memory behavior of applications executing in a global memory environment. It takes as input memory reference traces generated from applications instrumented by Atom [69]. It then simulates these memory references and models the effect of paging to both local disk and to remote memory. Paging policy is determined by a configurable memory management module; an LRU policy is used by default. Once all references have been consumed, the simulator produces as output a complete description of the paging behavior of the applications that generated the traces. This description details, among other things: the number and type of page faults, the number of faults overlapped with others, the contribution of the application's CPU time to total execution time, the time spent waiting for subpages, and the time for arrival of remaining page fragments on page faults.

The simulator models page faults to remote memory using a simplified model of the timelines shown in Figure 8.2, containing three components of each transfer: request time, on-the-wire time, and receive time. The request and receive times include software and DMA overheads on both nodes. These page fault components were chosen to permit modeling of the overlap of various operations in the page fault transfers. On the remote node, the request time for the remaining page fragment is overlapped with the wire time for the subpage. On the faulting node, the receive time for the subpage is overlapped with the wire time of the remaining page fragment. In addition to overlap, the simulator models congestion delays in the network.

The simulator estimates time using memory accesses as its clock events. In order to translate latencies (e.g., for subpage transfers) into events, we measured program execution time on a DEC Alpha for several applications; we then traced those applications and ran the traces through a cache simulator to model memory accesses. Using the results of the cache simulator, we then calculated the average time per trace event (i.e., per memory access) for these programs. For our applications, we calculated an average time per simulation event to be about 12 nanoseconds, i.e., 83,000 events correspond to one millisecond of execution time.

We validated the simulator by comparing its performance improvement estimates against those measured using the prototype. Both quantitative improvement for eager fullpage fetch and the trend with subpage size agreed with the prototype measures, i.e, both found the same optimal subpage size. We could not validate subpage pipelining, for reasons discussed in Section 8.3.3.

8.3 Performance Results

This section presents measurements of subpage performance and behavior from our trace-driven simulation. The purpose of these experiments is to determine how much of the potential for improvement (shown in Table 8.2 for eager fullpage fetch) is actually achieved in practice. We also examine the interplay between program behavior and achieved performance. As noted above, we parameterized the simulator using measurements taken from our prototype implementation. The simulator has the advantage of being easier to instrument and thus it gives us more insight into the behavior of the applications.

We traced and analyzed several applications executing in the simulated subpage environment. To see the effect of heavy paging activity, we ran the applications in different memory configurations. The applications we used were the following:

Modula-3 is the DEC SRC compiler for the Modula-3 [38] programming language. We traced a Modula-3 compilation of smalldb, a library that transparently maintains a copy of in-memory data structures on secondary storage. The trace includes 87 million memory references, and from 773 to 5655 faults, depending on the memory configuration.

ld is the unix object file linker. We traced a link of Digital Unix V3.2 to generate our ld data. The trace includes 102 million memory references, and from 6807 to 10629 page faults.

Atom [69] is the instrumentation software we used to generate the traces. To obtain this trace, we instrumented Atom itself (using Atom), then traced it while processing the gzip binary. The trace includes 73 million memory accesses and from 1175 to 5275 page faults.

Render [16] is a graphics rendering program that displays a computer-generated scene from a large (over 100MB) precomputed database. The trace includes 245 million memory references and from 1433 to 6145 page faults.

gdb is the GNU debugger. We traced the initialization phase of the debugger, run without loading a program. The trace includes .5 million references and from 138 to 882 page faults.

In the results below, we use the Modula-3 trace to evaluate the subpage mechanisms in detail. Modula-3 was average among the applications we studied with respect to performance improvement due to subpages. In general, the traces from all of the applications have the same basic behavior with respect to the various subpage options examined, and thus lead to the same conclusions. We present a summary of the performance gains for the subpage mechanisms for all of the applications at the end of this section.

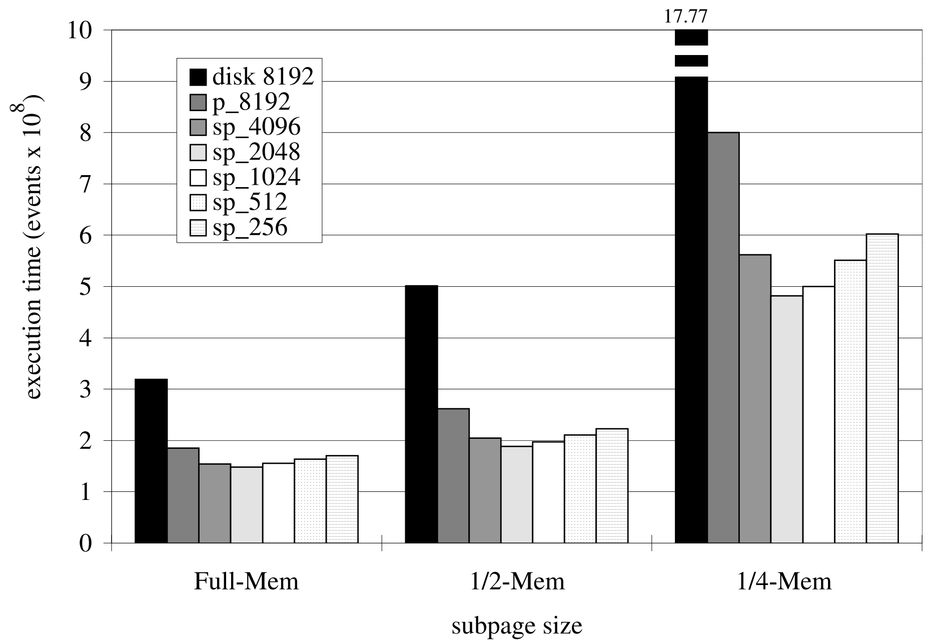


Figure 8.4: Subpage performance for 3 nemory sizes for Modula-3.

8.3.1 Performance of Eager Fullpage Fetch

Figure 8.4 shows the performance of the Modula-3 trace for a number of subpage sizes and memory configurations. This graph represents a warm (global) cache situation, that is, all pages are assumed to initially reside in remote memory. For these measurements, we used eager fullpage fetch, i.e., the first subpage is transferred with the remainder of the page following immediately afterwards. Three memory configurations are shown: *full-mem*, in which the program is given as much local memory as it needs (in this case, the faults are all initial page faults), *1/2-mem*, in which the program executes in half its maximum memory, and *1/4-mem*, in which the program executes in one quarter of its maximum memory.

For each memory configuration, the left-most bar, marked `disk_8192`, shows the performance for this trace when *all* page faults are serviced from disk, i.e., there is no global memory. The next bar, marked `p_8192`, shows the performance of global memory management using full 8K pages. As previous studies have shown, the performance improvement of global memory relative to disk is significant — in this case, the speedups range from 1.7 to 2.2, depending on memory availability. The remaining bars show the performance for subpage sizes ranging from 4096 to 256 bytes.

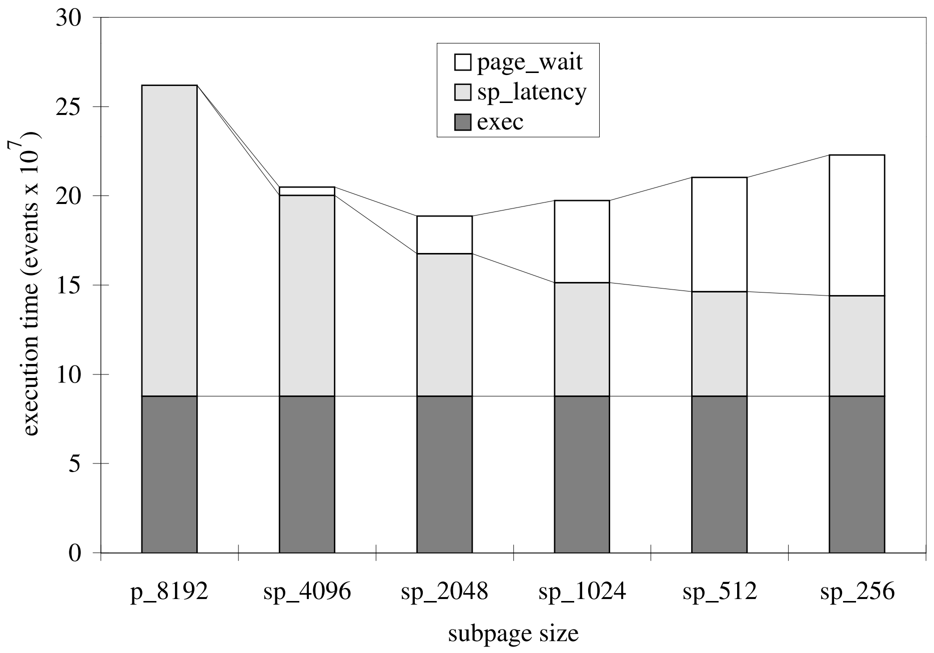


Figure 8.5: Subpage performance for Modula-3 at 1/2 memory.

Subpages offer improvement over full pages ranging from about 8% (for 256-byte subpages in full-mem) to 40% (for 2K subpages in 1/4-mem). The performance improvement of subpages increases as the program's memory demands are stressed, e.g., 1K subpages show a 16% improvement for full-mem, a 25% improvement at 1/2-mem, and a 38% improvement for the 1/4-mem configuration. Subpage sizes of 2K are the best choice for Modula-3, although all subpage sizes show improved performance relative to full pages. Over all the applications, subpage sizes of 1K or 2K were best, with only slight differences between them.

Figure 8.5 shows the results for 1/2-mem in more detail. Three components of the trace runtime are shown: the execution time of the program (exec), waiting time due to transfer of the first subpage on each faulted page (sp_latency), and time stalled waiting for the arrival of the remainder of the page (page_wait). The sp_latency bars demonstrate the fault latency improvement with smaller subpages; sp_latency decreases from 55% of total runtime at sp_4096 to 25% at sp_256. With decreasing latency, however, comes an increase in the page_wait component, from 2% at 4K to 35% at 256 bytes. This occurs for two reasons:

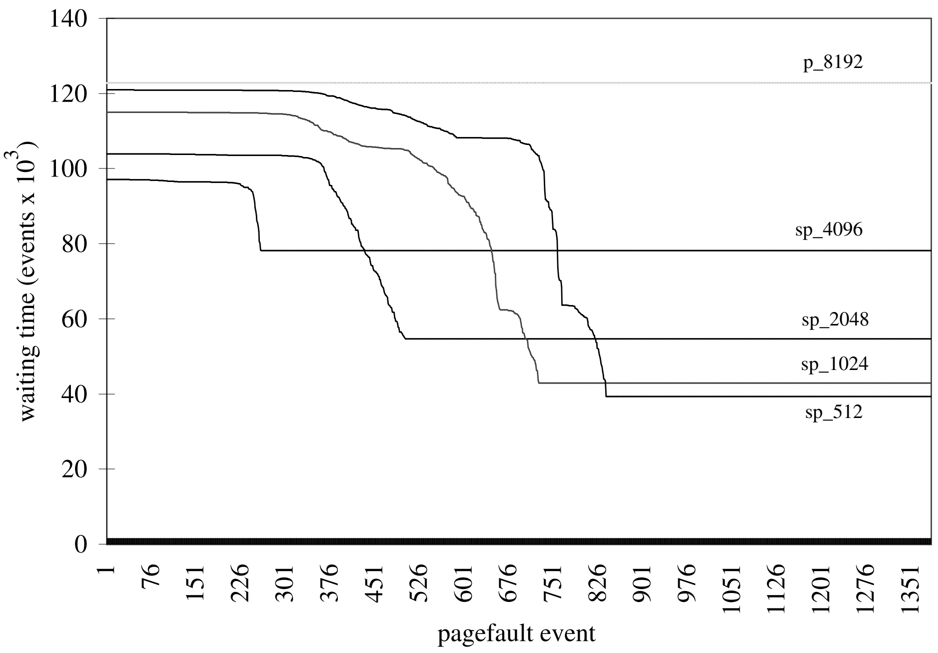


Figure 8.6: Sorted per-fault waiting times for different subpage sizes. Each curve consists of three sections: (1) a horizontal segment on the right, representing the best case, where faults wait only the subpage latency, (2) a horizontal segment on the left, representing the worst case, where faults stall until the full page arrives, and (3) a middle region, where there is some overlap.

first, smaller subpages increase the probability of accessing another subpage before either the page is complete or a miss on a different page occurs; second, because the subpage fault completes more quickly, there is increased *opportunity* (in time) to access a non-resident subpage before the full page arrives. Thus, there are both spatial and temporal reasons for the impact of smaller subpages.

In order for very small subpages to outperform larger ones, we will have to reduce this page_wait component. We examine one method of doing this, subpage pipelining, in Section 8.3.3.

8.3.2 Analysis of Subpage Performance

One might expect the benefit of eager fullpage fetch to increase as subpage size decreases, because the subpage fault completes more quickly, leaving more time to overlap. As we have seen, however, smaller subpages do not necessarily perform better. This is partially explained by the reduced opportunity for sender pipelining, as shown in Table 8.2. Additional insight is given in Figure 8.6, which shows the interaction of subpage size, latency, and waiting time. For each page fault, j, listed on the X axis, the graph plots the total waiting time that occurred for data on page j; waiting time includes the initial subpage latency plus any waiting time afterwards for arrival of the remainder of the page. The faults are sorted by waiting time, with the highest waiting times on the left, and the lowest waiting time on the right.

All of the subpage curves in Figure 8.6 are composed of three sections:

1. A horizontal segment in the lower right. The right intercept of this segment, for each curve, is the time to transfer the subpage of the specified size. These faults saw the best case: they waited only for the subpage transfer to complete, and then overlapped computation and I/O with the rest of the full-page transfer.

2. A horizontal segment in the upper left. The left intercept of this segment, for all curves, is the time to transfer the full page. These faults saw the worst case: they were not able to overlap the full-page I/O, and quickly blocked waiting for the remainder of the full page to arrive.

3. A small region in the middle, where some overlap occurred.

In this figure, we see that all subpage sizes benefit, by different amounts, relative to the normal 8K full-page global memory system. As well, we see that program behavior changes with subpage size in two distinct ways. As subpage size decreases, faults that are fully overlapped have higher benefit, as shown by the decreasing Y intercepts on the right. On the other hand, the marginal benefit decreases with decreased page size. Moreover, there are fewer faults that achieve best-case overlap, as shown by the decreasing *length* of the "best-case" (right-hand) segment. This decrease occurs because smaller subpages increase the probability of accessing data on another (non-resident) subpage. For Modula-3, 2K subpages offer the best trade-off between these alternatives.

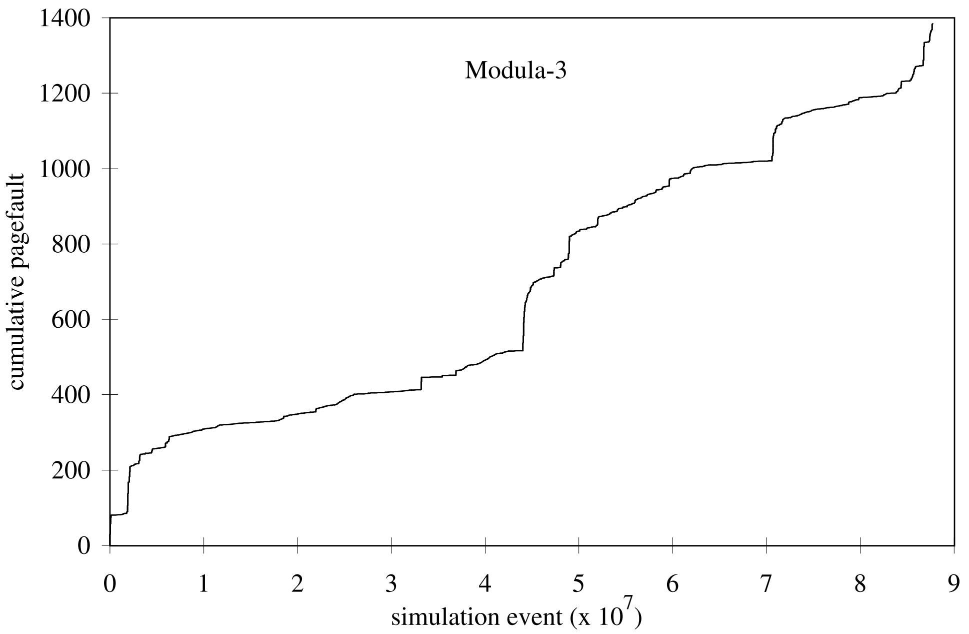


Figure 8.7: Temporal clustering of page faults for Modula-3.

As discussed in Section 8.1.2, full overlap should be difficult to achieve. It is therefore surprising that for all subpage sizes, a large fraction of the page faults achieve best-case overlap. If computational overlap accounted for a significant fraction of the total overlap, we might expect more of the page fault events to fall in the middle region of the curve. In fact, our measurements confirm that most of the speedup can be attributed to parallel I/O, rather than computational overlap.

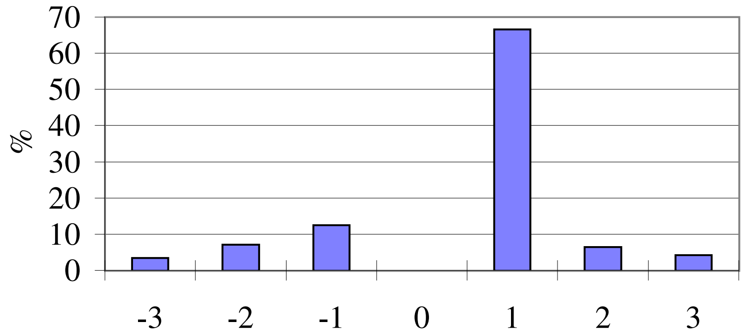
It is surprising that so much I/O overlap is possible, given that the average time between faults is much larger than the time to fetch the rest of the page. The reason this is possible is that many programs with low fault rates undergo periods of high faulting, e.g., during a phase change. Figure 8.7 plots the temporal clustering of page faults for Modula-3. For each simulation event, the graph shows the number of page faults that have occurred up to that point in time. Horizontal segments of the curve represent periods of high computation and little fault activity; vertical segments of the curve are periods of high fault rate. I/O overlap occurs mostly during the high-fault intervals; the larger the fraction of faults that occur during these periods of high faulting the greater the expected increase in performance from eager fullpage fetch.

8.3.3 *Pipelining Subpage Access*

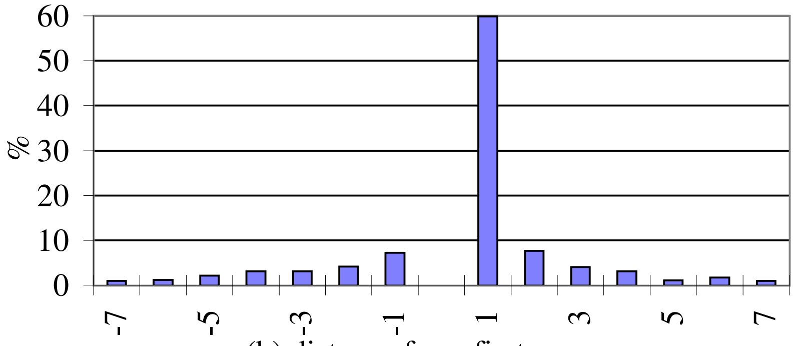
From Figure 8.6, we see that small subpages reduce the initial fault latency, but they also increase the probability of blocking before the full page arrives. Recall that in all measurements shown above, the remainder of the page is transferred in a large message following the initial subpage transfer. An alternative scheme is to *pipeline* that remaining data in multiple smaller messages. Pipelining has the potential to further improve performance for several reasons. First, the initial transfer of the faulted subpage includes the additional software latency of finding the node storing the page, sending it a message, and so on. Once the source node has been contacted, it can send the pipelined sections immediately after the initial subpage; these transmissions will have no additional latency beyond the minimum due to the time on the wire, the controller, and the receiver-side interrupt. Therefore, small pipelined follow-on transfers will complete quickly, one at a time, after receipt of the initial subpage; if the receiver-side interrupt overhead is small, the faulting program will be able to continue to execute while the pipelined subpages arrive. Second, with pipelining we have the potential to sequence the pipelined segments if we have information about the likelihood of accessing particular subpages relative to the first fault. The goal is to have the pipelined subpages arrive in the order in which they are most likely to be accessed, so as to reduce the program's stall time.

The crucial issue, then, is to determine the most likely order of access. To explore this issue, we modified our simulator to record the subpage number of the first *different* subpage to be touched after the initial subpage fault. Figure 8.8 shows this pictorially. The X axis shows the distance (in subpages) from the initial faulted subpage to the subpage next accessed. The Y axis shows the percentage of next accesses for each subpage distance. Figure 8.8a shows results for 2K subpages, while Figure 8.8b shows results for 1K subpages. In both cases, we see that these programs exhibit significant spatial locality: there is a high likelihood that the next subpage faulted on the same page will be the next consecutive subpage (distance +1).

Below we report simulation results examining pipelining and its potential benefits. In the simulation results shown we assume zero CPU overhead on the receiving node for the follow-on pipelined subpages. In our current prototype using the AN2 controller, however, each pipelined subpage causes an interrupt whose handling cost exceeds the wire time for the subpage (e.g., the overhead is 68 μs for a 256-byte subpage and 91 μs for a 1K subpage), making computational overlap impossible. This overhead could be reduced by eliminating



(a) distance from first access:
2K subpage



(b) distance from first access:
1K subpage

Figure 8.8: Distribution of distances to next accessed subpage on the same page.

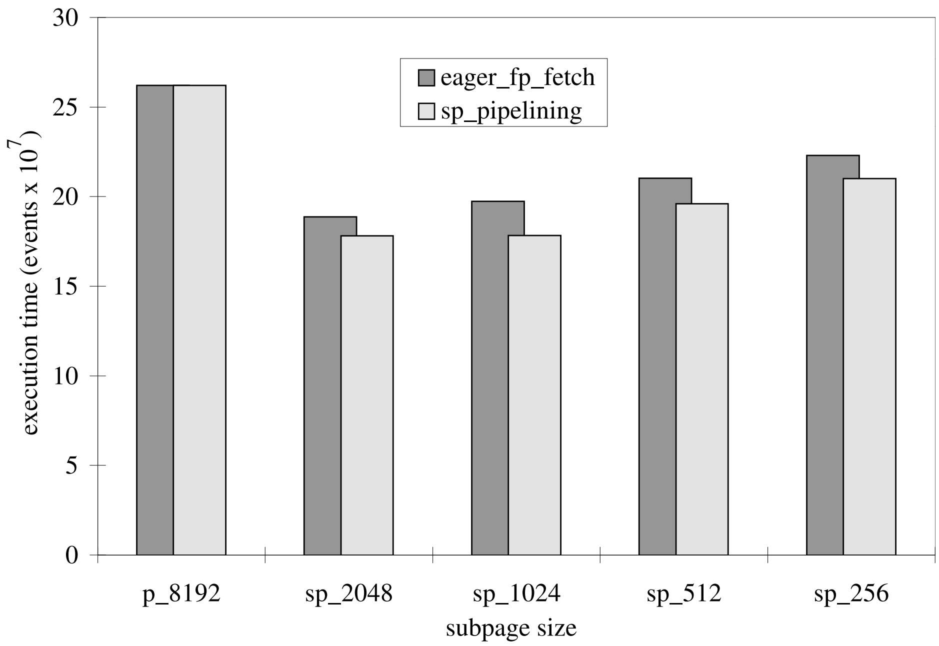


Figure 8.9: Comparison of Eager-Fullpage Fetch and Subpage Pipelining (Modula-3, 1/2 mem).

the need for copying. Eliminating the interrupt entirely is more difficult with the AN2, and would require a network controller that is able to update subpage valid bits directly. Therefore, on our current prototype, software pipelining does not outperform eager fullpage fetch. Our simulation results show the performance improvement subpage pipelining could yield if these overheads could be reduced or eliminated with a controller better suited to the task.

Based on the measurements from Figure 8.8, we simulated subpage pipelining where we transfer the faulted subpage and pipeline behind it the following and preceding subpages (the +1 and -1 subpages) on the page. The remainder of the page then follows in one message. Figure 8.9 shows the additional benefit that can be achieved through subpage pipelining on the Modula-3 trace with the 1/2 memory configuration, relative to subpages without pipelining. Note that pipelining will only reduce waiting that occurs after the first subpage arrives (shown as page_wait in Figure 8.5). At subpage size of 1K, pipelining reduces the page_wait component by 42%; however, relative to the entire execution at 1K,

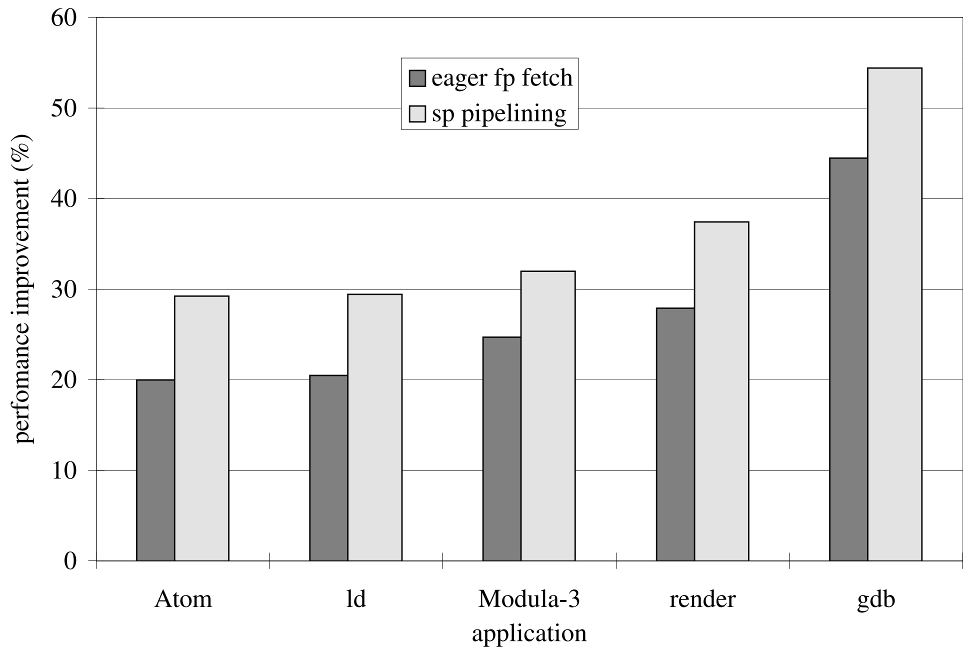


Figure 8.10: Reduction in execution time for Eager Fullpage Fetch and Subpage Pipelining (1/2-mem, 1K subpages).

the reduction is only 10%. The improvement is larger for smaller memory configurations.

We can understand the effect of pipelining by looking back at the waiting time curves in Figure 8.6. Pipelining will not affect the length of the "best-case" (lower-right) part of the curves because it does not affect the subpage latency. It can, however, significantly reduce the length of the "worst-case" (upper-left) segments of the curves; with pipelining, an access that would have blocked waiting for the full-page transfer time can continue much sooner, because the subpage it referenced has been pipelined and arrives more quickly.

In our studies, we simulated several different pipelining schemes. For example, in one experiment, we doubled the size of the pipeline transfers. That is, for 512-byte subpages, we sent the faulted 512-byte subpage followed by a pipelined transfer of the next 1K bytes. The motivation for choosing this strategy, particularly for small pages, is that there is little additional latency for doubling the length of the follow-on transfer. As another example, we transferred twice the subpage size for the initial fault, choosing to send either the preceding or following page along for the ride, depending on *where* in the subpage the faulted word was located. In general, we found that all of the schemes showed various amounts of improvement over eager fullpage fetch, but none made a substantial improvement relative to the basic scheme shown in Figure 8.9.

8.3.4 Summary

Overall, the simulation results for the Modula-3 compilation show that subpages have the potential to reduce latency for global memory access. Figure 8.10 summarizes the performance gains for eager fullpage fetch and subpage pipelining for all five of our applications, when executing in 1/2-mem configuration with 1K subpages. All applications show measurable benefit due to subpages; in two cases the benefit is greater than for the Modula-3 compilation. This further indicates that subpage mechanisms may be worthwhile. The performance increase due to subpages ranges from 20% to 44% for eager fullpage fetch and from 30% to 54% for subpage pipelining.

For current high-speed networks and memory technology, most of the benefit we see for eager fullpage fetch derives from overlapped I/O. Indeed, we measured for each application in Figure 8.10 the percentage of speedup improvement due to overlapped I/O, which varied from 53% (for Atom) to 83% (for gdb).

Figure 8.11 further supports the observation that the applications that obtain the greatest speedup are those with the largest fraction of faults occurring during periods of high fault rate. Here we see that for gdb, periods of high fault rate account for the majority of the page faults, as indicated by the steep vertical jumps in the curve. In contrast, for Atom all regions have relatively low fault rate, as indicated by the relatively smooth increase in the curve. As we would expect, gdb benefits significantly more from subpages than Atom.

Applications running on any given network memory system will have different degrees of nonuniformity in their fault rates. Often applications will have significant periods of high fault rate due to phase changes. Thus, intuitively, the benefit due to overlapped I/O estimated in our simulation experiments is likely to be observed in real workloads.

We have found that subpage pipelining improves performance for all the applications we examined, assuming an intelligent controller that eliminates the CPU overhead for pipelined subpages. This benefit derives from a combination of increased computational overlap and the new opportunities for I/O overlap. Note that the relative improvement of subpage pipelining is larger for the applications that derive smaller benefit from eager fullpage fetch.

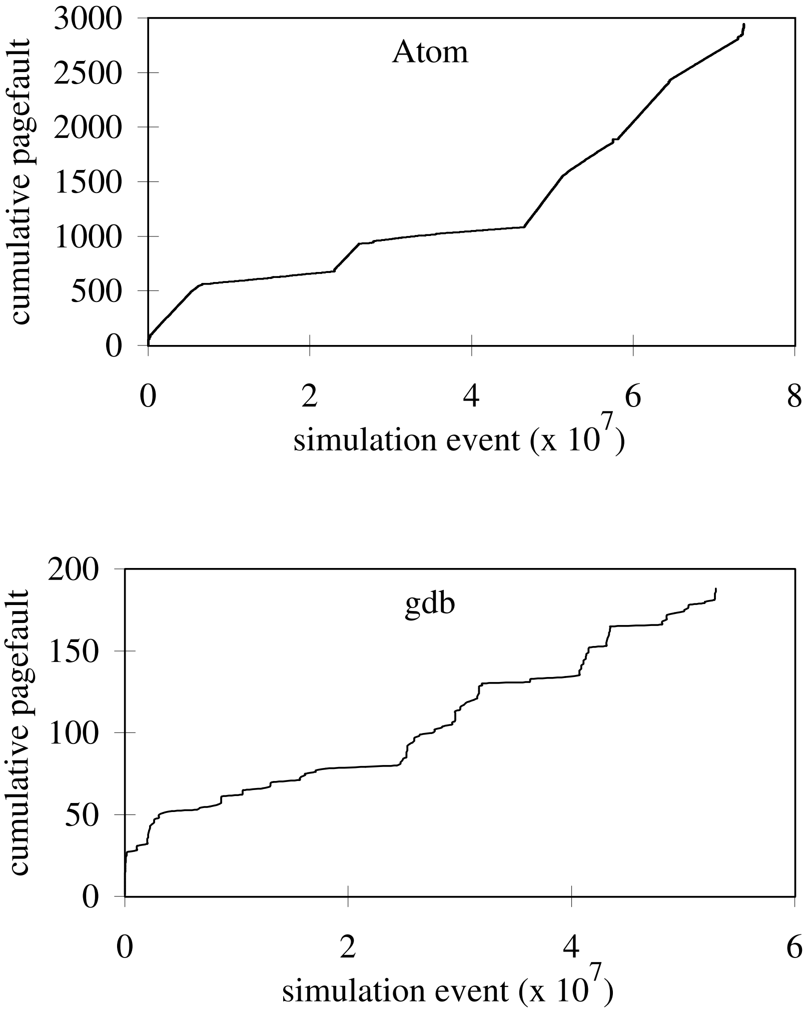


Figure 8.11: Temporal clustering of page faults for gdb and Atom.

Finally, while for current technological parameters our simulations indicate that the optimal subpage size is about 2K, we might expect that size to decrease in the future, particularly for subpage pipelining, as the ratio of network speed to memory speed increases.

8.4 Summary

This chapter examined the use of subpages to reduce the latency of network access in a network-wide memory system. Subpages are motivated by high-speed networks, whose transfer time is to a large extent proportional to message size.

We presented two techniques for using subpages. Eager fullpage fetch brings a faulted subpage and sends the remainder of the page as a large follow-on transfer. Subpage pipelining sends the page in smaller units following the faulted subpage, with the hope that those follow-on subpages will arrive (just) before they are needed.

Our prototype demonstrates that on ATM-speed networks, subpage transfers can substantially reduce fault latency relative to full pages. For example, on our GMS Alpha/AN2 prototype, a remote memory fault, even for 1K subpages, completes in about one third the time of an 8K full-page fault from remote memory. This is between 7 and 28 times faster than a fault serviced from disk by the NFS file system, depending on the nature of the file (e.g., sequential vs. random access). The need to fetch less than a page on a fault will increase with increasing page sizes, which are required by huge primary memories and small on-chip TLBs. Subpages would be best supported in hardware using extra per-page valid bits, but they can be supported in software as well, using several different techniques. Our prototype uses PALcode on the Alpha to trap access to incomplete pages, and emulates reads and writes directed to valid subpages. Despite the emulation, our prototype achieves speedup, e.g., 24% performance improvement over fullpages for eager fullpage fetch with 2K subpages on the Render application.

Our detailed simulations, validated by prototype measurements, show the performance benefits of using subpages. Our programs saw up to a 44% improvement when using subpages compared to full pages, and more with pipelining. Our "worst" application was able to decrease execution time by 20% with 1K subpages relative to full 8K pages, and by 29% when subpage pipelining was used. A detailed examination of the behavior of our applications shows that most of the benefit comes from I/O overlap.

Our analysis shows that spatial locality for these programs requires that the full page be transferred eventually. That is, simply reducing the page size to support smaller pages would actually degrade performance. However, by fetching most of the page asynchronously, as with eager fullpage fetch, we obtain significant benefit and improve TLB coverage relative to smaller pages as well.

As networks continue to improve, network-wide memories and subpage transfer units will be one key approach to achieving high performance.

Chapter 9

CONCLUSIONS

"Pooh," he said, "where did you find that pole?"
Pooh looked at the pole in his hands.
"I just found it," he said. "I thought it ought to be useful. I just picked it up."
"Pooh," said Christopher Robin solemnly, "the Expedition is over. You have found the North Pole!"
"Oh!" said Pooh.

— A. A. Milne

9.1 Summary

Current-generation networks, such as ATM, provide an order-of-magnitude performance increase over existing 10 Mb/s Ethernets; another order-of-magnitude—to gigabit networks—is visible on the horizon. Such networks permit a much tighter coupling of interconnected computers, particularly in local area networks. To benefit from this new technology, however, operating systems must integrate low-latency high-bandwidth networks into their design, in order to increase the performance of both distributed and parallel applications.

We have shown that global memory management is one practical and efficient way to share network-wide memory resources. We have designed a memory management system that attempts to make network-wide decisions on memory usage, dynamically adjusting the local/global memory balance on each node as the node's behavior and the network's behavior change. Our system does not cause data loss should nodes fail, because only clean pages are cached in global memory; cached data can always be fetched from disk if necessary.

The goal of any global memory algorithm is to reduce the average memory access time, by substituting remote-memory accesses for disk accesses. Key to our algorithm is its use of periodically-distributed network-wide age information in order to: (1) house global pages in those nodes most likely to have idle memory, (2) avoid burdening nodes that are actively using their memory, (3) ultimately maintain in network-wide primary memory the pages most likely to be globally reused, and (4) maintain those pages in the right places. Algorithms that do not have these properties are unlikely to be successful in a dynamic network environment.

Key to global memory management is its ability to make efficient and globally optimized page replacements. Global page replacement is a two step process: first, a local page is replaced to make room for a newly faulted page; second, the replaced page is forwarded to another node where it replaces a *globally* old page. Finding the nodes with the network's oldest pages is a key challenge for global memory management; doing so requires globally coordinated use of age information for every page in the network. Our GMS algorithm collects this information using a centralized coordinator node, an approach that minimizes the overhead on the network's active nodes. Decision-making is coordinated using a probabilistic algorithm in which nodes pick old nodes independently of each other, using a probability distribution that reflects the location of the network's oldest pages. Using this probabilistic approach, our algorithm approximates global LRU with minimal overhead.

A key advantage of our algorithm's global-LRU approximation is that the algorithm is stable across a broad range of workload and network characteristics and that it is thus relatively easy to analyze. Alternative approaches proposed by others have been shown to work well for Unix-like filesystem access patterns (i.e., small files accessed sequentially, and in their entirety, in networks that are mostly idle). Our analysis of these algorithms in this dissertation, however, shows that they do not provide the same stability features of our algorithm. For certain workloads they perform significantly worse than GMS and for others, their performance is difficult to predict.

We have implemented our algorithm in the Digital Unix operating system running on a network of Alpha workstations connected by a Digital AN2 ATM network. Our measurements show the underlying costs for global memory management operations in light of a real implementation, and the potential benefits of global memory management for applications executing within a local-area network. GMS reduces page fault latency to 1.5 ms and IO- and memory-intensive programs speedup between 1.5 to 3.5 times.

Remote-memory page fault latency can be further reduced using subpages, which are motivated by high-speed networks, whose transfer time is to a large extent proportional to message size. We have implemented alternative schemes that divide pages into subpages and satisfy a fault by transferring all but the faulted subpage in parallel with the continued execution of the faulted program. Our measurement show that page fault latency is reduced to 0.5 ms using 1K subpages and that programs see up to a 44% performance improvement over standard GMS.

9.2 Future Work

The most fundamental concern with respect to network-wide resource management is the impact of failures. In most distributed systems, failures can cause disruption, but they should not cause permanent data loss. Temporary service loss is common on any distributed system, as anyone using a distributed filesystem is well aware. With our current algorithm, all pages in global memory are clean, and can therefore be retrieved from disk should a node holding global pages fail. The failure of the coordinator or master nodes is more difficult to handle; while we have not yet implemented such schemes, simple algorithms exist for the remaining nodes to elect a replacement.

A reasonable extension to our system would permit dirty pages to be sent to global memory *without* first writing them to disk. Such a scheme would have performance advantages, particularly given distributed filesystems and faster networks, at the risk of data loss in the case of failure. A commonly used solution is to replicate pages in the global memory of multiple nodes; other approaches such as transaction logging are also possible.

Another issue is one of trust. As a cluster becomes more closely coupled, the machines act more as a single timesharing system. Our mechanism expects a single, trusted, network-wide administrative domain. All of the kernels must trust each other in various ways. In particular, one node must trust another to not reveal or corrupt its data that is stored in the second node's global memory. Without mutual trust, the solution is to encrypt the data on its way to or from global memory. This could be done most easily at the network hardware level [63].

Our current algorithm is essentially a modified global LRU replacement scheme. As discussed in Section 2.2.1, it is well known that in some cases, such as sequential file access, LRU may not be the best choice. The sequential case could be dealt with by limiting its buffer space, as is done currently in the Digital Unix file buffer cache. Other problems could exist as well. The most obvious is that a single badly-behaving program on one node could cause enormous paging activity, effectively flushing global memory. Of course, even without global memory, a misbehaving program could flood the network or disk, disrupting service. Again, one approach is to provide a threshold limiting the total amount of global memory storage that a single node or single application could consume.

If only a few nodes have idle memory, the CPU load on these nodes could be high if a large number of nodes all attempt to use that idle memory simultaneously. If there are programs running on the idle machines, this could adversely affect their performance. A possible solution is to incorporate CPU-load information with page age information, and to use it to limit CPU overhead on heavily-loaded machines.

Finally, as we saw in Section 2.1.2, cluster-networks such as Myrinet and Memory Channel offer dramatic performance improvements over high-speed local-area networks, such as the Digital AN2 network used in our GMS prototype. Cluster networks, however, are limited to only 16 or so nodes connected within close proximity of each other. Future networks are thus likely to use a combination of cluster and LAN technology to connect nodes in a hierarchical fashion. A hierarchical network adds an additional level to the network's memory hierarchy (i.e., it contains both fast and slow remote memory) and thus complicates memory management, impacting replacement and replication policies, page-location algorithms, and global-information gathering mechanisms.

Bibliography

[1] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS Memo TM-454, Laboratory for Computer Science, MIT, 1991.

[2] American National Standards Institute, ANSI X3.139-1987. *Fiber-Distributed Data Interface (FDDI) Token Ring Media Access Control (MAC)*, November 1987.

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ATM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[4] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems*, 11(4):319–352, November 1993.

[5] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical evaluation of the CRAY-T3D: A compiler perspective. In *International Symposium on Computer Architecture*, June 1995.

[6] O. Babaoglu and W. Joy. Converting a swap-based system to do paging in an architecture lacking page-reference bits. In *Proceedings of the 8th ACM Symposium and Operating Systems Principles*, pages 78–86, December 1981.

[7] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirrif, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[8] L.A. Belady. A study of replacement algorithms for virtual-storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[9] Brian N. Bershad, Mathew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 93 COMPCON Conference*, pages 528–537, February 1993.

[10] D. P. Bertsekas and D. A. Castinon. Auction algorithm for transportation problem. *Annals of Operations Research*, pages 67–96, 1989.

[11] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L Seitz, J. N. Seizovic, and Wen King Su. Myrinet: a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[12] Pei Cao, E. W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 14–17, November 1994.

[13] M. J. Carey, D. J. Dewitt, and J. F. Naughton. The OO7 benchmark. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, May 1993.

[14] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shekita. The architecture of the EXODUS extensible DBMS. In *Proceedings of the 12th VLDB Conference*, pages 52–65, 1986.

[15] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[16] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proc. of Graphics Interface '96*, May 1996.

[17] Albert Chang and Mark F. Merge. 801 storage: Architecture and programming. *ACM Trans. on Computer Systems*, 6(1), February 1988.

[18] Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier. The memory system of a high-performance personal computer. *IEEE Trans. on Computers*, C-30(10), October 1981.

[19] Douglas Comer and James Griffioen. A new design for distributed systems: The remote memory model. In *Proceedings of the USENIX Summer Conference*, June 1990.

[20] Alan L. Cox and Robert J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[21] Cray Research Incorporated. *The CRAY T3D Hardware Reference Manual*, 1993.

[22] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.

[23] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.

[24] P. J. Denning. The Working Set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[25] L. Dowdey and D. Foster. Comparative models of the file assignment problem. *Computing Surveys*, 14(2):287–313, June 1982.

[26] R. P. Draves. Page replacement and reference bit emulation in Mach. In *Proceedings of the USENIX Mach Symposium*, pages 201–212, November 1991.

[27] S. Haridi E. Hagersten, A. Landin. Ddm — a cache-only memory architecture. *Computer*, 25(9):44–54, 1992.

[28] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5), May 1986.

[29] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[30] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnick, Liviu Iftode, and Kai Li. Early experience with message-passing on the Shrimp multicomputer. In *Proc. of the 23rd International Symposium of Computer Architecture*, May 1996.

[31] Edward W. Felten and John Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.

[32] FORE Systems, 1000 Gamma Drive, Pittsburgh PA 15238. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.

[33] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server DBMS architectures. In *Proceedings of the 18th VLDB Conference*, August 1992.

[34] K. W. Froese and R. B. Bunt. The effect of client caching on file server workloads. In *Proceedings of the Twenty-Nineth Hawaii International Conference on System Sciences*, January 1996.

[35] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences Vol. I*, pages 134–143, Jan 1994.

[36] R. S. Garfinkel and G. L. Nemhauser. *Integer Programming*. John Wiley and Sons, New York, 1972.

[37] R. B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1), February 1996.

[38] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NY, 1992.

[39] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[40] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. Evans III, A. R. Karlin, H. M. Levy, and M. K. Vernon. Reducing network latency using subpages in a global memory environment. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

[41] R. P. LaRowe Jr. and C. S. Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel and Distributed Computing*, 11(2):112–129, February 1991.

[42] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Conference*, pages 115–132, January 1994.

[43] Kendall Square Research Inc., 170 Tracer Lane, Waltham, MA 02154. *Kendall Square Research Technical Summary*, 1992.

[44] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[45] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File access performance of diskless workstations. *ACM Trans. on Computer Systems*, 4(3), August 1986.

[46] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1(5):842–857, November 1983.

[47] Avraham Leff, Joel L. Wolf, and Philip S. Yu. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11), November 1993.

[48] Avraham Leff, Joel L. Wolf, and Philip S. Yu. Efficient LRU-based buffering in a LAN remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 7(2), February 1996.

[49] Daniel Lenoski, James Loudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessey. The DASH prototype: implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, May 1992.

[50] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[51] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 177–189, January 1996.

[52] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[53] Michael N. Nelson. Virtual memory vs. the file system. Technical Report 90/4, Digital Western Research Laboratory, March 1990.

[54] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[55] B. Reed and D. D. E. Long. Analysis of caching algorithms for distributed file systems. *Operating Systems Review*, 30(3):12–21, July 1996.

[56] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D.Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proc. of the ACM Sigmetrics Conf. on Measurement and Modelling of Computer Systems*, May 1993.

[57] Steve K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Proc. of the 2nd USENIX Symp. on Micorkernels and Other Kernel Architectures*, September 1993.

[58] Ted Romer, Wayne Ohlrich, Anna Karlin, and Brian Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proc. of the 22nd Annual Int. Symp. on Computer Architecture*, June 1995.

[59] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer Usenix Conference*, 1985.

[60] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, October 1996.

[61] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

[62] Bill N. Schilit and Dan Duchamp. Adaptive remote paging. Technical Report CUCS-004091, Department of Computer Science, Columbia University, February 1991.

[63] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, October 1991.

[64] A. Silberschatz. *Operating Systems Concepts*. Addison-Wesley Publishing Company, Inc., Reading, Mass., 1991.

[65] Silicon Graphics Inc., 2001 N. Shoreline Blvd, Mountain View, CA. *Power Challenge: Technical Report*, 1996.

[66] J. P. Singh, T. Joe, A. Gupta, and J. L. Hennessy. An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *Proceedings of Supercomputing '93*, pages 214–225, November 1993.

[67] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, One Burlington Woods Drive, Burlington, MA 01803, 1992.

[68] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, One Burlington Woods Drive, Burlington, MA 01803, 1992.

[69] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. Technical Report 94/2, DEC Western Research Lab, March 1994.

[70] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[71] D. Krekel T. Bönniger, R. Esser. CM-5E, KSR2, Paragon XP/S: A comparative description of massively parallel computers. *Parallel Computing*, 21(2):199–232, 1995.

[72] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proc. of the 6th Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, October 1994.

[73] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proc. of the 19th Annual Int. Symp. on Computer Architecture*, May 1992.

[74] A. S. Tanenbaum. *Operating Systems – Design and Implementation*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[75] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed systems. In *Proc. of the 6th Int. Conf. on Arch. Support for Prog. Languages and Operating Systems*, October 1994.

[76] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.

[77] R. Turner and H. Levy. Segmented FIFO page replacement. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 48–51, September 1981.

[78] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Sechrest. Trap-driven simulation with Tapeworm II. In *Proc. of the 6th Int. Conf. on Arch. Support for Prog. Languages and Operating Systems*, October 1994.

[79] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1996.

[80] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[81] B. W. Wah. File placement on distributed computer systems. *Computer*, 17(1):315–331, January 1984.

[82] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 2–11, may 1993.