INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Cooperative Caching in Local-Area and Wide-Area Networks

Geoffrey Michael Voelker

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

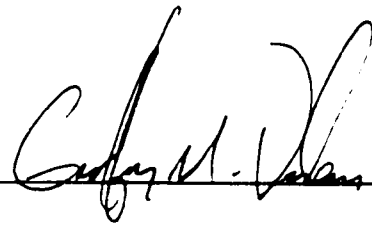
Doctor of Philosophy

University of Washington

2000

Program Authorized to Offer Degree: Computer Science and Engineering

In presenting this dissertation in partial fulfillment of the requirements for the Doctorial degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholary purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature 

Date JUNE 9, 2000

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Geoffrey Michael Voelker

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

Henry M. Levy

Anna R. Karlin

Reading Committee:

Henry M. Levy

Anna R. Karlin

Brian N. Bershad

Date: JUNE 9, 2000

University of Washington

Abstract

Cooperative Caching in Local-Area and Wide-Area Networks

by Geoffrey Michael Voelker

Co-Chairs of Supervisory Committee

Professor Henry M. Levy
Computer Science and Engineering

Professor Anna R. Karlin
Computer Science and Engineering

This dissertation extends cooperative caching systems in three new directions: (1) I study the performance tradeoffs of using load balancing techniques to reduce the contention of remote page requests at memory servers, (2) I develop a novel combined prefetching and cooperative caching system to further reduce disk stall time for I/O-bound applications, and (3) I explore the effectiveness of using cooperative caching techniques among World Wide Web proxy caches at large-scale client populations. I use analytic, simulation, and measurement techniques to explore the performance of these new directions in cooperative caching systems.

First, I develop an analytic queueing network model to evaluate the effectiveness of load balancing memory requests, and then use trace-driven simulation to study the potential harm of deviating from the replacement policy when balancing load. I find that deviating from the strict replacement policy does not significantly degrade overall application performance. Based upon these results, I conclude that cooperative caching systems can benefit considerably from load balancing requests with little harm from suboptimal replacement decisions.

Second, I present the design of a combined cooperative prefetching and caching algorithm and its implementation in a prototype system called Prefetching Global Memory System (PGMS). I measure the performance of the PGMS implementation on a workstation cluster using both synthetic

and real application benchmarks and conclude that, by using available global memory, disk, and CPU resources, cooperative prefetching and caching can obtain significant speedups for I/O-bound programs.

Lastly, I introduce a novel analytic model of Web accesses to explore the effectiveness of cooperative caching among World Wide Web proxies. This model evaluates the steady-state performance of proxy caches based upon server, object popularity, and document modification distributions. I use the model to evaluate cooperative Web proxy caching at population sizes that range from a single organization to millions of clients, explore the tradeoffs among various cooperative caching schemes, and speculate on the performance implications of future trends in Web workloads. Overall, I demonstrate that the utility of cooperative Web proxy caching is most effective among groups of small organizations, and only marginally useful for very large client populations.

Table of Contents

List of Figures

List of Tables

Acknowledgments

While finishing this dissertation – indeed, throughout my graduate studies – I had the support and assistance of a tremendous group of people. In particular, I would like to thank my advisors Hank Levy, for his guidance, patience, and commitment, and Anna Karlin, for an entirely different perspective on computer systems. Brian Bershad gave me free reign early on, and challenged me throughout. Mike Feeley's work on GMS made my thesis possible, and Jeff Chase made my thesis fast. Mary Vernon taught me queueing theory, and John Zahorjan taught me to keep my guard up. Alec Wolman was my partner in arms from the first quarter to the last. Chandu Thekkath and Dylan McNamee gave me valuable early advice on graduate school and research. Marc Fiuczynski was always willing to talk shop while getting drinkies, and Stefan Savage always had an answer to my research questions. Nancy Burr helped me keep the machines running. Frankye Jones shielded me from every bureaucratic predicament I managed to find myself in. Working with the Etch group, Brian, Hank, Alec, Ted Romer, Dennis Lee, Wayne Wong, and Brad Chen, was an unforgettable rollercoaster ride. Rich Draves, Bill Bolosky, Bob Fitzgerald, and Mike Jones mentored me as an intern at Microsoft Research. Finally, it was a pleasure to collaborate with Hervé Jamrozik, Eric Anderson, Tracy Kimbrel, Neal Cardwell, Tashana Landray, Nitin Sharma, Molly Brown, and Denise Pinnel on various research projects and papers.

Graduate school would not have been possible without the support of many friends who ensured that life was more than just work. From the beginning, Brad Chamberlain was always there to talk and listen, kept me up to date on the comics and music scenes, and allowed me to retreat with him to the cabin at the lake. Ruth Anderson and Neal Lesh trained me to run marathons, and shared a tent as snow fell overnight in Alaska. Ben Dugan trained me to climb mountains. Sung-Eun Choi always remembered birthdays and holidays, and created a work of art on my cast. Jim Fix shared cubicles with humor and fellowship, and was always willing to answer my theory questions. Iris

Zhu braved and endured my excessive work habits with warmth, care, and tenderness. Stefan was a loyal running partner and fellow deadweight on The Electric Cookie Monster Dream Team. Mike Perkowitz was the music man of the bistro. The Spuds soccer and softball teams kept my reflexes up. And, from distant cities, Ben Gilmartin shared a tradition of coffee and fireside talks over the holidays, Peter Morgan was a willing opponent in games to distract both of us from our work, and Julie Tannenbaum always asked the tough questions.

Parts of this thesis have been published previously as conference papers. Chapter 3 is based on the paper "Managing Server Load in Global Memory Systems" published in the *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* [Voelker et al. 97]. Chapter 4 is based on the paper "Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System" published in the *Proceedings of the 1998 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* [Voelker et al. 98]. And Chapter 5 is based in part on the paper "On the scale and performance of cooperative Web proxy caching" published in the *Proceedings of the 17th ACM Symposium on Operating System Principles* [Wolman et al. 99b].

Chapter 1

Introduction

High-speed switched local-area networks, such as ATM and Myrinet, have reduced the latency of network page transfers significantly below that of local disk. This hardware trend has motivated the development of software systems that use network-wide memory as a *cooperative cache*. In these systems, (1) the unused memory on idle or lightly-loaded machines is used to *cache* virtual memory pages and file blocks evicted from the memories of active machines; and (2) all of the nodes in the system *cooperate* so that local memory resources are managed to maximize overall global system performance. Previous work in cooperative caching systems has explored algorithms for coordinating cache contents, approaches for improving algorithm accuracy and efficiency, and techniques for increasing system reliability and portability.

In this dissertation, I build upon previous work to explore cooperative caching system performance in three new domains:

1. I study the performance tradeoffs of using load balancing in cooperative caching systems. Load balancing remote memory requests reduces contention at memory servers and limits the impact of serving requests on local jobs executing on memory servers. However, load balancing can also cause the system to deviate from its original replacement algorithm. This deviation can increase the overall number of disk accesses and offset the benefits of load balancing.

2. I extend cooperative caching systems to support prefetching in the presence of optional hints of future demands. Prefetching enables cooperative caching systems to use idle disk resources in the cluster to transfer data into the cooperative cache before it is accessed by applications, thereby reducing or eliminating I/O stall time at little or no cost. Furthermore, the system can

use disks on multiple idle machines to issue prefetches in parallel. Combining cooperative caching with prefetching and parallel I/O results in a system whose design and performance is significantly different than one using any single approach alone.

3. I explore the potential of applying cooperative caching techniques to large-scale wide-area distributed systems like the World Wide Web. Individual Web proxy caches cache requests on behalf of companies or organizations. Ultimately, though, the effectiveness of a proxy cache is a function of the size of the population it manages – a size often dictated by political, organizational, or geographic constraints. To overcome these constraints, multiple proxy caches can cooperative with each other to further increase total client population, improve hit ratios, and reduce document-access latency.

I use analytic, simulation, and measurement techniques to explore cooperative caching in these new domains. First, I develop an analytic queueing network model to evaluate the effectiveness of load balancing memory requests, and then use trace-driven simulation to study the potential harm of deviating from the replacement policy when balancing load. Second, I present the design of a combined cooperative prefetching and caching algorithm and its implementation in a prototype system. I then measure the performance of the implementation on a workstation cluster using both synthetic and real application benchmarks, and demonstrate the performance advantages of combining prefetching and cooperative caching. Lastly, I introduce a novel analytic model of Web accesses to explore the effectiveness of cooperative caching among World Wide Web proxies. This model evaluates the steady-state performance of proxy caches based upon server, object popularity, and document rate of change distributions, and I apply it at scales ranging from small-scale organizational populations to large-scale national populations.

The following sections discuss in more detail the new cooperative caching domains that each part of this dissertation explores and the research questions it answers, present the contributions made by this dissertation, and provide an overview of the remaining chapters.

1.1 Load balancing

A crucial aspect of the design of cooperative caching systems is the global page replacement algorithm, which selects the target nodes to receive evicted (i.e., paged-out) pages. For example, in the xFS distributed file system [Dahlin et al. 94], pages displaced from one node are sent randomly to other nodes and an algorithm called *N-chance* is used to recover from bad selections. In the Global Memory Service (GMS) [Feeley et al. 95], the system maintains global information describing the ages of pages on all cluster nodes; using this information, GMS implements an approximation to network-wide global LRU replacement. Experiments with GMS show that the use of global information improves performance relative to a random algorithm. [Sarkar et al. 96] propose a fully distributed information exchange (piggybacked on global page forwarding operations) to gather less complete but perhaps more up-to-date page age information, leading to a different approximation of global LRU replacement.

Although using age information can provide near-optimal page replacement selections, it can still lead to suboptimal overall performance due to contention at memory servers. First, dynamic application behavior can result in both sustained and short-term periods of high request rates. Sufficiently high request rates increase the latency of remote page requests due to contention at memory servers. Furthermore, the global page replacement algorithm can also concentrate old or young pages at one or a few memory servers. Combined with periods of high request rates, these concentrations will focus request traffic at these servers, further increasing contention. Second, the load imposed on any given node that is handling global memory traffic may degrade the performance of the local jobs on that node. For example, experiments with GMS showed that a node supplying pages to seven active nodes used 56% of its CPU cycles just for servicing the remote page requests [Feeley et al. 95].

The first part of this dissertation explores the use of load balancing to reduce contention at memory servers and limit the impact of serving requests on local jobs on memory servers. Server contention arises when the total request rate from active nodes to memory servers is sufficiently high that the requests queue at servers and interfere with each other. When requests queue at memory servers, they take longer to serve. And the longer they take to serve, the slower applications run when using cooperative caching. If the request rate is sufficiently high, there is so much contention

that requests to memory servers take longer than if they were to use the local disk.

Memory servers can experience extremely high request rates either in sustained stretches of time, or sudden, short-term bursts. When there are only a few memory servers for a large number of active nodes, the combined request traffic to those servers can result in sustained contention that generates sufficient overhead to negate the benefit of cooperative caching. In this case, the system should disable cooperative caching, or at least only allow a subset of the active nodes to use the memory servers.

Even with a reasonable balance of active nodes and memory servers, there are two situations that can lead to short-term, dramatic increases in the request rate to the memory servers. First, during the course of their execution applications go through periods where their peak fault rate is significantly higher than their average fault rate. For example, experiments by [Jamrozik et al. 96] found that, during the execution of the Modula-3 compiler `m3`, the average fault rate for the program was 670 page faults per second, but that there were periods where it experienced 41,500 page faults per second over the course of 5 seconds. Such page fault rates can easily overwhelm a memory server.

Second, when a node transitions state from *idle* to *active* (e.g., because the user starts to use the node again), it needs to make its memory resources available for use by local applications. To do this, the memory pages it has been storing for other active nodes now need to be moved to other servers. Considering that a node may be serving thousands of pages on contemporary machines, this transition can rapidly result in thousands of page forwarding requests to the remaining servers in the cluster.

These bursts of request traffic can experience contention if they are concentrated on just a few or even one memory server. The distribution of request traffic is determined by the global page replacement algorithm used by the cooperative caching system. When a global page replacement algorithm chooses servers based upon the age of pages on those servers, such as in GMS or the algorithm proposed by Sarkar *et al.* , then the distribution of requests to those servers will depend upon the relative ages of pages they store. If the distribution of page ages among servers is uneven, such as when one server has a concentration of old pages, then the distribution of requests to the servers will be uneven. When the request distribution is uneven, requests will concentrate on and potentially overwhelm individual servers.

A common situation where memory servers experience uneven distributions of page ages is when a node transitions state from *active* to *idle* and becomes a memory server (e.g., some time after the user stops using the node). After this transition, its memory resources become immediately available for use by the rest of the cluster. Since this memory is now free for use, active nodes will begin to concentrate their eviction requests at the newly idle node.

This situation can have long-term effects as well. For example, assume one server holds a large set of pages of the same age. When that set of pages becomes the oldest in the cluster, eviction requests from active nodes will concentrate at the server as those pages are replaced by newer ones. The combination of a concentration of oldest pages at one node, and other nodes operating at high fault rates due to transitions in application behavior or node state, can lead to significant server contention.

To mitigate the effects of contention at memory servers in these situations, this dissertation studies the potential benefit and the potential harm of using memory server CPU load information, in addition to age information, in cooperative caching replacement policies. By considering load information in the replacement algorithm, cooperative caching systems can reduce contention at servers and limit CPU utilization by remote requests. There are tradeoffs to the use of load information, however. Replacements made to balance load can cause the system to deviate from its original replacement algorithm, potentially increasing the overall number of disk accesses when load balancing is used.

Extending cooperative caching to incorporate load balancing raises a number of challenging questions. First, what is the impact on remote page fault latency of contention at memory servers? How sensitive is remote page fault latency to uneven load distributions across memory servers? Under what conditions does contention at memory servers negate the benefits of cooperative caching? What is the impact on application performance when load balancing causes suboptimal global page replacement decisions (e.g., replaces younger pages before older pages)? And finally, how can we extend the global page replacement policy to incorporate load information as well as age information? Chapter 3 explores these questions in detail.

1.2 Cooperative prefetching and caching

Cooperative caching benefits memory-intensive applications, applications whose virtual memory and file buffer cache working sets are larger than the memory resources of a single node. By caching evicted pages in the remote memories of idle nodes instead of on local disk, cooperative caching transforms slow disk faults into much faster network faults. However, I/O-bound applications, either with large working sets or a small degree of reuse, only benefit to a limited degree from cooperative caching. For example, one application executing on GMS achieved a speedup of 1.7 once the system had enough idle nodes to contain its working set [Feeley et al. 95]. Adding more nodes to the system would not improve its performance because it was limited by the I/O time required to load all of its data into the system-wide cache.

By transforming slow disk faults into fast network faults, cooperative caching is just one approach for reducing disk stall time. Recent research has also focused on reducing disk stall time through two other approaches. One is the development of algorithms for prefetching data from disk into memory to avoid stalls [Cao et al. 96, Patterson et al. 95, Kimbrel et al. 96b, Tomkins et al. 97, Rochberg et al. 97]. These approaches use hints from programmer-annotated [Patterson et al. 95] or compiler-annotated [Mowry et al. 96] programs, speculative execution [Chang et al. 99], or markov prediction [Bartels et al. 99] to enable the system to predict future accesses and overlap disk I/O with execution. The second approach is the striping of files over multiple disks [Patterson et al. 88, Cao et al. 93], or the use of multiple networked nodes in distributed file systems [Hartman et al. 95, Anderson et al. 96, Thekkath et al. 97, Gibson et al. 97] and distributed storage systems [Lee et al. 96, Hartman et al. 99, Gibson et al. 99] to access disks in parallel, dramatically increasing I/O throughput.

To further improve the performance of cooperative caching systems, the second part of this dissertation proposes the synthesis of all three approaches into a novel *cooperative prefetching and caching* system — the use of network-wide memory to support prefetching, caching, and parallel I/O in the presence of optional hints of future demands. The synthesis of these approaches results in a system that is significantly different than one using any single approach alone. In the presence of global memory, a node has three choices for data prefetching: (1) from disk into local memory, (2) from disk into global memory (i.e., the disk and memory of *another* node), and (3) from

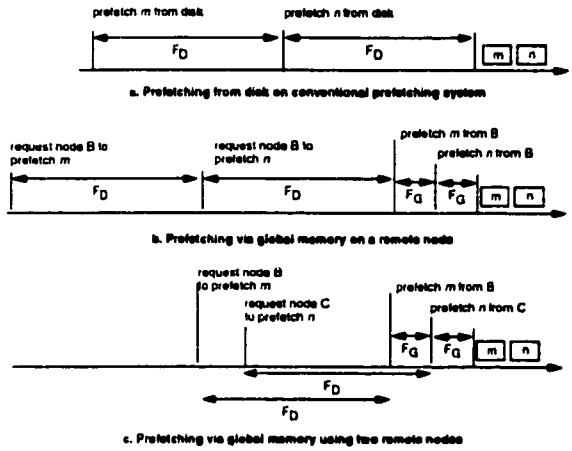


Figure 1.1: Prefetching in conventional and global-memory systems

global memory into local memory. When used together, these options greatly increase the power of prefetching relative to a conventional, non-cooperative-caching system.

For example, Figure 1.1a shows a simplified view of a conventional prefetching system. Node A issues prefetch requests to missing blocks m and n in advance, so that both blocks are available in memory just in time for the data references. In this case, buffers must be freed on node A for blocks m and n about $2F_D$ and F_D in advance of their use, respectively, where F_D is the disk fetch time. There are two possible problems with this scheme. First, node A's disk may not be free in time to prefetch these blocks without stalling. Second, if prefetched early enough to avoid stalling, blocks m and n may replace useful data, causing an increase in misses; whether or not this happens depends on how far in advance the data is prefetched (which depends on F_D) and the access pattern of the program.

In contrast, Figures 1.1b and 1.1c show two examples of prefetching in a global-memory system. From these scenarios, we see that combining prefetching and global memory has several possible advantages:

- A prefetching node can greatly delay its final load request for data that resides in global memory, thereby reducing the chance of replacing useful local data. In Figure 1.1b, for example, node A requests that node B prefetch pages from disk into B's memory ahead of time. As a

result, node A need not free a buffer for the prefetched data until F_G (the time for a page fetch from global memory) before its use. On a 1Gb/sec network, such as Myrinet, F_G may be up to 50 times smaller than F_D [Yocum et al. 97], so this difference is substantial.

- The I/O bandwidth available to a single node is ultimately limited by its I/O subsystem – in most cases, the disk subsystem. However, using idle nodes to prefetch data into global memory greatly increases the available I/O bandwidth by adding in parallel: (1) the bandwidth of the network, (2) the bandwidth of remote disk subsystems, and (3) the execution power of the remote CPUs. Use of this parallelism for the global prefetching shown in Figures 1.1b and 1.1c effectively reduces page prefetch time for I/O-bound processes from F_D to F_G.

- Figure 1.1c shows that distributing prefetch requests among multiple nodes in parallel allows those nodes to delay their own buffer replacement decisions (in this case, node B benefits relative to Figure 1.1b), thereby making more effective use of their memories.

- With the high ratio of disk latency to global memory latency, a highly-conservative process could choose to prefetch *only* into global memory; the process would fault on reference to a non-resident page, but would still benefit from the 50-fold reduction in fault time.

- Given that there is idle memory and CPU power in the network, a process could afford to prefetch *speculatively*, using old or unused global pages as the speculative prefetch cache.

While the idea of using network memory for prefetching is conceptually straightforward, it raises a number of questions. For example, how do nodes *globally* choose the pages in which to prefetch from the global memory pool? When should data be prefetched and to what level of the storage hierarchy? When should pages be moved from global memory to local memory? How do we trade off the use of global memory pages for prefetching versus the use of those frames to hold evicted VM and file pages for non-prefetching applications? And finally, how do we value each page in the network, in order to best utilize each page frame? Chapter 4 explores these questions in the design, implementation, and evaluation of a cooperative prefetching and caching system.

1.3 Cooperative caching at large scales in wide-area networks

Cooperative caching has been shown to improve the performance of file and virtual memory systems in a high-speed, local-area network environment. Unfortunately, the algorithms, mechanisms, and policies used in cooperative caching systems for workstation clusters do not scale beyond hundreds of nodes due to assumptions inherent in local-area network performance. For example, GMS uses periodic broadcasts to synchronize the global state of all nodes in the cluster [Feeley et al. 95], an operation not practical in wide-area systems. As a result, it is not known to what degree cooperative caching techniques might apply to large-scale distributed systems comprising millions of nodes deployed across wide-area networks. The final part of this dissertation explores this fundamental question of scaling cooperative caching to wide-area distributed systems. In particular, it examines the potential of using cooperative caching among World Wide Web proxies deployed in the Internet.

Internet proxy caching has become a commonplace approach for improving the performance of Web browsers. Typically, the proxy sits in front of an entire company or organization. By caching requests for a group of users, a proxy can quickly return documents previously accessed by other clients. Ultimately, though, the effectiveness of the proxy is a function of the size of the population it serves – a size often dictated by political, organizational, or geographic considerations. An obvious question, then, is whether multiple proxies should cooperate with each other in order to increase total client population, improve hit ratios, and reduce document-access latency.

Whether cooperative proxy caching is useful for improving performance depends on a number of issues. These include the client access patterns of Web documents and the cacheability properties of those documents, the performance of proxy caches in light of client access patterns and document characteristics, and the specific organization of proxy caches used to mitigate the long latencies and limited bandwidth of wide-area networks.

Web tracing and caching are highly active research areas. Recent studies of Web traffic include analyses of Web access traces from the perspective of browsers [Cunha et al. 95, Mah 97], proxies [Almeida et al. 96b, Breslau et al. 99, Caceres et al. 98, Cao 98, Crovella et al. 96, Douglis et al. 97, Duska et al. 97, Feldmann et al. 99, Gribble et al. 97, Kroeger et al. 97, Rabinovich et al. 98], and servers [Almeida et al. 96a, Arlitt et al. 96, Mogul 95]. Earlier tracing studies were limited in request rate, number of requests, and diversity of population; more recent tracing studies have been

larger in scale and more diverse in characteristics. In addition to static analysis, some studies have also used trace-driven cache simulation to characterize the locality and sharing properties of large traces [Almeida et al. 96b, Caceres et al. 98, Duska et al. 97, Feldmann et al. 99, Gribble et al. 97, Kroeger et al. 97], and to study the effects of cookies, aborted connections, and persistent connections on the performance of proxy caching [Caceres et al. 98, Feldmann et al. 99].

Researchers have studied the temporal locality of Web proxy traces and examined how hit-ratio depends, asymptotically, on cache size and the number of requests. Several interesting findings have been identified. First, it has been repeatedly found that, for most traces, the relative frequency with which Web pages are requested follows a Zipf-like distribution, where the number of requests to the i^{th} most popular document is proportional to $1/i^{\alpha}$ for some constant α [Almeida et al. 96b, Breslau et al. 99, Cao et al. 97, Cunha et al. 95, Glassman 94, Kroeger et al. 96]. Second, for infinite-sized caches, it has been shown empirically and analytically that the hit ratio for a Web proxy grows logarithmically with the client population of the proxy and the number of requests seen by the proxy [Breslau et al. 99, Cao et al. 97, Duska et al. 97, Gribble et al. 97, Kroeger et al. 96].

Finally, several cooperative-caching techniques have been proposed for use in the Web. These proposals include hierarchical schemes like Harvest and Squid [Chankhunthod et al. 96, Squid et], hash-based schemes [Karger et al. 99, Valloppillil et al. 98], directory-based schemes [Fan et al. 98, Menaud et al. 98, Tewari et al. 99] and multicast-based schemes [Michel et al. 98, Touch 98]. Although each of these research efforts includes a performance evaluation of the protocols proposed and a discussion of algorithm scalability, only [Krishnan et al. 98] presents empirical evaluations of cooperation (and only for small populations), and none present empirical or analytical evaluations of the effectiveness of their schemes for the large client populations found in a wide-area setting.

As a result, several questions remain unanswered regarding the potential performance benefits of cooperative caching in large-scale Web environments. For what range of client populations can cooperative Web caching perform effectively in terms of hit rate, latency reduction, and bandwidth utilization? Previous studies have just looked at performance from the basis of a fixed number of requests. What is the effect on the performance of cooperative caching if the systems were running for a month, a year, five years? In other words, how will cooperative caching schemes perform in steady-state operation over long-term time scales? Furthermore, many characteristics of the Web change constantly, such as client request rate, the rate of change of documents, and the size of the

Web (in terms of clients, servers, and documents). How do the results of cooperative caching apply in the face of future trends of Web characteristics? Lastly, various cooperative caching architectures have been proposed, but not compared at large-scales. What arrangement of cooperating caches provides the best performance, and/or the most efficient use of storage resources? Chapter 5 explores these questions using analytic modeling to examine steady-state cooperative-caching performance with large client populations in wider-area environments.

1.4 Contributions

The contributions of this thesis are to demonstrate (1) that cooperative caching systems are considerably more effective when combined with memory server CPU load balancing and disk and network prefetching, and (2) that cooperative caching of World Wide Web documents among proxy caches in wide-area networks is effective only within limited population bounds. Specifically:

1. I explore the tradeoffs of using memory server CPU load information, in addition to age information, in cooperative caching replacement policies. I first develop an analytic queueing network model to study the extent to which server load can degrade remote memory latency, and demonstrate that load balancing minimizes this degradation. I then use trace-driven simulation to study the impact on application performance of deviating from a global LRU replacement policy, and demonstrate that minor deviations do not affect application performance. Finally, based upon these analyses, I propose a new family of algorithms that incorporate load considerations, as well as age information, in global memory replacement decisions.

   The load balancing results are applicable to global memory systems that implement approximate global LRU replacement using varying types of global information. Furthermore, the general insights about the benefit of balancing load at minimal costs due to less perfect replacement decisions, as well as the new family of algorithms suggested by these insights, can also be applied to global memory systems that use replacement algorithms other than LRU.

2. I further extend workstation cluster cooperative caching systems to support disk and network prefetching, exploiting high-speed local-area networks to efficiently manage the mem-

ory, disk, and CPU resources in the cluster:

- I introduce an operating system prefetching module that provides three different prefetching mechanisms in the cluster: (1) prefetching from local disk directly into local memory (conventional file prefetching); (2) prefetching a page from the disk and into the memory of a remote node; and (3) prefetching a page over the network from a remote node's memory into local memory.
- Using these prefetching mechanisms, I introduce a combined prefetching and caching algorithm for managing cluster disk resources in conjunction with cluster memory resources. The algorithm uses hints of application reference patterns to make prefetching and cache replacement decisions. A novel aspect of the prefetching algorithm is that it is a hybrid algorithm, where prefetching into local memory and into global memory uses different policies. Local memory is more valuable than global memory as a cache because local pages are much more likely to be referenced than global pages. Reflecting its relative values as a cache, prefetching into local memory is *conservative* to maintain as large of a local cache as possible. In contrast, prefetching into global memory is *aggressive* since using global memory for prefetching is more valuable than using it as a cache. The goal is to use idle disk resources on behalf of applications, but not to be so aggressive as to degrade caching performance.
- I present an implementation of a combined prefetching and caching system in a prototype called the Prefetching Global Memory System (PGMS). I implemented PGMS in the Digital Unix operating system as an extension to Feeley's Global Memory System (GMS) [Feeley et al. 95]. The prototype extends GMS to support the three prefetching mechanisms, a system call interface for passing application reference hints to the kernel, and a prefetch module for tracking an application's page references and scheduling prefetch requests on its behalf.
- I evaluate the PGMS prototype on a cluster of Digital Alpha workstations using a set of real applications and synthetic benchmarks. With this evaluation, I demonstrate that combining prefetching with cooperative caching in a globally managed cluster can greatly improve the performance of I/O bound applications. In particular, I show that:

(1) the use of prefetching significantly increases application performance beyond the use of cooperative caching alone; (2) global prefetching provides substantial speedup over local, conventional prefetching alone; (3) parallel disk prefetching leverages the aggregate disk I/O bandwidth in the cluster, scaling application performance with cluster size; (4) prefetching over the network from global to local memory has much less impact than disk prefetching; and (5) the system isolates prefetching and non-prefetching applications, allocating system resources among them fairly.

3. I introduce a novel analytic model of Web accesses to explore the effectiveness of cooperative caching among World Wide Web proxies. This model evaluates the steady-state performance of proxy caches based upon client request, sharing, and document rate of change distributions for small-scale organizational populations to large-scale regional populations.

   With the model, I demonstrate that cooperate Web proxy caching is effective at organizational scales, but it is not necessary; a monolithic proxy cache provides performance at least as good, but with less complexity. I further show that cooperation among the organizational caches within a medium to large city will provide additional performance benefit, although only marginal, over cooperative caching at small scales. Assuming that bandwidth within a city is plentiful and latencies are small, the overhead of cooperative caching would be low and therefore worth the secondary benefits that such caching provides. However, extrapolating to larger scales, cooperative caching provides limited or no additional benefit, particularly in the face of increased latencies among caches.

1.5 Overview of the Dissertation

The rest of this dissertation is organized as follows:

- Chapter 2 provides background information on cooperative caching systems for workstation clusters and World Wide Web proxies, and presents related work in this area.

- Chapter 3 considers the potential benefit and the potential harm of incorporating memory server CPU load information in cooperative caching replacement policies. It concludes with

a proposal for a new family of algorithms that combine both load and age information in replacement decisions.

- Chapter 4 presents a combined cooperative prefetching and caching system for workstation clusters that unifies previous work in cooperative caching, prefetching, and parallel I/O. It concludes with the evaluation of the Prefetching Global Memory System, a prototype implementation of a combined cooperative prefetching and caching system.

- Chapter 5 considers the issue of scaling cooperative caching to wide-area distributed systems by examining the potential of using cooperative caching among World Wide Web proxies deployed in the Internet.

- Finally, Chapter 6 summarizes and concludes the dissertation.

Chapter 2

Background and Related Work

This dissertation extends cooperating caching systems to (1) manage additional hardware resources in workstation clusters to improve performance, and (2) explore the potential of cooperative caching for large-scale wide-area distributed systems. This chapter gives an overview of cooperative caching systems and background information for the techniques used to extend them in this dissertation. Section 2.1 surveys cooperating caching systems designed and implemented for workstation clusters; it presents the fundamental background material necessary for understanding the structure and behavior of cooperative caching systems. Section 2.2 discusses previous efforts to combine caching and load balancing in cluster servers; it serves as background for Chapter 3, which applies load balancing to cooperative caching systems. Section 2.3 describes the development of techniques for combining caching and prefetching for use in centralized systems; it provides the background for Chapter 4, which applies prefetching techniques to a cooperative caching system. Finally, Section 2.4 surveys previous cooperative caching systems designed for the World Wide Web, which have been evaluated only at small scales; it provides the background and motivation for Chapter 5, which studies the potential of using cooperative caching techniques for the Web at very large scales.

2.1 Cooperative caching in workstation clusters

Cooperative caching in workstation clusters globally manages the memory resources of workstation nodes to improve application performance. Typically, a node caches file and virtual memory data only on behalf of applications running on that node. With cooperative caching, nodes can also contribute a portion of their memory for use as part of a global cache distributed among all of the nodes in the cluster.

Cooperative caching improves performance in two ways. First, it can coordinate the contents of the caches across all nodes in the cluster. This coordination can exploit temporal sharing among

applications as they access the same files (e.g., header files when compiling) or virtual memory pages (e.g., executable code pages). Instead of duplicating shared pages among individual caches, the nodes can coordinate their caches to maintain fewer copies and increase overall cache efficiency. Second, cooperative caching can exploit idle memory resources as backing store for data on active nodes. If a node is idle or lightly-loaded, its memory or a portion thereof can be used to cache data evicted from active nodes.

By globally coordinating the contents of node caches and using idle memory as backing store, cooperative caching converts what otherwise would be disk accesses into network accesses. With the advent of low-latency, high-bandwidth, switched local-area networks like ATM, Myrinet, and Fast and Gigabit Ethernet, accessing data over the network can be up to 50 times faster than accessing it from local disk. By converting disk accesses into much faster network accesses, cooperative caching significantly decreases average fault time, thereby increasing application performance.

The design space of cooperative caching is extensive, and systems vary depending upon (1) the kind of data stored in the cache, (2) how nodes partition their memory between local and global use, (3) the degree to which nodes coordinate their caches, (4) the replacement policy used for managing the cooperative cache, (5) the approaches for tolerating faults, (6) the granularity at which data is cached, and (7) the degree of integration with the operating system or applications. The following section summarizes early cooperative caching systems, and the subsequent sections describe recent systems and their contributions in detail.

2.1.1 Early work

Early work in cooperative caching primarily used remote memory as backing store for evicted virtual memory pages. These systems had little global sharing of cache contents among clients, and no global coordination of cache resources. The Apollo DOMAIN system implemented a distributed single level store (SLS) that coordinated concurrent caching of object pages [Leach et al. 83]. Client caches were used to operate on object data backed by persistent storage on local disk or the disk of a remote node. Client caches cooperated to implement concurrency control and detect concurrency violations. Caches did not explicitly share data (cache misses always went to the home node storing the object), nor did they globally coordinate cache resources. [Comer et al. 90] propose the use

of dedicated remote memory servers for use by diskless clients. In the diskless environment clients normally page to a remote disk, but heavily paging clients could allocate memory from these servers and use them as backing store instead. Pages could potentially be shared among clients through the servers. [Felten et al. 91] generalized this idea to use the memory on idle workstations as backing store. Nodes register with a central manager when they become idle, and active nodes randomly choose among them to receive evicted pages. Clients do not share their remote backing store resources. [Schilit et al. 91] use remote paging in mobile environments. Memory-starved portables can page to the large memories of nearby stationary servers. These evicted pages can migrate among servers as the portable moves.

[Franklin et al. 92] is the first system to incorporate global sharing of cache resources. They propose the use of remote memory in a client-server database management system (DBMS). The centralized DBMS server uses the client caches as an extension of its own by keeping track of the contents of all client caches. When a client requests a page not in the server cache, the server determines whether another client is caching the page. If so, the server has the caching client forward the page back to requesting client. Although the server and clients share the contents of their caches, they do not coordinate their contents, e.g., to eliminate duplicates.

2.1.2 xFS

The xFS scalable network file system uses cooperative caching to coordinate client and server file caches [Dahlin et al. 94, Anderson et al. 96]. Clients can contribute a portion of their memory for use as part of a global cache on behalf of all clients in the system. Idle clients, for example, can store file blocks in their memories that have been evicted from the server or other clients. And with coordination, clients can share the contents of their caches to reduce duplication and increase overall cache efficiency.

[Dahlin et al. 94] is an interesting exploration of the design space for cooperative caching algorithms from the perspective of file system caching. They describe a taxonomy of six algorithms. The algorithms differ as to whether clients share cache contents, how they coordinate their caches, and how they partition memory between local and global use.

With *direct client cooperation*, active clients use the memory of idle clients as a private back-

ing store for pages evicted from local memory; this approach approximates the model used by [Comer et al. 90, Felten et al. 91, Schilit et al. 91]. Direct client cooperation is simple because it requires no modifications to the server, although it does require mechanisms for identifying idle nodes and coordinating their use by active nodes. A significant drawback is that clients cannot share the contents of their caches: a request by one client must go to disk if the server does not have the block cached, even if another client is caching the block in either its local cache or its private global cache on an idle node.

With *greedy forwarding*, the memories of all clients are used as a single global cache, but the clients do not coordinate the contents of these caches; this approach models the one used by [Franklin et al. 92]. The server maintains a data structure describing the contents of all client caches. If a client cannot satisfy a request in its local cache, it forwards the request to the server. If the server does not have the requested block cached, it forwards the request to any client caching the block. This approach is greedy because each client manages its local cache without regard to the contents of other caches or the needs of other clients in the system. For example, the same block may be cached on multiple clients, a potentially inefficient use of the memory resources.

Centrally coordinated caching extends greedy forwarding to coordinate the contents of the global cache. The cache on each client is divided into local and global portions. The client directly manages the local portion for its sake alone, and the server centrally manages the global portion of all client caches for the sake of all clients. A client forwards a request that misses in its local cache to the server. If the requested block is not cached by the server, but it is cached by a client in the global portion of its cache, then the server forwards the request to that client. The server manages the global cache using a global replacement algorithm such as LRU. When the server evicts a block from its cache, it replaces the least recently used block among all blocks in the global portion of the client caches. A drawback of this approach is that it places a high load on the server, and the partition between local and global portions of a client's cache is static.

N-chance forwarding extends greedy forwarding to dynamically adjust the boundary between the local and global portions of a client cache. N-chance forwarding behaves like greedy forwarding except for the handling of *singlets*, blocks stored in only one client cache. When a client evicts a block from its local cache, it checks block metadata or asks the server if the block is not cached on any other client. If it is cached elsewhere, the client discards the block. If it is the last copy, the

client sets a *recirculation count* and randomly forwards the block to another client. The receiving client adds the block to its local LRU list as if it had just been accessed. If the receiving client evicts the block, it decrements the recirculation count and repeats the forwarding algorithm. Once the recirculation count reaches zero, the block is discarded. N-chance provides a dynamic tradeoff between caching data for local benefit and caching data for the benefit of the system. Active clients will tend to displace global data with local data, and idle clients will accumulate global data on behalf of active clients.

Hash-distributed caching extends centrally coordinated caching by partitioning the management of the global cache among the clients rather than the server. This partitioning is done by hashing block identifiers to clients. The server evicts blocks from its cache to the appropriate client based upon the hash, and clients forward requests that miss in their local cache to the client responsible for the appropriate portion of the global cache.

Weighted LRU is similar to N-chance in that it dynamically manages all client caches to balance between local and global use. When weighted LRU makes a replacement, it attempts to replace a block with the best cost-benefit ratio. Duplicates provide local access, but they consume local cache space, and, if replaced, can always be accessed in another cache over the network. However, singlets require a disk access if replaced. Weighted LRU balances the benefits of keeping duplicates against the cost of evicting singlets to optimally manage the client caches. This approach is used by the GMS system [Feeley et al. 95], discussed next.

Dahlin *et al.* simulated the different algorithms using distributed file system traces as input. They found that (1) N-chance had the lowest average request latency and server load, (2) direct client cooperation had the highest average request latency, and (3) centrally coordinated caching had the highest server load. Hash-distributed caching significantly reduced the server load of centrally coordinated caching, but had no effect on response time. Response time for Weighted LRU was slightly worse than N-chance, although it is not clear why (the next section addresses this issue in more detail). They later implemented N-chance as the cooperative caching algorithm for the xFS prototype [Anderson et al. 96].

2.1.3 Global Memory Service

The Global Memory Service (GMS) uses cooperative caching to globally manage the memories of all workstations in a local-area network [Feeley et al. 95]. The goal of GMS is to minimize the average page access time across all applications in the system. GMS implements cooperative caching at the lowest level of the operating system, making remote memory available for virtual memory backing store, mapped files, and the file buffer cache. The use of cooperative caching is transparent to applications and is managed entirely by the operating system.

Each workstation can contain both *local* pages and *global* pages. A local page is one that contains data for an application executing on the same machine; a global page is one that contains data for an application executing on another machine. The balance between local and global pages on a node changes over time, depending on the behavior of that node and others. When a node becomes idle, its pages will age, and eventually will be replaced by younger global pages displaced from other nodes. When an idle node begins computing again, it will increase its use of local memory, forwarding or discarding global pages as it does so. Eventually, if the applications on the node have high memory demands, the node will fill its memory with local pages and will begin to use global memory on other nodes as it replaces its oldest local pages.

The global memory across the nodes in the cluster forms a shared cache between local memory and disk. Coherency is maintained by flushing dirty local pages to disk before placing them in global memory at another node. Pages in global memory can then be shared among nodes without any locking. In practice, file buffer cache pages and the code pages of an application are shared by multiple instances of the application executing on different nodes, while the data pages are private.

In the xFS taxonomy, GMS essentially implements *weighted LRU*: a globally coordinated, dynamically partitioned cooperative cache with a global LRU replacement policy. GMS approximates global LRU across all of the nodes in the cluster in fixed periods of time called *epochs*. At the start of each epoch, all nodes send the ages of all global pages to a coordinator node. The coordinator examines the page ages, and broadcasts the distribution of all old pages in the cluster to each node. When a node has to evict a page during the epoch, it randomly chooses a destination node with old global pages weighted by the fraction of old pages it contains. In contrast to N-chance, where clients autonomously cooperate by forwarding evicted pages to other clients randomly, the nodes in

GMS globally coordinate by evicting the least-valuable pages across the entire cluster.

Feeley *et al.* implemented GMS in the Digital Unix operating system kernel and found that it achieved excellent speedups for a variety of applications. They also studied the impact of global coordination by implementing the xFS N-chance algorithm and comparing its performance to the GMS approximate global LRU algorithm. They found that, when idle memory is unevenly distributed across nodes in the cluster, the global coordination of GMS performed significantly better than the autonomous coordination of N-chance. N-chance forwards evictions at random, a policy that works well for a uniform distribution of idle memory but is suboptimal when the distribution is uneven. Furthermore, N-chance favors singlets over duplicates, even when duplicates have been recently referenced. When there is a high degree of sharing among applications in the cluster, N-chance will evict duplicates in favor of singlets even when the duplicates have greater value to the system overall.

2.1.4 Remote Memory Pager

[Markatos et al. 96] use cooperative caching as a reliable Remote Memory Pager (RMP). In the xFS taxonomy, RMP uses direct client cooperation. Idle remote memory in workstation clusters is used as a backing store for application virtual memory pages. The RMP is implemented as a client that issues page read and write requests and a server that satisfies these requests. The client is a kernel block device driver that transparently handles pagein and pageout requests. The server is a user-level daemon that stores pages in its memory, and handles read and write requests to that memory from clients. Additional work explored variations on the basic pager (1) to support remote memory access at the granularity of individual words [Markatos 96], (2) as a log for transaction support for recoverable virtual memory [Ioannidis et al. 98], and (3) as the underlying system for a network RAM disk [Flouris et al. 98]. This discussion focuses on the design and implementation of the basic remote paging system.

Whereas xFS and GMS focused on designing efficient, near-optimal distributed memory management algorithms, global memory management in the RMP is relatively unsophisticated. Since the remote memory client is a block device driver, it does not participate in local memory management and uses the default kernel policy. Clients allocate from servers in a round-robin fashion.

No global replacement algorithm is used; once clients allocate memory on a server, that memory is never paged out or reused until explicitly deallocated by the client. Once all server memory is allocated, clients page to local disk.

The novel aspect of the RMP is its support of *reliable* remote memory. To tolerate faults, xFS and GMS write dirty pages to local disk before paging them out into network memory. As a result, network memory only contains clean pages. If a memory server or the network fails, preventing clients from accessing pages in network memory, clients can always recover their pages from local disk. In these systems, the local disk can bottleneck performance if the rate at which the system evicts dirty pages is greater than the disk write bandwidth.

Since network bandwidth is often greater than local disk bandwidth in high-speed local-area networks, the RMP explores techniques for making network memory reliable so that it can store dirty pages in memory. Markatos *et al.* proposed three techniques to make network memory reliable to single node failures: mirroring, parity, and parity logging. The techniques have different tradeoffs in storage overhead, number of additional page transfers, and increased client CPU overhead required to provide reliability. Note that these techniques are still susceptible to network failures.

In *mirroring*, pages are evicted to two different servers. If one server fails, clients simply use the other. Mirroring requires significant space overhead, however, since half of remote memory is used to store redundant pages. Furthermore, each eviction requires two network page transfers from the client, one to each mirror.

The *parity* technique is similar to the one used in RAID disk arrays [Patterson et al. 88]. Memory servers are partitioned into parity groups of size N, and pages across servers are partition into parity stripes. Logically, in each group one server is used to store parity pages and the remaining servers store memory pages. When a client evicts a page, it sends the page to a memory server. The server XORs the page with the previous version, if present. The result is then sent to the parity server, which XORs it with the parity page to form the new parity. Compared to mirroring, the parity approach requires much less space overhead ($1/N$) but a similar number of page transfers (although only half of them are from clients).

The *parity logging* technique tries to achieve the low overhead of the parity approach while reducing the number of page transfers. In parity logging, clients construct parity pages. For each batch of $N - 1$ page evictions, the client computes a parity page and sends it to the parity server.

Once a page is written into a new stripe, the old version already in network memory is marked invalid. A cleaner is used to garbage collect invalid pages in parity stripes to free up network memory. To reduce the frequency of running the cleaner, memory servers reserve scratch space to handle the overhead of storing redundant copies of the same page in different stripes. Parity logging requires $1/N$ plus scratch space (10% in the prototype) overhead, only $1 + 1/N$ network page transfers, but additional client CPU overhead to compute the parity on the client.

Markatos *et al.* evaluated the different techniques using DEC Alpha workstations connected by fast Ethernet[Markatos et al. 96]. He found that parity logging performed the best and mirroring the worst. The space overhead of mirroring induced too many extra faults, but parity logging required fewer network transfers than plain parity even with the additional scratch space overhead. However, the cleaner was never used, so its impact on parity logging performance is unknown. Furthermore, they did not measure the cost of recovering from failure or estimate the frequency of failures, so it is unclear whether the cost of recomputing parity offsets the benefit of parity logging.

They also compared parity logging to writing dirty pages to local disk, and found that using the local disk performed better than parity logging. In their system, local disk performance was comparable to Ethernet performance, so this result is not surprising. As a result, it is still unknown whether local disk write bandwidth is a bottleneck for these systems when using much faster local-area networks. Lacking measures of recovery performance, it is also not known how recovery performance of RMP compares to that of local disk.

2.1.5 Dodo

Dodo is another cooperative caching system that uses idle memory resources in workstation clusters as a remote pager and, in the xFS taxonomy, implements a direct cooperative cache [Koussih et al. 99]. It was designed in the spirit of Condor, a system for exploiting idle CPU resources in local-area networks [Litzkow et al. 88]. As with Condor, Dodo emphasizes portability so that it can be used on multiple operating systems and architectures. In contrast to GMS and xFS, Dodo sacrifices close integration with the operating system for the portability of a user-level system. It is designed as a set of user-level daemons for detecting idleness on machines, coordinating the use of idle memory, and storing remote data. It can use UDP or U-Net [vonEicken et al. 95], if available, to transfer data.

The Dodo prototype has been implemented to support the use of remote memory by only a single application at any one time, and its design reflects this orientation.

Dodo manages data using *memory regions* of arbitrary size, in contrast to previous systems that manage data at the granularity of pages. Furthermore, applications must explicitly use memory regions to gain the performance benefits of using idle memory on remote nodes, and Dodo provides applications with a library for opening, reading, writing, and closing memory regions.

When an application wants to make data cacheable in remote memory, it "opens" a remote memory region. Once a memory region has been opened, the application can issue read and write requests to it. Requests to regions cached locally are handled normally. On a read request to a memory region stored in the idle memory of a remote node, Dodo will first evict regions out of the local cache and then transfer the required region from the remote node into the local cache. On a write request to a remote region, Dodo will propagate the write to the remote node while simultaneously writing the data to the backing store on local disk (*no write allocate*); for correctness, it only stores read-only data on idle nodes.

Memory regions can be stored on only one idle node. Once a memory region has been allocated and assigned to an idle node, the region will only be faulted back and forth between the active node and that particular idle node. Although an application can read and write portions of a memory region, Dodo transfers memory regions between local and remote memory in their entirety. The local memory region replacement policy in Dodo is configurable and extensible. Applications can implement their own policy, or use a Dodo library that implements two default policies: LRU and *first-in*, a policy where once a region has been cached in local memory it is never replaced. Because Dodo does not allow a memory region to be allocated if a remote node does not have enough idle memory to back it up, it does not need or use a global replacement policy.

Idle memory allocation

Dodo will only allocate a memory region if there is an idle node in the cluster that has enough idle memory to store it. As a result, Dodo fundamentally couples the *use* of idle memory with the *allocation* of idle memory. Unfortunately, the coupling of use and allocation imposes two constraints on applications.

First, Dodo places the burden of adapting to the availability of idle memory on application developers, rather than the system. Since the amount of idle memory in the cluster changes over time, applications have to be written to explicitly handle the situation where there is not enough idle memory in the cluster to satisfy their allocations. Otherwise, an application might be able to run when N nodes are idle, but not when $N - 1$ nodes are idle.

Second, Dodo does not provide any fairness properties for the use of idle memory. This lack of fairness reflects the implicit assumption that only one application will use idle memory at any given time; if only one application uses idle memory, then it will only be competing with itself.

Regions vs. pages

In Dodo data is managed at the granularity of regions, rather than pages as in the previous systems. The use of regions enables applications to allocate data in arbitrary sizes. However, the design of memory regions and its programming interface compels programmers to allocate and use large memory regions. For example, the Dodo API makes it easy to associate a file with a memory region, but associating each file block with its own memory region requires management overhead on behalf of the application. Unfortunately, the use of large regions can also limit the performance of the system.

Dodo will only store a memory region on one node, and large memory regions cannot be split among multiple nodes. Even if there is enough idle memory in the cluster to store a memory region, an application cannot allocate it unless all of that memory is on one node. Furthermore, with the use of large regions, the opportunity for pipelining and overlapping application execution with data transfer is greatly diminished [Jamrozik et al. 96].

2.1.6 Summary

Early client-server systems looked at the benefit of cooperatively managing the caches between clients and servers to make more effective use of their memory resources.

The xFS studies investigated various degrees of cooperation and coordination among file system clients and servers, and showed that it is important for cooperative caching systems (1) to coordinate cache contents, and (2) to dynamically adjust memory allocation between local and global cache

use. The Global Memory System validated these results for a broader class of applications, and further showed that it is important for the caches to coordinate globally rather than autonomously. The Remote Memory Pager explored techniques for making the cooperative cache reliable so that it can store dirty pages. Dodo also explored the use of cooperative caching to implement a network memory pager, but it focused on the development of a portable user-level implementation, support for variable-sized paging granularities, and an efficient and effective distributed idle node detector.

2.2 Load Balancing

Cooperative caching systems manage the memory resources in clusters to maximize the locality of requests to the data stored on the nodes in the cluster. In maximizing locality, though, the distribution of future accesses to that data can become highly skewed. This skew creates a load imbalance among the nodes caching data, and this load can greatly increase the latency to access that data. To prevent this situation, the first part of this dissertation extends cooperative caching to manage the CPU resources of the nodes in the cluster by judiciously trading off locality to balance load.

In cooperative caching systems, balancing load fundamentally depends upon the *placement* of globally cached *data* among the nodes in the cluster. Load balancing in cooperative caching systems differs from the traditional process load balancing problem in three fundamental ways. First, traditional load balancing techniques balance load by distributing jobs among the nodes in the network, whereas cooperative caching systems balance load by distributing data. Second, traditional techniques are able to manage load directly since jobs directly consume CPU resources. Cooperative caching systems, however, can only manage load indirectly since the load associated with the placement of data depends upon future accesses to that data. Third, cooperative caching systems must first give priority to locality, and only load balance when locality results in imbalanced load.

Cluster nodes incur load from two operations. The first is when they receive evicted data. In this operation, the system has complete control over the distribution of this load across the nodes because they can essentially choose any node to receive the data. The second is when the node must send cached data to another node. In this case, the system has limited control over balancing these operations since it is the access behavior of applications that determines the data that is accessed, and hence the node to which requests are sent for that data.

Cooperative caching systems place data to maximize locality, and it is by maximizing locality that they achieve most of their performance benefits. It is only when data placement induces a highly skewed distribution of requests to memory servers that load impacts the performance of the system. So load balancing in cooperative caching systems should prioritize locality, but do so in a way that makes it straightforward to curtail locality when load becomes an issue.

For background and motivation in adding load balancing to cooperative caching system, the remainder of this section contrasts traditional load balancing with load balancing in cooperative caching systems, and surveys two systems that combine caching with load balancing in clusters.

2.2.1 Traditional load balancing

The traditional load balancing problem in distributed systems is to balance jobs across a set of distributed servers to maximize system throughput and minimize delay. This problem occurs for a wide range of applications, from balancing traditional Unix processes among nodes in a cluster of workstations to balancing HTTP requests among a set of widely distributed Web servers. Traditional load balancing algorithms must address three core issues: how to assign jobs to servers, how to select and migrate jobs during execution, and how to collect and incorporate timely load information from servers into the load balancing algorithm.

How should jobs be assigned to servers? Without knowing job lifetimes in advance, a scheduler that does load balancing must estimate their lifetimes and schedule the jobs appropriately according to the current load on the servers. It is well known that the naive approach of assigning the next job to the least-loaded server can lead to a herding effect that results in bad performance (e.g., [Dahlin 99]). When jobs are strictly assigned to the least-loaded server, that server quickly becomes overloaded as all nodes independently send their jobs to the same server.

[Eager et al. 86a] shows that a load balancing algorithm can be effective at assigning new jobs to servers using only a small amount of load information. The algorithm of Eager *et al.* randomly samples the load on a few servers, and chooses the least-loaded of the sample. Because the sample is randomly chosen, it avoids the herding effect. Because it still chooses the least-loaded server of the sample, it is still able to achieve good system performance. Unfortunately, load balancing placement algorithms like [Eager et al. 86a] are not directly applicable to cooperative caching systems since

the future load impact of assigning data to servers is not necessarily known at the time of placement, and cooperative caching systems must first take into account cache locality as well as future load impact.

Should already executing jobs be migrated from a loaded server to an unloaded server? An early study by [Eager et al. 86b] argued that migrating jobs provides only limited performance benefits and therefore was not worth doing. A more recent study by [Harchol-Balter et al. 97] challenged this result. Harchol-Balter *et al.* showed that the issue of whether jobs should be migrated fundamentally depends upon job lifetime distributions. They further showed that observed Unix process lifetime distributions differed significantly from the distribution assumed by [Eager et al. 86b], and that the observed distributions motivated dynamically migrating jobs would noticeably increase system performance. The corresponding issue for cooperative caching systems is whether data on loaded memory servers should be shuffled to unloaded servers to balance future load. And the corresponding job lifetime distributions for cooperative caching systems is the distribution of future accesses to the pages on memory servers. This distribution is determined by the placement algorithm of the cooperative caching system and the reference behavior of the application.

How does the timeliness of server load information affect job placement and migration? In distributed systems, it is impossible for the system to know the instantaneous load on all servers simultaneously. As a result, any information that the system has regarding server load is going to be stale to some extent, and the degree of staleness can affect the ability of the system scheduler to balance load. [Mitzenmacher 97] shows that even very stale server load information is valuable for balancing load in distributed systems as long as servers are chosen randomly from a small sample. [Dahlin 99] generalizes the results of [Mitzenmacher 97] and proposes a family of Load Interpretation algorithms that adapts to the degree of staleness of load information. These algorithms choose servers as a function server load, the age of the load information, and an estimate of the rate at which new jobs enter the system. When the load information is fresh, the algorithms are able to load balance aggressively. But when the load information is old, the algorithms can significantly outperform random algorithms that choose only among a small sample of servers.

For cooperative caching systems, it is relatively inexpensive to obtain up-to-date load information for the memory servers. First, cooperative caching systems require nodes to be connected by high bandwidth, low latency networks, so the time to transmit load information is quite small. Sec-

ond, cooperative caching systems already distribute page age information among the nodes in the system, and server load information can be combined and distributed with this page age information. Third, active nodes are in frequent communication with the idle nodes in the system as pages are transmitted back and forth, and load information can be easily piggy-backed on these page requests to keep nodes up-to-date on server load. As a result, cooperative caching systems can use load information to aggressively load balance as advocated by [Dahlin 99].

2.2.2 Scalable network services

[Fox et al. 97] propose an architecture for building cluster-based scalable network services that they successfully used to implement a transformation and caching proxy for a modem pool and a Web search engine. A key component of their architecture includes *workers*, such as transformation engines, search engines, and caches. The cluster system implements a centralized load balance policy to prevent workers from becoming overloaded. Workers periodically send load hints to a central manager. These hints are used by the front end to distribute requests across the workers, and by the manager to spawn additional workers on unused nodes or take them down as load fluctuates.

When this architecture is used for caching, data is partitioned among workers by hashing on the name space. When workers are added or removed due to load, the name space is automatically rehashed. This rehashing will produce a short-term drop in locality, but meets the needs of the system since it considers caching a soft-state optimization that can help if present but is not necessary to function.

2.2.3 Locality-aware request distribution

The most closely related work that combines caching with load balancing in clusters is *locality-aware request distribution* (LARD) [Pai et al. 98]. The system they study consists of a front-end node that receives requests from clients and forwards the requests to back-end nodes to serve the requests. The front-end node chooses a back-end node based upon (1) the object requested and (2) the load on the back-end nodes.

This approach is locality-aware because back-end nodes cache responses to requests, so forwarding requests to the same object takes advantage of locality in the request stream. And it balances

load by preventing high degrees of locality (hot spots) from overwhelming a single back-end node, dynamically distributing it across load multiple servers. The particular application they study is cluster-based Web service where the front-end node receives HTTP requests and forwards them to back-end Web servers.

Pai *et al.* studied six policies for combining locality and load information:

- *Weighted round-robin* (WRR) distributes requests in round-robin fashion across the back-end nodes weighted by a load metric (CPU and/or disk utilization, number of open connections, etc.); this policy maximizes a balanced load while ignoring locality.

- *Weighted round-robin/GMS* (WRR/GMS) is an extension of WRR that uses an approximation of the global memory management algorithm of GMS on just the back-end nodes to share cache contents.

- *Locality-based* (LB) statically partitions requests across back-end servers, e.g., by hashing request URLs; this policy emphasizes locality while ignoring load.

- *Locality-based/global cache* (LB/GC) simulates an ideal global cache by distributing requests to the back-end nodes that cache the object, and to the back-end node with the best replacement opportunity if the object is not cached; this policy maximizes locality and global cache replacement, but also ignores load.

- *Locality-aware request distribution* (LARD) distributes requests to back-end nodes according to a locality/load balance tradeoff. The front-end node maintains assignments between objects and back-end nodes for locality. The first time the front-end sees a request to a object, it assigns and forwards it to the least loaded node. On subsequent requests to the object, the front-end simply forwards the request to its assigned back-end. However, if the load on the assigned back-end node is above a threshold, and there exists a relatively unloaded back-end node, the front-end node reassigns and forwards the request to the least-loaded node instead.

- *Locality-aware request distribution/replication* (LARD/R) extends LARD to maintain a set assigned back-end nodes for a object, and the front-end forwards requests to the least loaded

back-end in the set. However, if the least-loaded back-end has high load and there exist unloaded nodes, LARD/R adds the least loaded load to set of assigned nodes for the object. The set assignments are dynamic, growing as needed to handle load and shrinking them when the assignments are stable for a period of time.

Pai *et al.* evaluated the different policies using trace-based simulation of Web server workloads. WRR balanced the load the best, but had the lowest throughput and hit rates because it ignores locality. The LB policies had the best hit rates, but only moderately better throughput than the WRR policies because they do not balance load. By first emphasizing locality and then balancing load only during overload, the LARD policies achieve the highest throughputs and balance load nearly as well as WRR.

The analyses by Pai show that cooperative caching achieves excellent locality, but when requests are not well distributed across servers, traffic can overload servers. In this situation, it is better to favor load balancing over locality.

2.3 Prefetching

The second part of this dissertation extends cooperative caching systems to use idle disks to improve their performance. Although cooperative caching can help reduce the number of disk faults, applications can still spend a significant amount of time loading data from disk in the first place. For example, one application executing on GMS achieved a speedup of 1.7 once the system had enough idle nodes to contain its working set. However, it still spent 80% of its execution time stalling on disk I/Os as it loaded its working set from disk into memory. For this application, adding more nodes to the system would not improve its performance because it was limited by the I/O time required to load all of its data into the system-wide cache.

An effective method for using disks on behalf of caching systems is to use them to *prefetch* data into the cache. The first half of this section surveys previous work in combining caching and prefetching techniques into a single system as background for applying those techniques to cooperative caching systems.

A combined cooperative prefetching and caching system also needs a storage system that is able to place application data so that it can be accessed from anywhere in the cluster, and accessed such

that the degree of parallelism and bandwidth scales with the number of nodes in the cluster. The second half of this section describes various scalable distributed storage and file systems that can be used by a cooperative prefetching and caching system to scale performance with the amount of disk resources in the cluster.

2.3.1 Combined prefetching and caching

A variety of techniques have been developed to combine caching and prefetching, and this section surveys them in order of increasing complexity:

- *Single disk, single process.* A single process manages its memory both to cache data and prefetch from a single disk.

- *Multiple disk, single process.* A single process manages its memory both to cache data and prefetch from multiple disks in parallel.

- *Multiple disks, multiple processes.* Multiple processes share system memory to cache their data and prefetch from multiple disks in parallel.

- *Remote disks.* Caching and prefetching for multiple processes when the disks are on remote machines.

The combined prefetching and caching systems discussed in this section all depend upon applications providing hints of their future access patterns to the prefetching system. To show that the use of hints is practical for real applications, the section ends with a survey of various techniques for providing these hints to the prefetching system.

Single disk, single process

[Cao et al. 95] derive several theoretical results regarding optimal combined prefetching and caching strategies for the single-disk case. Assuming that the entire sequence of future block references is known in advance, then any optimal prefetching algorithm whose goal is to minimize I/O stall time must obey four rules:

1. *Optimal Prefetching*: Every prefetch should bring into the cache the next block in the reference stream that is not in the cache.

2. *Optimal Replacement*: Every prefetch should discard the block whose next reference is furthest in the future.

3. *Do No Harm:* Never discard block A to prefetch block B when A will be referenced before B.

4. *First Opportunity*: Never perform a fetch-and-replace operation when the same set of operations could have been performed previously.

These theoretical results provide fundamental guidance for the design of single-disk prefetching and caching algorithms. The first two rules determine which disk block to prefetch and cache block to replace when the decision to prefetch has been made, and the second two rules regulate when (or, rather, when *not*) to prefetch.

Although an optimal prefetching and caching algorithm is impractical to implement, Cao *et al.* propose an algorithm called *controlled-aggressive* that is provably near-optimal in theory and has been shown to be even closer to optimal in practice. Controlled-aggressive prefetches blocks according to rule 1, replacing blocks in the cache according to rule 2. And it issues prefetches at every opportunity not precluded by rules 3 and 4.

[Cao et al. 96] implemented the controlled-aggressive prefetching algorithm in a file system that integrates application-controlled file caching, prefetching, and disk scheduling. The system performs cache management at two levels. First, the kernel uses a policy called Least-Recently-Used with Swapping and Placeholders (LRU-SP) to allocate memory blocks among processes; Section 2.3.1 discusses the multiple process prefetching and caching aspects of this system in more detail. Second, each process uses the controlled-aggressive algorithm to issue prefetches based upon a list of future references supplied by the application. Processes also specify general cache replacement policies, such as LRU and MRU, to enable the kernel to make better cache replacement decisions. Furthermore, the kernel issues prefetch requests in batches to make better use of low-level disk scheduling algorithms.

Using a prototype implementation called Application-Controlled File System (ACFS), Cao *et al.* demonstrate that (1) controlled-aggressive is an effective prefetching algorithm for the single-disk case, (2) prefetching is significantly more effective when combined with cache replacement algorithms that exploit knowledge of application reference patterns, and (3) practical optimizations, such as issuing prefetch requests to disks in batches, have as much impact on performance as the prefetching and caching algorithms, and are therefore essential in practical implementations.

Multiple disks, single process

The TIP2 system [Patterson et al. 95] extends combined prefetching and caching to the case where data is stored on multiple disks, such as in RAID arrays [Patterson et al. 88]. Multiple disks enable the prefetching algorithm to issue prefetches in parallel, further reducing the average latency of individual requests and exploiting the high bandwidth of disk arrays.

Patterson *et al.* introduce *cost-benefit analysis* as a new technique for dynamically managing the tradeoff between caching and prefetching. Cost-benefit analysis uses a single buffer allocator to manage memory resources. Demand faults and prefetch requests consume buffers from the allocator. The system LRU cache for unhinted processes and the hint-managed cache for hinted processes supply buffers to the allocator. On a demand fault, TIP2 consumes the least valuable buffer among the unhinted and hinted caches. When a hinted process has the opportunity to prefetch, TIP2 compares the benefit of prefetching (reduced stall time) to the cost of evicting cache buffers (reducing cache size). TIP2 decides to issue prefetches for a process by examining its future request stream up to a fixed *prefetch horizon*, defined by the time at which the process no longer benefits from prefetching. TIP2 prefetches when the benefit of prefetching missing blocks before the horizon is greater than the cost. To issue prefetches, it consumes the least-valuable block among the unhinted and hinted caches.

In both [Cao et al. 96] and [Patterson et al. 95], applications specify future references through a file and/or block hinting interface. In [Cao et al. 96], applications specify general cache replacement policies for pools of blocks in their cache, and the kernel makes replacements according to those policies on a per-process basis. In contrast, [Patterson et al. 95] shows that the reference hints can be used both to issue prefetches as well as make cache replacement decisions, and to do it system-wide

across all applications.

[Kimbrel et al. 96a] extend the theoretical studies of [Cao et al. 95] for the multiple-disk case of combined prefetching and caching. Prefetching for multiple disks is more complicated, and an optimal algorithm can violate the first three rules identified for the single-disk case. The source of the complication is the parallelism available among the multiple disks. Algorithms like controlled-aggressive and fixed-horizon will greedily use available parallelism among the disks when issuing prefetches. However, the degree of parallelism is limited by the placement of data among disks, leading to load imbalance. If the blocks that need to be prefetched are on only a subset of the disks, then the requests will have to queue. This imbalanced load can lead to non-intuitive optimal prefetching schedules. For example, there exist reference sequences where making a "poor" cache replacement decision early in a prefetching schedule can provide enough additional parallelism later in the schedule to not only recover from the "poor" decision but also issue more prefetches.

Kimbrel *et al.* introduce the *reverse-aggressive* combined caching and replacement algorithm, and prove that it is theoretically near-optimal for the multiple-disk case. Reverse-aggressive operates on the *reversed* reference sequence. Whenever a disk is unused, request the next block B not needed for the longest time on that disk. If the next request for B is after the first missing block, issue a prefetch for the missing block to replace B. Then transform these prefetch requests into a forward schedule by treating each fetch on the reverse schedule as a fetch on the forward, and vice-versa.

Since reverse-aggressive requires complete knowledge of the future reference stream among all processes to all disks, it is impossible to implement in practice. Following their theoretical studies, [Kimbrel et al. 96b] proposed a new algorithm, *forestall*, that is practical to implement and near-optimal in performance. Forestall issues prefetches when the time to fetch the next i missing blocks is greater than the time until the i^{th} missing block is referenced; if it does not, the application will surely block at some point before the i^{th} block is referenced. In this situation, Forestall chooses blocks to prefetch according to the *optimal fetching* and *optimal replacement* rules, and as long as it does not violate the *do no harm* rule in doing so.

Kimbrel *et al.* used trace-driven simulation to study the performance of forestall compared to reverse-aggressive and two other algorithms: a multiple-disk version of the controlled-aggressive algorithm of [Cao et al. 95], and a version the TIP2 fixed-horizon algorithm of [Patterson et al. 95]. The results are very enlightening. Controlled-aggressive performs nearly as well as reverse-aggressive

in practice when applications are relatively compute-bound. However, when I/O bound, controlled-aggressive is suboptimal because of the CPU overhead of its large number of disk requests. Fixed-horizon has similar near-optimal performance when applications are I/O-bound and there is sufficient I/O bandwidth in the disk subsystem to satisfy prefetch requests. However, when applications are compute-bound, fixed-horizon fails to look far enough ahead in the reference stream to issue all possible prefetches. Forestall exhibits the best performance of both algorithms for both system modes. It behaves like controlled-aggressive when applications are compute-bound and like fixed-horizon when applications are I/O-bound.

Multiple processes and multiple disks

The studies by [Kimbrel et al. 96a] and [Kimbrel et al. 96b] demonstrate the need for prefetching and caching algorithms to adapt to the dynamic load on the disks to obtain the best performance. However, those studies examined the problem for a single process alone, and did not address the problem of allocating limited memory resources among multiple processes when using the adaptive prefetching algorithms. [Tomkins et al. 97] study this issue.

ACFS [Cao et al. 96] and TIP2 [Patterson et al. 95] are systems that allocate memory resources among multiple processes, but their prefetching algorithms do not adapt to disk load. ACFS uses a policy called LRU-SP to partition memory among processes. LRU-SP maintains all memory blocks in a system-wide LRU list. When one process needs to consume a block, either on a cache miss or a prefetch, LRU-SP chooses the least-recently-used block for replacement (with some additional rules depending upon whether the process is explicitly managing its own cache in a manner other than LRU). TIP2 uses the global cost-benefit allocator to balance memory among all processes. The value of every block in memory can be converted into a uniform "currency" so that, when a process needs to consume a block, the least valuable block among all processes is chosen.

Tomkins *et al.* updated both systems with adaptive prefetching algorithms and compared their performance using trace-driven simulation. They extended LRU-SP with the forestall algorithm (LRU-SP/Forestall) [Kimbrel et al. 96b], and extended TIP2 with "Temporal Overload Estimators" (TIPTOE). Temporal overload estimators are new cost-benefit estimators for (1) the benefit of prefetching and (2) the cost of ejecting hinted blocks. To determine the benefit of prefetching,

TIPTOE anticipates I/O hotspots according to the forestall model. To determine the cost of ejecting hinted blocks, TIPTOE extends TIP2 to identify when disks are constrained and the block cannot be prefetched at a later time, therefore requiring a full disk access to reinstate it in the cache.

Using trace-driven simulation, Tomkins *et al.* found that the adaptive versions of ACFS and TIP2 perform better than the non-adaptive versions when multiple processes compete for memory and disk resources. However, TIPTOE does a better job of allocating resources among multiple processes than LRU-SP/Forestall. The reason is that the LRU approach allocates buffers to applications that consume data quickly but have limited reuse, instead of favoring applications that consume data more slowly but have higher overall reuse.

Remote disks

The combined prefetching and caching systems discussed in this section all assume that the disks are directly attached to the host. [Rochberg et al. 97] extend earlier work to study the problem of prefetching over the network from a file server in the Client-Drive Remote TIP (CTIP) system. CTIP extends TIP2 [Patterson et al. 95] to issue prefetches to file blocks stored on a remote server. A remote prefetch starts as an NFS request to a remote server, which reads the requested block from disk into its memory and then sends it over the network to the prefetching host. Demonstrating the extensibility of the TIP2 design, CTIP incorporates network storage by simply modeling it as an attached disk with longer latency. CTIP greatly reduces I/O stall time for programs that access data remotely. However, CTIP does not perform as well as TIP with local access for two reasons. The overhead of an NFS request is greater than a local disk request, and any unhinted disk accesses take longer via NFS than local disk.

In using remote hosts to perform prefetches, CTIP is the closest example to a combined cooperative prefetching and caching system. As future work, Rochberg *et al.* suggest that the local and remote NFS caches can be managed together to improve the performance of the system. This kind of local and remote cache management is essentially cooperative caching.

Reference hints

The combined prefetching and caching systems previously discussed all depend upon applications providing hints of their future access patterns to the prefetching system. Although this requirement might at first seem unrealistic, there are four practical methods by which applications can provide these hints without undue burden to application developers: manual, compiler, runtime, and predictive.

Manual: One approach is to use manual-based hinting, which can be used to describe arbitrary application reference sequences. [Patterson et al. 95] argue that programmers can add explicit calls in their applications to *disclose* hints to the system. These hints are declarative, specifying the identity of files and blocks and the order in which the application will access them, but not *when* they will be accessed. These disclosures declare deterministic behavior of the application, and have three advantages over advising interfaces that, e.g., declare policies such as LRU or MRU. First, disclosure remains independent of system implementation and execution environment. Second, it allows the system to effect policy without having to hardwire it. And third, it allows programmers to express hints in the same terms as their application I/O commands.

With their benchmarks, Patterson *et al.* demonstrate that manual-based hinting is viable for a wide set of applications. For example, applications that loop through a file can be easily modified to duplicate the loop. The first version performs the same offset calculations as the original, but discloses these offsets to the prefetching system without issuing the I/Os. The second loop is just like the first, but now issues the I/Os using the offsets already calculated in the first loop.

Compiler: A second approach is to have the compiler generate code to issue prefetch hints on behalf of the application without any intervention from the programmer. [Mowry et al. 96] implement this approach to hinting in the SUIF compiler[Wilson et al. 94]. For array-based scientific applications that make regular accesses to large arrays of data, the compiler can deduce the access pattern to the arrays. Based upon the access pattern, it can automatically generate code to issue prefetch requests to pages of array data before the application uses that data. Using an execution model of the system, it places the generated prefetch code just far enough ahead of the access to allow a page of data to be fetched from disk.

Although promising, this approach only applies to a particular class of applications, and is tied

to the execution model of a particular system. Also, because it statically schedules the prefetch when it generates the code to issue the prefetch, the current approach can only prefetch one page ahead of time is not suitable for the more aggressive prefetching algorithms.

Runtime: A third approach is to use the runtime system to issue hints on behalf of the application. [Chang et al. 99] present one interesting design and implementation of this approach. Applications are linked to a runtime that speculatively executes application code while it is stalled waiting for an I/O request. The speculatively executed code automatically discovers the future reference sequence of the application as it calculates the offsets of future I/O requests. The runtime does not issue I/O requests at this point; instead, it just informs the prefetching system of the future request in the form of a hint. The runtime also does not speculatively execute code that would produce side effects that would change the behavior of the application once it executes again.

Speculative execution has little overhead since the application is already stalled waiting for its I/O to return. However, this approach works well when the application is the only one running on the system. If the system is executing more than one application in a multiprogrammed environment, speculatively executing it to discover its future I/O requests is overhead that consumes CPU resources that other applications would otherwise use. This approach also does not work if future references depend upon the result of past references.

Predictive: A fourth approach is to have the system monitor the reference stream of an application, predict when the application will stall, and automatically issue prefetches on behalf of the application to prevent stalls. This approach requires no changes to the application.

[Bartels et al. 99] implement this kind of predictive prefetching in the GMS cooperative caching system. The kernel keeps track of the *fault sequence* of applications; since it does not see references to pages already in memory, it cannot deduce the true page reference sequence. It uses a Markov model to predict future faults and issue prefetches. When an application experiences a fault on page P, the system records all pages Q faulted after P. The next time the application faults on P, the system uses the Markov model to determine the next page Q most likely to be faulted next. If Q is likely to be faulted with a sufficiently high probability, the system prefetches Q in addition to fetching the faulted page P.

Bartels *et al.* found that this approach works best on applications with regular access patterns, such as array-based scientific applications, because Markov models are able to predict their faults

with high accuracy. With applications that have more irregular access patterns, like Windows-based office productivity applications, prediction accuracy is too low to make prefetching worthwhile. Furthermore, they found that, in practice, this approach only improves performance when prefetching over the network from global memory to local memory. Because the Markov model is not completely accurate, even a small number of mispredictions consume too much local disk resources, preventing the disk from being used to serve the pages necessary to handle the faults.

2.3.2 *Parallel network I/O*

By using multiple nodes for prefetching, a cooperative caching system can leverage the parallelism and aggregate I/O bandwidth of all of the disks in a workstation cluster. A combined cooperative prefetching and caching system needs a storage system that is able to place application data so that it can be accessed from anywhere in the cluster, and accessed such that the degree of parallelism and bandwidth scales with the number of nodes in the cluster.

This section describes various scalable distributed storage and file systems that can be used by a cooperative prefetching and caching system. It begins with a description of RAID disk subsystems, which subsequently inspired and influenced the design of scalable file systems. It then describes the evolution of scalable file systems, where file server nodes are the unit of scalability, and scalable storage systems, where low-level storage server nodes are the unit of scalability.

RAID disk subsystems

Redundant Arrays of Inexpensive Disks (RAID) are a classic example of a scalable storage system [Patterson et al. 88]. Externally, a RAID presents a block-level virtual disk storage abstraction to clients. Internally, a RAID consists of a controller and a set of physical disks. The controller stripes file data across $N - 1$ of the disks, and stores a computed parity block on the N^{th} disk. RAIDs achieve scalable aggregate system bandwidth with the parallelism of multiple disks, and they achieve scalable single-file bandwidth by striping data across the disks. The parity blocks provide reliability for single-disk crashes.

There is a vast amount of related work on RAID performance optimizations, reliability, and design variations on parity placement and recovery strategies. One of note is the TickerTAIP RAID

architecture [Cao et al. 93]. The original RAID model uses a single controller for all of the disks, which ultimately limits performance and provides a single point of failure. TickerTAIP addresses the problems of a single controller by replacing it with a cooperating set of controller nodes. The TickerTAIP architecture uses a controller node for a single string of disks; scalability is achieved by incrementally adding controller nodes and disk strings together. The controller nodes communicate with each other across an internal network to coordinate and distribute storage requests and parity computations. Although TickerTAIP was designed to communicate with hosts via commodity I/O channels like SCSI and HPPI, its distributed architecture suggests the use of distributed systems for scalable storage in the spirit of RAID.

Distributed storage and file systems

The Zebra file system stripes file data across a set of distributed file servers in the spirit of RAID [Hartman et al. 95]. By striping file data, single-client storage bandwidth scales with the number of servers. By computing and storing parity, the file system can tolerate the failure of a single server without impacting availability of data. Zebra implements a distributed log-based file system (LFS) [Rosenblum et al. 91]. Clients batch individual writes into large, fixed-size *logs* that are written to the file servers. Zebra then stripes these logs across the servers (as opposed to striping individual file data as in a RAID). Striping logs instead of files simplifies the parity implementation and allows the client to batch small writes into larger chunks of data that can then be striped across the servers.

xFS is a serverless network file system that extends Zebra in three key directions to improve scalability [Anderson et al. 96]. Like Zebra, xFS implements a distributed LFS, striping client logs across a set of storage servers for scalable performance and computing parity on the logs for reliability. However, in Zebra a single file manager tracks where clients store data in the striped logs and handles all cache consistency operations. Zebra also relies upon a single cleaner for the entire cluster. Lastly, Zebra stripes log segments across all servers. xFS addresses the first two limitations by (1) allowing any file system service (client, storage server, metadata manager, cleaner) to dynamically run on any node as demand and server failures require, and (2) using the N-chance cooperative caching algorithm to coordinate client caches. It addresses the third limitation by partitioning servers into groups to limit the degree of striping and impact of failures on clients.

Petal implements a distributed *virtual disk* abstraction on a cluster of servers [Lee et al. 96]. Petal virtual disks provide a block-level storage interface to clients that matches the interface of physical disks. Physically, Petal virtual disks are composed of multiple server nodes managing multiple disks. Virtual disk bandwidth scales with the number of nodes and disks comprising it, while latencies remain comparable to those of a local disk. Petal virtual disks also mirror blocks across nodes for reliability. By exporting a block interface, standard client file systems can use Petal virtual disks as if they were local physical disks while gaining the scalable performance and reliability provided by the virtual disks. File systems can also be created to exploit the features of virtual disks. For example, Frangipani is a distributed file system layered on top of Petal that provides scalable file system throughput and strong coherency guarantees [Thekkath et al. 97].

The Swarm storage system is a combination of the architecture of Zebra and the service abstraction of Petal [Hartman et al. 99]. Swarm exports the *striped log* storage abstraction, which is a low-level storage abstraction at the same level as Petal. Nodes in a Swarm cluster are just used for storage, and file systems run on clients or dedicated file servers. Like Zebra, Swarm stripes logs across all nodes in the Swarm cluster for scalable bandwidth, and computes parity on the logs for reliability.

Network-Attached Secure Disks (NASD) also provide a distributed, scalable storage abstraction [Gibson et al. 97]. In NASD, physical disks are attached directly to the network. A storage manager handles client metadata operations, and clients communicate directly with the disks for data operations. On top of the NASD storage abstraction, Gibson *et al.* have implemented three file systems: NFS (NASD-NFS), an MPI parallel file system (NASD-PFS), and a video file system. The parallel file system and video file system stripe file data across the NASD disks, providing bandwidth performance that scales with the number of disks in the system.

2.4 Cooperative caching in the World Wide Web

The last part of this dissertation explores the potential of applying cooperative caching techniques to large-scale distributed systems deployed across wide-area networks. In particular, it explores the potential of organizing Web proxy caches into a wide-area cooperative caching system. Web proxy caches have already been widely deployed as a means to improve the performance of Web browsing,

and it is still unknown to what effect Web proxy caches can cooperate with each other at large scales to improve hit rates, reduce document-access latency, and decrease network utilization.

Although a variety of architectures for wide-area cooperative Web caching have been proposed, little is understood about their performance in the large-scale World Wide Web environment. This section surveys proposed cooperative caching architectures for the Web, and discusses the limitations of the evaluations of these architectures.

2.4.1 Cooperative caching architectures for the Web

The Harvest Cache introduced the use of a hierarchy of cooperating Web caches to reduce average request latency, network utilization, and server load [Chankhunthod et al. 96]. Harvest organizes caches into a hierarchy depending upon their location in the Internet, with organizational caches at the leaves and root caches at the edge of the Internet backbones. Harvest has scaling limitations, however: the protocol used to locate documents stored in other caches introduces significant control traffic, and the static configuration of the hierarchy overloads the root caches.

To address the limitations of the Harvest design, [Tewari et al. 99] propose the use of a hierarchy only for maintaining metadata information. Individual caches maintain hints saying whether a document is stored in the caching system, and the hierarchy is used to find a cache storing the document. If the document is not stored in the system, the request is immediately forwarded to the server and the hierarchy is not used. Furthermore, different metadata hierarchies are logically overlayed across the physical caches according to a partitioning of the document name space to distribute and balanced load.

The Adaptive Web Caching design uses a mesh of overlapping multicast groups of caches and servers [Michel et al. 98]. Requests are forwarded from group to group along a path towards the server. Groups use multicast at each stage to determine if any group member stores the document. The focus of this design is on algorithms and protocols for automatically configuring group membership, as opposed to the more manual configurations of previous designs, and routing requests among these dynamically changing groups. It is not clear whether the problem of manual configuration of cache organizations warrants such complex protocols, however.

The Summary Cache uses efficient summaries of cache contents to eliminate hierarchies and

meshes altogether [Fan et al. 98]. In this design, each caches store summaries of the contents of the other caches. As a result, when a cache receives a request it knows immediately whether the system is caching the requested document, and which cache is storing it.

Summary Cache is similar in spirit to Tewari *et al.* in that the system can immediately determine whether a request is a hit or miss. The difference between the two designs is that Tewari *et al.* use a hierarchy to scale the management of the directory metadata so that no single cache has to store a directory of the contents of all other caches. Summary Cache, however, argues that with its efficient representation each cache can store complete information for at least 100 other caches. Given the likelihood that a caching infrastructure for the entire Web will contain far more caches than just 100, it seems unlikely that Summary Cache would be used at such large scales.

2.4.2 Evaluating cooperative caching

Unfortunately, even with all of this research on the design of cooperative Web caching systems, it is still largely unknown how the approaches compare to each other and which one is most appropriate for use as an Internet-scale caching system. More fundamentally, it is unknown whether scaling such caching systems to the wide-area Internet provides enough benefit to even warrant their use.

The problem is that the cooperative Web caching proposals evaluate their designs using small-scale workloads, and assume that the properties of these workloads scale to the size of the Internet. For example, Chankhunthod *et al.* evaluated Harvest using a synthetic workload of 5 clients, 2000 objects, and assumed a 80% hit rate among the clients. Tewari *et al.* evaluated their design using a number of traces, the largest of which was a 3-day trace from Prodigy generated by 35,000 clients. Fan *et al.* evaluated Summary Cache with a number of traces, the largest of which was the 1996 DEC trace generated by 10,000 clients [Kroeger et al. 96]. And Zhang *et al.* do not even evaluate the performance of Adaptive Web Caching, and the possible overheads their complex protocols have.

Evaluating these designs at large scales is difficult to do without large-scale traces of many sites and clients. Without such traces, analytic modeling appears to be the best approach for understanding the effectiveness of cooperative Web caching at large scales. A useful analytic model must capture the essential workload characteristics. One key characteristic that appears to be con-

sistent across traces and scales is the document popularity distribution. Many studies have found that this distribution follows Zipf's law: the relative probability that the i^{th} most popular document is proportional to $1/i$. Glassman found that the document popularity from a small trace of an Australian cache could be modeled well using a Zipf distribution, and others have found similar results: [Gribble et al. 97] using a trace of client requests from the U.C. Berkeley modem pool; [Almeida et al. 96a] and [Cunha et al. 95] from a set of client traces; and [Breslau et al. 99] from the DEC trace [Kroeger et al. 96].

Zipf's law has already been used to design more efficient individual cache replacement algorithms [Breslau et al. 99]. When document popularity is modeled using Zipf's law, [Breslau et al. 99] also show that Web cache hit rate has well defined asymptotic behavior as a function of request rate and cache size, and that these properties are consistent across traces. Inspired by these results, Chapterchap:web-coop uses an analytic model and Zipf's law as a basis for evaluating the effectiveness of cooperative Web caching at large scales in wide-area networks.

2.5 Summary

This chapter has surveyed previous work in cooperative caching systems and discussed the work related to the techniques used to extend cooperative caching systems in this dissertation. There has been a wide variety of local-area cooperative caching systems designed and implemented for workstation clusters. These systems have explored algorithms for coordinating cache contents [Comer et al. 90, Felten et al. 91, Schilit et al. 91, Franklin et al. 92, Dahlin et al. 94, Feeley et al. 95, Anderson et al. 96], approaches for improving algorithm accuract and efficiency [Feeley et al. 95, Anderson et al. 96, Jamrozik et al. 96], and techniques for increasing system reliability and portability [Markatos et al. 96, Markatos 96, Flouris et al. 98, Ioannidis et al. 98, Koussih et al. 99]. This dissertation extends this body of work by (1) studying the tradeoffs of using load balancing in cooperative caching systems, (2) combining cooperative caching with prefetching, and (3) exploring the use of cooperative caching techniques in large-scale wide-area distributed systems.

Traditional load balancing techniques for distributed systems have focused on algorithms for assigning jobs to servers, dynamically migrating jobs from loaded to unloaded servers, and incorporating stale load information into load balancing scheduling algorithms [Eager et al. 86a, Eager et al. 86b,

Harchol-Balter et al. 97, Mitzenmacher 97, Dahlin 99]. However, load balancing in cooperative caching systems differs from these traditional techniques in three fundamental ways. First, traditional techniques balance load by distributing jobs among the nodes in the network, whereas cooperative caching systems balance load by distributing data. Second, traditional techniques manage load directly since jobs directly consume CPU resources, whereas cooperative caching systems can only indirectly manage load since the impact of data placement on load depends upon futures accesses to that data. Third, cooperative caching systems must first give priority to locality, and only load balance when locality results in imbalanced load. Two cluster architectures, Scalable Network Services (SNS) [Fox et al. 97] and Location-Aware Request Distribution (LARD) [Pai et al. 98], employ both caching and load balancing techniques. However, SNS does not use load balancing for its cache nodes, and the techniques used by LARD are restricted to HTTP server workloads.

This dissertation proposes combining prefetching techniques with cooperative caching. Research in file-based prefetching has developed a progression of prefetching techniques for increasingly complex problem domains, starting with single disk/single process algorithms [Cao et al. 95, Cao et al. 96] and progressing through multiple disk/single process algorithms [Patterson et al. 95, Kimbrel et al. 96a, Kimbrel et al. 96b], multiple disk/multiple process algorithms [Tomkins et al. 97], and remote disk algorithms [Rochberg et al. 97].

These prefetching techniques rely upon hints from applications of their future access patterns. These hints can be provided in many ways, including programmer annotations [Patterson et al. 95], hints generated by the compiler [Mowry et al. 96], runtime hints generated by speculatively executing the application during disk stalls [Chang et al. 99], and hints provided by predicting future faults using Markov models [Bartels et al. 99]. For cooperative caching, I combine the Aggressive [Cao et al. 95] and near-optimal Forestall [Kimbrel et al. 96b] algorithms into a novel hybrid cluster-based distributed prefetching algorithm. Unlike previous algorithms, this prefetching algorithm is able to issue prefetches on multiple nodes at once, leveraging the combined parallel throughput of all of the nodes in a cluster in manner analogous to RAID systems [Patterson et al. 88, Cao et al. 93], distributed storage systems [Lee et al. 96, Gibson et al. 97], and distributed file systems [Hartman et al. 95, Anderson et al. 96, Thekkath et al. 97, Hartman et al. 99].

Finally, this chapter surveyed large-scale wide-area cooperative caching systems that have been designed for the World Wide Web. The architectures of these systems have been structured around

hierarchies [Chankhunthod et al. 96], metadata hierarchies [Tewari et al. 99], directory exchanges [Fan et al. 98, Menaud et al. 98], hash routing [Karger et al. 99, Valloppillil et al. 98], meshes of topologically overlapping multicast groups of caches and servers [Michel et al. 98], and interest-based multicast groups [Touch 98]. Unfortunately, these systems have been evaluated only using small scale workloads, and previous work has yet to address the question of whether cooperative caching techniques are worth scaling to very large workloads. To answer this fundamental question, this dissertation explores the potential use of cooperative caching for the Web across scales ranging from small organizations of hundreds of clients to national regions of tens of millions of clients.

Chapter 3

Load Balancing in Cooperative Caching Systems

This chapter explores the use of load balancing in cooperative caching systems for workstation clusters. A crucial aspect of the design of cooperative caching systems is the global page replacement algorithm, which selects the target nodes to receive evicted (i.e., paged-out) pages. Previous cooperative caching systems have primarily used page age information to select target nodes. Although using age information can provide near-optimal page replacement selections, it can still lead to suboptimal overall performance due to contention at memory servers. This contention increases the latency of remote page requests, thereby decreasing overall application performance.

Memory servers experience contention in two situations. First, dynamic application behavior can result in both sustained as well as short-term periods of high request rates. Sufficiently high request rates increase the latency of remote page requests due to contention at memory servers. Furthermore, the global page replacement algorithm can also concentrate old or young pages at one or a few memory servers. Combined with periods of high request rates, these concentrations will focus request traffic at these servers, further increasing contention. Second, the load imposed on any given node that is handling global memory traffic may degrade the performance of the local jobs on that node.

To mitigate the effects of contention at memory servers in these situations, I explore the use of load balancing to reduce server contention and limit the impact of request traffic on local jobs on memory server nodes. In particular, I study the potential benefit and the potential harm of using memory server CPU load information, in addition to age information, in cooperative caching replacement policies. By considering load information in the replacement algorithm, cooperative caching systems can reduce contention at servers and limit CPU utilization by remote requests.

I explore the use of load balancing in cooperative caching systems using a combination of an-

alytic models and trace-driven simulation. Using an analytic queueing network model, I show that uneven request distributions among servers can substantially degrade memory request performance for various idle node configurations. I then show the benefits that can be gained by load balancing requests while ignoring the impact of altering the replacement policy. I also show that cooperative caching systems do not scale if there are only a few memory servers available. However, when requests are load balanced across servers, I show that eight memory servers are sufficient to handle the load of a 70 node cluster.

Although load balancing is important for performance and scaling, there are tradeoffs to its use. Replacements made to balance load can cause the system to deviate from its original replacement algorithm. This deviation from policy can increase the overall number of disk accesses when load balancing is used, mitigating the benefits of load balancing. To evaluate this tradeoff, I use trace-driven simulation to measure the impact on application performance of deviating from the cooperative cache replacement policy. I find that deviating from the strict replacement policy, even substantially for some applications, does not significantly degrade overall application performance. Based upon these results, I conclude that cooperative caching systems can benefit considerably from load balancing requests with little harm from suboptimal replacement decisions. Finally, I use the intuition gained from the experiments to propose a new family of algorithms that incorporate load considerations as well as age information in global memory replacement decisions.

3.1 Introduction

A crucial aspect of the design of cooperative caching systems is the global page replacement algorithm, which selects the target nodes to receive evicted (i.e., paged-out) pages. For example, in the xFS distributed file system [Dahlin et al. 94], pages displaced from one node are sent randomly to other nodes and an algorithm called *N-chance* is used to recover from bad selections. In the Global Memory Service (GMS) [Feeley et al. 95], the system maintains global information describing the ages of pages on all cluster nodes; using this information, GMS implements an approximation to network-wide global LRU replacement. Experiments with GMS show that the use of global information improves performance relative to a random algorithm. [Sarkar et al. 96] propose a fully distributed information exchange (piggybacked on global page forwarding operations) to gather less

complete but perhaps more up-to-date page age information, leading to a different approximation of global LRU replacement.

Although using age information can provide near-optimal page replacement selections, it can still lead to suboptimal overall performance due to contention at memory servers. First, dynamic application behavior can result in both sustained and short-term periods of high request rates. Sufficiently high request rates increase the latency of remote page requests due to contention at memory servers. Furthermore, the global page replacement algorithm can also concentrate old or young pages at one or a few memory servers. Combined with periods of high request rates, these concentrations will focus request traffic at these servers, further increasing contention. Second, the load imposed on any given node that is handling global memory traffic may degrade the performance of the local jobs on that node. For example, experiments with GMS showed that a node supplying pages to seven active nodes used 56% of its CPU cycles just for servicing the remote page requests [Feeley et al. 95].

The first part of this dissertation explores the use of load balancing to reduce contention at memory servers and limit the impact of serving requests on local jobs on memory servers. Server contention arises when the total request rate from active nodes to memory servers is sufficiently high that the requests queue at servers and interfere with each other. When requests queue at memory servers, they take longer to serve. And the longer they take to serve, the slower applications run when using cooperative caching. If the request rate is sufficiently high, there is so much contention that requests to memory servers take longer than if they were to use the local disk.

Memory servers can experience extremely high request rates either in sustained stretches of time, or sudden, short-term bursts. When there are only a few memory servers for a large number of active nodes, the combined request traffic to those servers can result in sustained contention that generates sufficient overhead to negate the benefit of cooperative caching. In this case, the system should disable cooperative caching, or at least only allow a subset of the active nodes to use the memory servers.

Even with a reasonable balance of active nodes and memory servers, there are two situations that can lead to short-term, dramatic increases in the request rate to the memory servers. First, during the course of their execution applications go through periods where their peak fault rate is significantly higher than their average fault rate. For example, experiments by [Jamrozik et al. 96]

found that, during the execution of the Modula-3 compiler `m3`, the average fault rate for the program was 670 page faults per second, but that there were periods where it experienced 41,500 page faults per second over the course of 5 seconds. Such page fault rates can easily overwhelm a memory server.

Second, when a node transitions state from *idle* to *active* (e.g., because the user starts to use the node again), it needs to make its memory resources available for use by local applications. To do this, the memory pages it has been storing for other active nodes now need to be moved to other servers. Considering that a node may be serving thousands of pages on contemporary machines, this transition can rapidly result in thousands of page forwarding requests to the remaining servers in the cluster.

These bursts of request traffic can experience contention if they are concentrated on just a few or even one memory server. The distribution of request traffic is determined by the global page replacement algorithm used by the cooperative caching system. When a global page replacement algorithm chooses servers based upon the age of pages on those servers, such as in GMS or the algorithm proposed by Sarkar *et al.* , then the distribution of requests to those servers will depend upon the relative ages of pages they store. If the distribution of page ages among servers is uneven, such as when one server has a concentration of old pages, then the distribution of requests to the servers will be uneven. When the request distribution is uneven, requests will concentrate on and potentially overwhelm individual servers.

A common situation where memory servers experience uneven distributions of page ages is when a node transitions state from *active* to *idle* and becomes a memory server (e.g., some time after the user stops using the node). After this transition, its memory resources become immediately available for use by the rest of the cluster. Since this memory is now free for use, active nodes will begin to concentrate their eviction requests at the newly idle node.

This situation can have long-term effects as well. For example, assume one server holds a large set of pages of the same age. When that set of pages becomes the oldest in the cluster, eviction requests from active nodes will concentrate at the server as those pages are replaced by newer ones. The combination of a concentration of oldest pages at one node, and other nodes operating at high fault rates due to transitions in application behavior or node state, can lead to significant server contention.

To mitigate the effects of contention at memory servers in these situations, this dissertation studies the potential benefit and the potential harm of using memory server CPU load information, in addition to age information, in cooperative caching replacement policies. By considering load information in the replacement algorithm, cooperative caching systems can reduce contention at servers and limit CPU utilization by remote requests. There are tradeoffs to the use of load information, however. Replacements made to balance load can cause the system to deviate from its original replacement algorithm, potentially increasing the overall number of disk accesses when load balancing is used.

Extending cooperative caching to incorporate load balancing raises a number of challenging questions. First, what is the impact on remote page fault latency of contention at memory servers? How sensitive is remote page fault latency to uneven load distributions across memory servers? Under what conditions does contention at memory servers negate the benefits of cooperative caching? What is the impact on application performance when load balancing causes suboptimal global page replacement decisions (e.g., replaces younger pages before older pages)? And finally, how can we extend the global page replacement policy to incorporate load information as well as age information?

To answer these questions, I use an analytic queueing network model and trace-driven simulator of a cooperative caching system. With the analytic model, I show that uneven request distributions among servers can substantially degrade memory request performance for various idle node configurations. I then show the benefits that can be gained by load balancing requests while ignoring the impact of altering the replacement policy. In a small cluster of 18 nodes with moderate skew in load, the overall execution time of a database application increased by 12%. For larger clusters, the impact is more severe. For a slightly larger cluster of 24 nodes, the execution time of the same application increased by 37%.

With the trace-driven simulator, I study the qualitative impact on application performance of deviating from the cooperative cache replacement policy. I show that deviating from a strict replacement policy, even substantially for some applications, does not significantly degrade application performance. For a representative application, replacing pages even 30% from the bottom of the LRU stack only introduces 9% more disk faults (for a 2% increase in execution time). From these results, I conclude that global LRU page replacement is relatively *insensitive* to exactly which

of the oldest pages is replaced. Since applications are relatively insensitive to which oldest page is replaced, the cooperative caching system has the opportunity to choose a replacement based upon load information as well as age information, and thereby gain the benefits of load balancing shown by the analytic model.

Finally, I use the intuition gained from the experiments by proposing a new family of algorithms that incorporate load considerations as well as age information in global memory replacement decisions. These algorithms use a small set of measured system parameters, as well as two tunable parameters (i.e., the weight of load balancing relative to age information, and the stack location for exchange of global page age information) to balance the load across memory servers while still implementing approximate LRU global page replacement.

The results of our study are applicable to global memory systems that implement approximate global LRU replacement using varying types of global information. Furthermore, the general insights about the benefit of balancing load at minimal costs due to less perfect replacement decisions, as well as the new family of algorithms suggested by these insights, can also be applied to global memory systems that use replacement algorithms different than LRU.

The remainder of this chapter is organized as follows. Section 3.2 describes the specific cooperative caching system I study in this chapter. Section 3.3 describes the queueing network model and trace-based simulator used in this study. In Section 3.4 I present performance results that show the extent to which load balancing can benefit global memory systems, and the impact of deviating from LRU replacement. Section 3.5 studies the performance issues involved in using non-idle nodes, in addition to idle nodes, as memory servers. I then present our new family of global memory replacement algorithms in Section 3.6 that incorporate server load information as well as page age information into the page replacement decision. Finally, Section 3.7 summarizes our results and conclusions.

3.2 Cooperative Caching Systems

Our study is based on cooperative caching systems such as the *Global Memory Service* (GMS) described and implemented in [Feeley et al. 95] or the cooperative caching systems developed for file systems [Dahlin et al. 94, Sarkar et al. 96]. The use of cooperative caching is transparent to

applications and is managed entirely by the operating system. GMS, for example, implements cooperative caching at the lowest level in the operating system; thus cooperative caching is available for virtual memory backing store, mapped files, and the file buffer cache.

Each node in a cooperative caching system can contain both *local* pages and *global* pages. A local page is one that contains data for an application executing on the same machine; a global page is one that contains data for an application executing on another machine. The balance between local and global pages on a node changes over time, depending on the behavior of that node and others. When a node becomes idle, its pages will age, and eventually will be replaced by younger global pages displaced from other nodes. When an idle node begins computing again, it will increase its use of local memory, forwarding or discarding global pages as it does so. Eventually, if the applications on the node have high memory demands, the node will fill its memory with local pages and will begin to use memory on other nodes as it replaces its oldest local pages.

The global pages across all of the nodes in the cluster are called *global memory*. Global memory forms a shared cache between local memory and disk. (Note that pages in local memory are never shared). Coherency in this cache can be handled, for example, by flushing dirty local pages to disk before placing them in global memory at another node, as is done in GMS. Pages in global memory can then be shared among nodes without any locking. In practice, the code pages of an application are shared by multiple instances of the application executing on different nodes, while the data pages are private.

Current cooperative caching systems [Feeley et al. 95, Sarkar et al. 96] maintain approximate global age information for both local and global pages, and use this age information to manage the memory in the cluster as a single resource. When a node needs to replace a page, it locally chooses a target node based on its global age information. The replaced page is then sent to that target node, where it becomes a global page. I describe the gathering of global age information further in Section 3.6.

In general, there are two classes of jobs that run in a cooperative caching system: *local jobs*, whose memory requirements are completely satisfied by local memory resources on the node on which they run, and *global jobs*, which benefit from the use of global memory while executing. Given these two job classes, nodes in the system can be separated into three classes:

- *local nodes*, which are running local jobs only,
- *global nodes*, which are running global jobs (and possibly local jobs as well), and
- *idle nodes*, on which no jobs are running, but which may provide global pages for global jobs on other nodes.

Note that in general a global node is running exactly one global job and perhaps also some local jobs as well.

Idle and local nodes can act as *memory servers* for global nodes, in the sense that a portion of their memory can be allocated as global memory. Global jobs running on global nodes make global memory requests, and these requests are sent to the idle or local nodes managing the global memory for these jobs. These requests come in two forms, *getpage* requests and *putpage* requests. A getpage is a request to read a global page from another node's global memory, e.g., in response to a page fault or file cache miss. The global node thus sends a getpage request to the remote node to ask for the page, and the remote node returns it. A putpage is a request to store a page in another node's global memory, e.g., as the result of a page replacement decision. The global node sends a putpage request along with the page to be stored.

In GMS, a node cannot both be a global memory server and a global memory client. Furthermore, in any cooperative caching system, a global node that is an active global memory client will not simultaneously have enough memory to be a significant memory server for other nodes.

3.3 Model and Simulator

This section describes the queueing network model and the simulator I use in this study. Our first goal is to characterize the performance of getpage requests (remote page requests) in the face of contention at memory servers. By doing so I can show the potential benefits of load balancing global memory requests across servers to reduce contention. I use a queueing network model and Approximate Mean Value Analysis (AMVA) for this experiment, because it is a natural tool for measuring the effect of contention on getpage response time. The queueing network analysis is also useful for studying the interference that local jobs experience from global memory requests.

Table 3.1: The applications used in this study.

Application	Description	Input	Mem Refs (10^6)	Footprint (pages)	Z (ms)
m3	Modula 3 compiler	smalldb	87	773	2.7
ld	Linker	Digital Unix 3.2	102	6807	0.36
atom	Binary rewriting tool	kproc tool on gzip	117	1175	1.5
render	Graphics renderer	model of tree	245	1433	4.1
gdb	GNU debugger	init phase	0.5	138	0.17
OO7	OODB benchmark	synthetic workload	-	-	5.0

The AMVA model does not, however, take into account the effect of deviating from a strict LRU replacement policy when managing global pages. Therefore, our second goal is to measure the effect on application performance when global pages are not replaced using strict LRU. I use the simulator for this experiment because it allows us to measure and compare deviations from strict LRU in terms of the number of page faults and execution time for a set of applications.

The applications used with the AMVA model and in simulations in this chapter are summarized in Table 3.1. For each application, the table gives the name and a description of the application, the input used to drive it, the number of memory references made and memory footprint, and the average time between global memory requests.

The number of memory requests is the total number of loads and stores made by the application. The memory footprint is the total number of pages touched by the application during its execution; in other words, this is the total number of local and global pages that would be required to prevent the application from having to page to disk. The average time between global memory requests is the resulting fault rate to global memory when GMS is configured to be able to hold 1/4 of an application's footprint in local memory and 3/4 of the footprint in global memory. I chose this ratio of local to global memory as a reasonable approximation to the resources applications will likely find available in real cooperative caching systems. These parameters were all derived from simulation experiments, except for the OO7 benchmark [Carey et al. 93] whose parameters were obtained from the GMS prototype. Note that, since I do not have a trace for OO7, I do not have

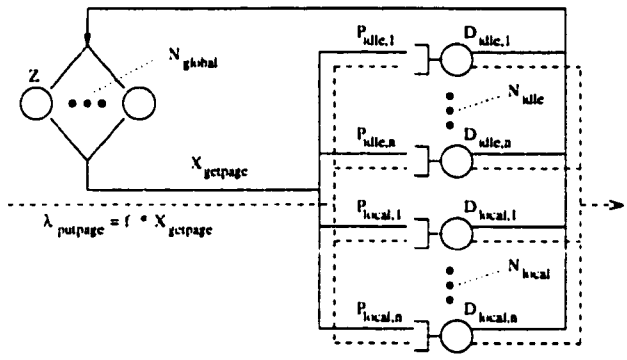


Figure 3.1: Queueing network model of the Global Memory Service.

measurements of the number of memory references or its memory footprint.

3.3.1 MVA Model of Global Memory Requests

I use the 2-class queueing network model shown in Figure 3.1 to model global memory request behavior, and use approximate mean value analysis to compute performance metrics of interest. The model is designed to measure the effect of contention on getpage latency and the CPU utilization of memory servers. To accomplish this, a closed class of customers in the network represents the global nodes, each executing one global job that alternates between executing and sending a synchronous getpage request to an idle or local node. The open class of customers represents the asynchronous putpage requests that are issued by the global nodes. The effect of server contention on getpage requests can be estimated by solving for the response time of the getpage requests. Memory server utilization can be estimated by solving for the utilization of the server queues in the network.

The input N_{global} denotes the number of global nodes. Each global node is executing a single global job as well as possibly some local jobs. The center with average delay Z corresponds to time the global nodes spend executing between a getpage response and the next getpage request by the global job. The other queues correspond to idle and local nodes in the system; these are the nodes that serve the getpage and putpage requests. Note that in the simple case I assume that global nodes are homogeneous; that is, all global nodes have the same average delay between getpage requests and the same routing probabilities for the requests. Routing probabilities are perhaps expected to

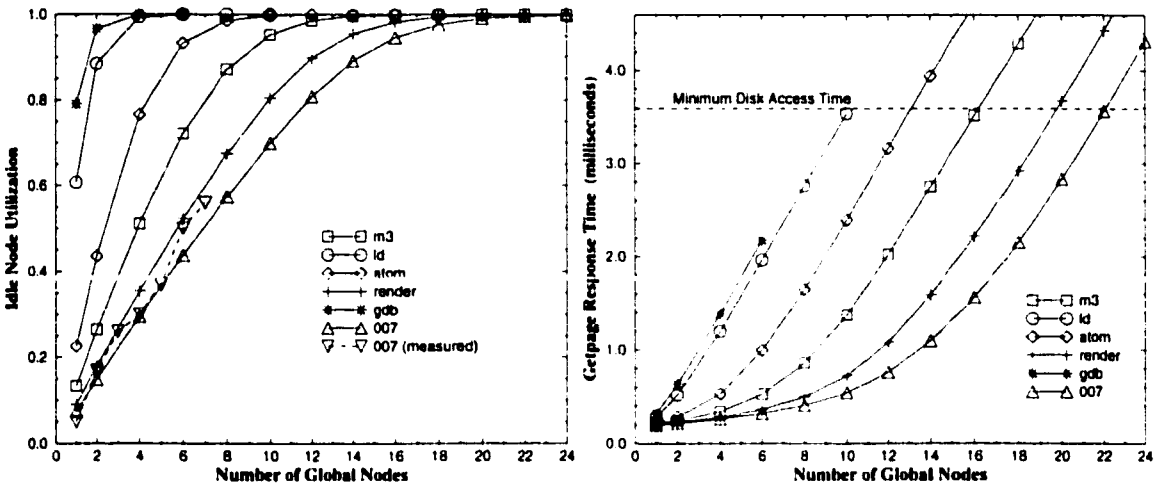


Figure 3.2: Utilization of the idle node and response time of a getpage request versus number of global nodes.

be homogeneous because they depend on the page forwarding algorithm, which is the same for all nodes. However, the model is easily extended to handle non-homogeneous request rates or routing probabilities.

Since putpage requests are asynchronous operations (a process can continue to execute while the page is being sent to the server), I model them as an open customer class. Their arrival rate is denoted by $\lambda_{putpage}$, which is a constant f times the getpage throughput $X_{getpage}$, where f is an input to the model. Note that $X_{getpage}$ is an output of the model, which means that I must solve the model iteratively to determine the correct value for $\lambda_{putpage}$ as well as the desired output measures.

Both getpage and putpage requests flow into and out of a subsystem consisting of two sets of service centers, one set corresponding to the idle nodes in the system and the other corresponding to the local nodes in the system. Note that, although I distinguish between idle and local service centers in the model notation and routing probabilities, each type of node services getpage and putpage requests in the same way due to the priority of global requests. That is, since operating systems typically manage memory in the kernel, a cooperative caching system will service requests in the kernel. Consequently, when a global request arrives at a local node, the request gets top priority at the local node and is immediately scheduled ahead of any local work on that node. As

with idle nodes, the only interference a global request will encounter on a local node will be due to other global requests being serviced at that node at the arrival instance. Given their priority, global requests can therefore impact the performance of local jobs on the node; this impact is investigated in Section 3.5.

The number of idle nodes is denoted by N_{idle}, and the number of local nodes is denoted by N_{local}; both parameters are inputs. The service demands of the idle and local service centers are defined to be $D_{idle,i}$, $1 <= i <= N_{idle}$, and $D_{local,l}$, $1 <= l <= N_{local}$, respectively. Note that these service demands are inputs that apply to both getpage and putpage requests.

There are also two sets of probabilities associated with the service centers, $P_{idle,i}$, $1 <= i <= N_{idle}$, and $P_{local,l}$, $1 <= l <= N_{local}$. These parameters correspond to the probability that a request will go to the associated service center. With these probabilities, I can control the distribution of requests among the service centers within a set as well as the distribution between the two sets of service centers. The former enables us to model arbitrary request distributions among idle nodes in Section 3.4.2, and the latter to model uneven request distributions between idle and local nodes in Section 3.5. Note that both getpage and putpage requests share these probabilities, and that the sum of both sets of probabilities equals 1. These probabilities are also inputs.

3.3.2 Model Parameterization and Validation

I parameterize the model using workload parameters both from an implementation of the Global Memory Service [Feeley et al. 95] and from results from our simulator, which is described subsequently in Section 3.3.3.

Based upon measured values reported in [Feeley et al. 95], I set the service demands of both idle and local service centers to 194μs. Since I assume putpage and getpage requests have the same demand, and the page size never changes, this service demand never changes for any of our experiments. Based upon simulation results, I have found that, in a cluster running a heterogeneous workload, the number of getpage and putpage requests are nearly equal. I therefore set $f = 1$ to correspond to an equal number of putpage and getpage requests. The think time Z is set according to the application being modeled; values for each application are listed in Table 3.1. Unless otherwise specified, the probability distributions are uniform across all service centers. The other inputs, such

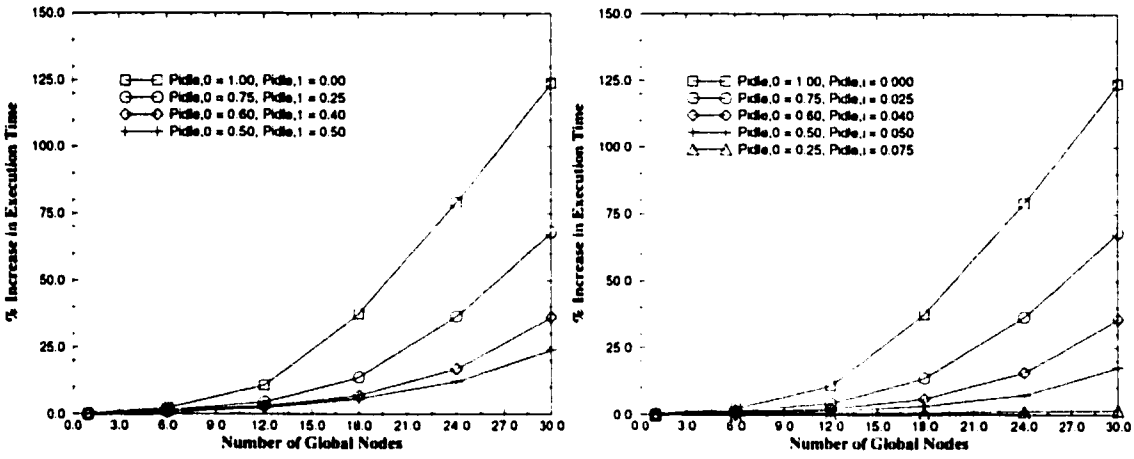


Figure 3.3: Percent increase in execution time due to uneven distributions of requests across idle nodes. The first graph corresponds to a network with 2 idle nodes; with probabilities $P_{idle,0}$ and $P_{idle,1}$, requests go to one idle node or the other. The second graph corresponds to a network with 11 idle nodes; with probabilities $P_{idle,0}$ and $P_{idle,i}$, requests go to the first and each of the remaining 10 idle nodes, respectively.

as the number of each kind of nodes, are set specifically for each experiment with the model.

Using these inputs and the average request rate for the OO7 benchmark, I can now perform an experiment similar to the one performed in [Feeley et al. 95] and compare the results. In the experiment, I simply vary the number of global nodes (N_{global}) making requests to a single idle node ($N_{idle} = 1$). The experiment performed on the prototype measures the utilization of the idle node as the number of global nodes ranges from 1 to 7.

The results of this experiment from the MVA model are shown along side results from the prototype in Figure 3.2. This graph plots the utilization of the idle node and response time of a getpage request versus the number of global nodes in the system. To put the getpage response times in perspective, the fastest disk page fault time of 3600μs reported in [Feeley et al. 95] is shown on the response time graph. As with the prototype, the model exhibits linear growth in idle node utilization when there are fewer than 8 global nodes. Furthermore, with 7 global nodes, the measured idle node utilization of the prototype was 56%, and the model reports a utilization of 51%, which is in surprisingly close agreement, particularly considering the level of abstraction in the model compared to the actual system implementation. This degree of agreement between measured

system performance and the MVA model calculations gives a reasonable amount of assurance that the model is generating good results regarding the impact of contention on the average performance metrics I am interested in.

3.3.3 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 [Srivastava et al. 94]. 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 with global knowledge 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 spent waiting for pages. More details on the simulator, including the capability to simulate subpage fetching policies not used in this chapter, can be found in [Jamrozik et al. 96].

3.4 Issues in Load Balancing Memory Requests

This section studies the potential benefits and also the potential harm that can result when global memory requests are load balanced across a given set of memory servers. First, I discuss the conditions under which uneven request distributions occur in cooperative caching systems. Then, using the analytic model, I study the degree to which uneven load degrades the performance of remote page requests, or conversely how much load balancing can improve request performance by eliminating contention. Load balancing the requests also forces possibly undesirable deviations from the strict LRU replacement policy. These deviations can potentially negate the benefits of load balancing. In Section 3.4.3 I use the simulator to study the impact on application performance when page replacements deviate from a strict LRU replacement policy.

3.4.1 Sources of Contention

Server contention arises when the total request rate to global memory servers by all global nodes is sufficiently high that the requests interfere with each other. On average, each global node will issue global memory requests relatively infrequently; otherwise application performance suffers. However, a number of situations can lead to sudden short-term increases in the request rate to one or more of the servers. As one example, during the course of their execution applications go through periods where their peak fault rate is significantly higher than their average fault rate. As another example, when nodes transition from one state to another, they can produce increased traffic to the servers. When a node transitions from idle to local or global (e.g., because the user starts to use the node again), the global pages it has been storing for other global nodes need to be moved to other servers as those pages become dislocated by local processes on the node. Considering that a node may be serving thousands of pages on contemporary machines, this transition can quickly result in thousands of putpages to the remaining servers in the cluster.

When a node transitions from local or global to idle (e.g., some time after the user stops using the node), its pages become immediately available to the rest of the cluster. Since these pages are likely the oldest in the cluster, putpage requests will begin to concentrate at the newly idle node. This situation can have long-term effects as well. For example, assume one server holds a large set of pages of the same age; when that set of pages becomes the oldest in the cluster, putpage requests will concentrate at that node as those pages are replaced by newer ones. The combination of a concentration of oldest pages at one node, and another node operating at high fault rate due to a transition in its application behavior or its node state, may lead to particularly significant server contention.

3.4.2 Benefits of Load Balancing

To study the potential benefit of load balancing global requests, I use the analytic model to quantify the relative performance of getpage requests (page faults) for varying degrees of imbalance among servers. In particular, I quantify:

- how the contention produced by situations with load imbalance degrades getpage performance,

- how load balancing improves getpage performance,
- the number of idle nodes required to serve various numbers of global nodes under balanced conditions.

Potential improvement in execution time

I study the first two issues with the following experiment. I consider a system with two idle nodes and a variable number of global nodes. Each of the global nodes is running the OO7 benchmark described in Section 3.3.2. I then vary the probabilities ($P_{idle,0}$, $P_{idle,1}$) so that requests will go to one idle node or the other with various distributions. For this experiment, the distributions range from (0.5, 0.5) to (1.0, 0.0), covering the range from completely balanced to completely unbalanced. I then solve the model for getpage response time R for each of the distributions and, using R and the think time Z, compute the increase in execution time for the application. Note that since the OO7 benchmark has a larger value of Z than the other applications I have traced (see Table 3.1), the estimates for performance degradation due to load imbalance (and the estimate of potential performance improvement that could be realized by load balance) will be conservative for this set of applications.

Because the first system is small compared to what one would expect to find in real cooperative caching systems, I consider a second system similar to the first but larger. Instead of two idle nodes, though, the second system has 11 idle nodes. The distributions for this experiment range from (0.25, 0.075) to (1.0, 0.0); in this system, the first probability $P_{idle,0}$ is for the first idle node, and the second probability $P_{idle,i}$ is the probability for each of the 10 remaining idle nodes. With these distributions, I are able to model a large cluster that has old memory heavily skewed to one node. Such a skew reflects situations such as when a node becomes a newly idle node in the cluster.

Figure 3.3 graphs the results of the experiment on the two systems. Each curve corresponds to a request distribution, and shows the increase in execution time of the OO7 application as the number of global nodes increases. There are two important observations to note from these graphs. The first is that skew increases execution time. With a skew of (0.75, 0.025) at 18 nodes, for example, the execution time of all applications making global memory requests increases by 12%. The second is that the total number of idle nodes matters little when the skew is primarily to one node. With a

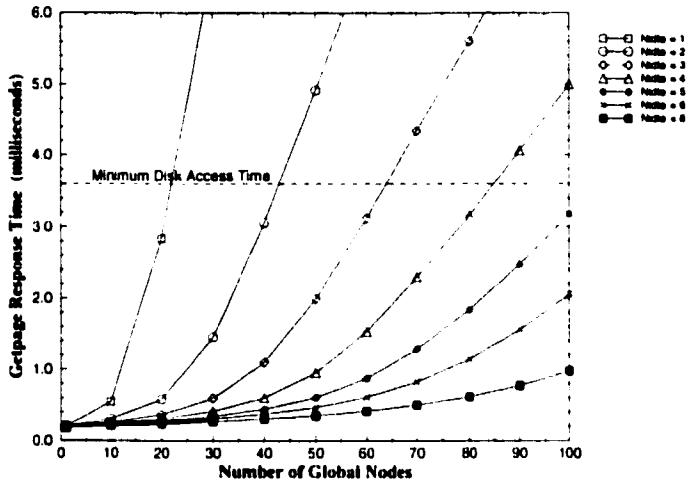


Figure 3.4: Response time of getpage requests versus the number of global nodes as the number of idle nodes in the network, N_{idle}, varies from 1 to 8. For each solution of N_{idle} I use the parameters for the OO7 benchmark. Requests are balanced across idle nodes.

similar skew of (0.60, 0.040) at 24 nodes, the execution time increases by nearly the same amount.

Required number of global memory servers

Another issue involving request contention at memory servers is the relationship between the number of global nodes and the number of idle nodes in the cluster. At the two extremes, a large number of global nodes will overwhelm a small number of idle nodes, and a relatively large number of idle nodes are going to smoothly handle a small number of global nodes. But typical clusters are not likely to be at the two extremes. The flip side of the load balancing question is how many servers do I need to balance the load? At one extreme, a large number of idle nodes will effectively handle a small number of global nodes even in the presence of skew in the requests (see Figure 3.3). At the other extreme, a large number of global nodes will overwhelm a small number of idle nodes even if load is balanced. However, during the operation of a cluster there will typically be significant fluctuations between the two extremes. A relevant question when faced with a moderate number of idle nodes and a significant skew in the requests by the default replacement policy is, to how many nodes should I redistribute some of the skewed requests?

Figure 3.4 provides insight into this issue for a wide range of system configurations between the two extremes. Each curve in the graph corresponds to a particular number of idle nodes in the cluster, and plots getpage response time as the number of global nodes increases. The global nodes are configured to run the OO7 benchmark, which gives optimistic results for our set of applications in the sense that the other applications will have response times that are greater than the ones shown in the figure. The dashed horizontal line at 3.6 ms corresponds to the minimum disk access time, as described in Section 3.3.1. Note that requests are again balanced evenly across idle nodes.

This graph shows that, when requests are balanced across memory servers, more than three idle nodes are needed to provide good service to medium or large numbers of global nodes. On the other hand, with eight balanced idle nodes the getpage response time only doubles when there are 60 global nodes in the cluster. This indicates that, with large clusters, traffic of the global nodes can be spread across a relatively small number of idle nodes.

3.4.3 Potential Harm of Load Balancing

The tradeoff with load balancing global memory requests is the impact on the global page replacement algorithm. The page replacement algorithms in current cooperative caching systems implement some form of approximate LRU replacement [Dahlin et al. 94, Feeley et al. 95, Sarkar et al. 96]: when they have to make a page replacement decision, they strive to replace the oldest page in the system. However, based on the MVA results, there is potential benefit to modifying the replacement decision to balance the request load across servers. This means that, due to balancing, pages can be replaced at some position possibly relatively high on the LRU stack instead of at the bottom. Such suboptimal replacements can potentially degrade application performance.

In this section I study this issue in further detail. First I look at the relative frequencies of accesses to pages in the LRU stack for several of the applications in Table 3.1, as well as a number of additional applications described below. Then I measure the effect on application performance when the replacement policy deviates from strict LRU. Note that it has long been known that for many applications approximate implementations of LRU replacement have nearly the same performance as exact LRU; what I study here is the potential impact of repeatedly and deliberately replacing pages that are as high as 50–90% up in the LRU stack. I use the trace-driven simulator for our experiments,

since an MVA model cannot capture the impact of specific replacement policy decisions.

Table 3.2: Additional applications used for LRU experiments.

Application	Description	Length	Pages
applu	Solve parabolic/elliptic PDEs	1068	3631
blizzard	DSM binary rewriting tool	2122	3908
m88ksim	Microprocessor simulator	10020	4838
murphi	Protocol verifier	1019	2345
perl	Interpreted scripting language	18980	9836
swim	Shallow water simulation	438	3754

For the LRU experiments, I measure the behavior of an additional set of applications to broaden the scope of the results. The additional applications are from the study in [Glass et al. 97], and are summarized in Table 3.2. For each application, the table gives the length of the traced application (in millions of instructions) and the amount of total memory used by the application (in number of 4K pages).

The memory behavior of these applications was simulated based upon compressed traces of page references. Note that the simulator used for these traces is different than the one in 3.3.3; since the traces are incompatible with the GMS simulator, I implemented another simple simulator to measure the LRU stack behavior of these applications. The format of the page reference traces, as well as the constraints imposed on simulations using the traces, are fully described in [Glass et al. 97]. I describe how these constraints interact with our LRU simulator when I describe the results of our experiments.

The LRU stack

To quantify the relative importance of a page to an application, I simulated the individual execution of each application on a single node. For each execution, the node had enough memory to hold all the pages the application needed. After simulating the memory behavior of each application on its inputs, the simulator calculated the fraction of memory references that occurred to each position in the LRU stack. Due to the method by which page references were compressed in the compressed

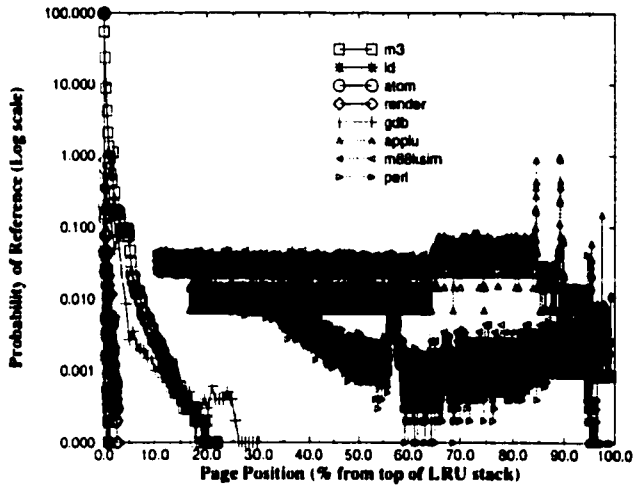


Figure 3.5: Probability of reference for pages in the LRU stack.

traces, references to the top of the stack (which is always in memory) were not recorded. Thus, for those applications, the simulator cannot calculate the fraction of references to positions in the top 10–20% of the stack; furthermore, the fraction of references for lower positions in the stack are the fraction of the *recorded references* that occurred to each position.

Figure 3.5 shows the results of this experiment. Each curve corresponds to the LRU stack of an application. For each position in the LRU stack, the probability that it was referenced is plotted on a log scale. So that clarity is not entirely sacrificed, only three of the compressed trace applications, applu, m88ksim, and perl, are shown in the figure. These three applications belong to three different types of applications reported in [Glass et al. 97], respectively. Interestingly, for each compressed trace application not shown, its curve would overlie the curve shown for the application in its class.

The GMS applications are clustered on the left of the graph due to very low probability of accessing pages in the bottom 70% of the stack. Note that, for these applications, the pages on the bottom 95% of the LRU stack are each accessed with less than 1% probability. The curves for the compressed trace applications begin around 20% from the top of the LRU stack due to the constraints imposed by the method in which the traces were generated. Note that, none of the positions measured are accessed with greater than 1% probability; furthermore, the probabilities for the portion of the LRU stack that could be accurately simulated are inflated because the references to the

top 10–20% of the stack, missing in the compressed trace, could not be counted in the denominator.

These long tails of positions with low probability of reference have promising implications for load balancing. Since load balancing global memory requests will result in page replacements above the bottom of the LRU stack, but always the oldest page on a give node, these suboptimal replacements may not severely degrade the performance of the application.

Deviating from LRU

To quantify the effect on overall execution time of deviating from strict LRU, I simulated the execution of the GMS applications running on one global node and paging to one idle node. The global node had 1/4 of the total memory needed by each application, and the idle node had 1/2 of the total memory needed. I modified the replacement policy of the simulator to replace the page at a specific position from the bottom of the LRU stack, instead of replacing the oldest page. This position was varied from the oldest page to the youngest page in *global memory*, denoted by 0% to 100% from the bottom of the LRU stack, respectively. Replacements are only varied within global memory because load-balancing only affects pages in global memory.

For the compressed trace applications, I simulated a memory capable of holding 3/4 of the total amount required by an application; the 3/4 was chosen so that the 1/4 and last 1/2 of the memory would correspond to the breakdown between local and global memory of the GMS simulations. The compressed trace simulator was likewise modified to replace pages at a specific position from the bottom of the LRU stack, again varying from oldest to youngest in the global memory portion of the LRU stack.

Figure 3.6 shows the results of these simulations. Each curve corresponds to an application and plots the percent increase in number of disk faults versus the position in the global LRU stack which was selected for page replacement. The GMS applications are shown with solid lines, and the compressed trace applications are shown with dotted lines.

These curves show two important effects. The first is that some applications (ld, atom, render, and applu) improve their performance when the policy deviates from LRU. For these applications LRU is not the optimal replacement policy; deviating from LRU therefore improves their performance. The second is that, for most of the other applications, replacing pages up to 30% or more

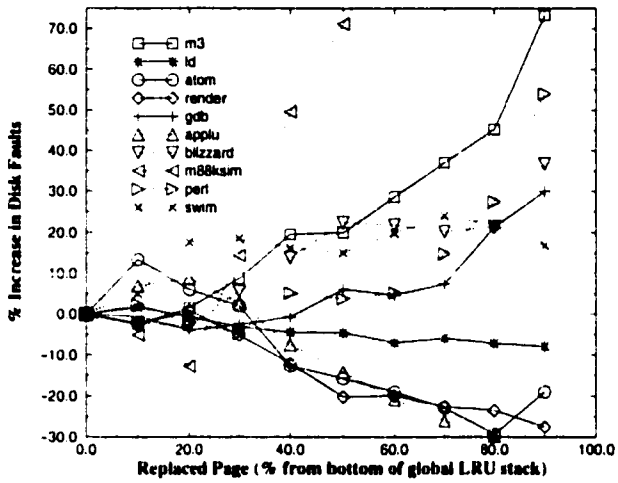


Figure 3.6: Effects of suboptimal LRU.

from the bottom of the LRU stack does not significantly degrade performance. With m3, for example, replacing pages 30% up from the bottom of the stack only introduces 9% more disk faults (which translates to an increase in execution time of only 2%).

These results indicate that application performance should not significantly degrade from deviations from strict LRU due to load balancing. Note, though, that one application, m88ksim, is sensitive to bad replacements once pages are persistently replaced at least 25% up the global LRU stack. I conclude from this that there is a limit to which LRU ordering can be ignored, but that some deviation from LRU will not cause significant performance degradation.

Naive Load Balancing

Given the results of the previous experiments, a reasonable question one might ask is whether a cooperative caching system could just ignore page age information and naively balance putpage requests across the available idle nodes. After all, such an approach should lead to a relatively uniform distribution of oldest pages across the idle nodes in the long run. I investigate a possible drawback of that approach, related to suboptimal use of global memory when a node transitions to the idle state, in the following experiment.

To quantify the effect of load balancing when new idle nodes join the cluster, I simulated a

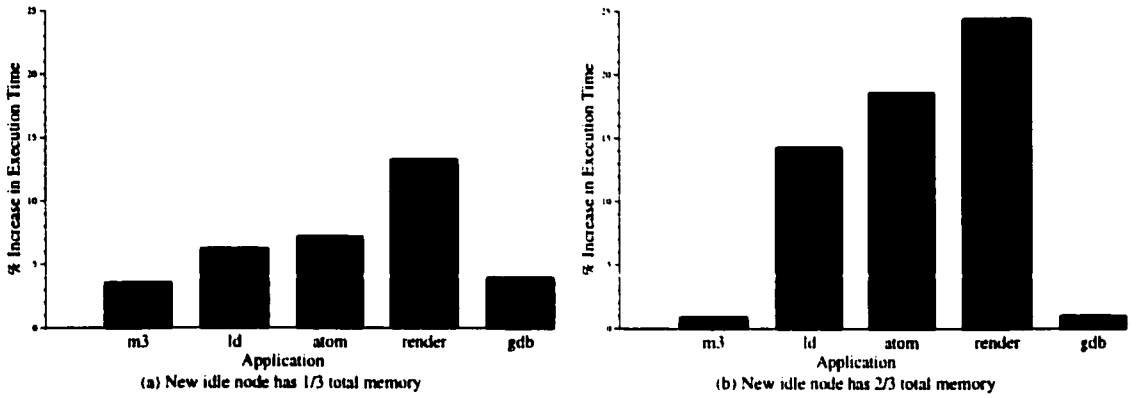


Figure 3.7: Upper bound on the increase in execution time for naively balancing the load across a new idle node, as compared with global LRU.

system with one global node running an application and three idle nodes being used as memory servers. For the first third of the execution time, only two of the idle nodes are used by the global node. Then, the third idle node joins the system and the application can begin to use its idle memory.

I use two different policies for introducing the memory on the new idle node into the system. In the first policy, the idle memory is made immediately available to the application; this policy corresponds to strict LRU (the new memory is the globally oldest memory). In the second policy, the idle memory is made available to the application at a fixed rate. In this case, every third fault goes to the new idle node; this policy corresponds to strict load balancing (the new idle node receives one third of the requests).

I then measure the increase in execution time of the second policy over the first to compare the effect of load balancing new memory into the system. Note that one would expect contention at the new idle node to be significantly higher for the LRU policy than for the load balancing policy, but that contention is not modeled in the simulator. Therefore, this experiment is conservative in that the increase in application execution time would be less if contention were modeled.

The results of the experiment are shown in Figure 3.7. Each graph has a bar for each of the simulated GMS applications, and the height of the bar shows the percent increase in execution time as a result of using the load balancing policy for new idle nodes. The two graphs each correspond to a different cluster configuration. In the first configuration, the global node holds 1/3 of the memory required by the application, the first two idle nodes each hold 1/6, and the new, third idle node holds

the remaining 1/3. In the second configuration, the global node and the first two idle nodes each hold 1/9 of the total memory required, and the new, third idle node holds the remaining 2/3 of the memory.

Although this experiment measures maximum possible increase in execution time, rather than the actual increase that would occur in the presence of contention, it appears that naive load balancing does not use the memory available at a new node aggressively enough. One could also expect that other short term concentrations of old pages would likewise not be replaced aggressively enough under this naive load balancing policy. Thus, although global LRU algorithms might be improved by load balancing global memory requests, indications are that they should not be replaced with a load balancing strategy that ignores page age information altogether.

3.5 The Cost of Global Memory on Local Nodes

In a cooperative caching system node, the balance between local and global pages changes dynamically. A key issue, then, is how and when to change that balance. In GMS, the balance depends on the ages of the pages on a node relative to the ages of (replaced) pages on other nodes. However, there is a tension that is not considered in this calculation, namely, the CPU time cost to a node that is storing global pages.

This section explores the performance issues involved in using local nodes (in addition to idle nodes) as memory servers. Using local nodes as memory servers can benefit global jobs in two ways: by making more global memory available, and by providing more memory servers to help distribute load. Previous work has quantified the benefit of the increased global memory [Feeley et al. 95], so I concentrate here on the benefit of having more servers with which to distribute and balance the remote memory request load.

Using the analytic model, I perform an experiment on a system composed of 10 idle nodes, 10 local nodes, and a variable number of global nodes. I vary the number of global nodes (N_{global}) in the network to vary the request rates from global nodes to the idle and local nodes. I then partition the load from the requests between the idle nodes and the local nodes such that a request will go to a local node with probability P_{local}, and a request will go to an idle node with probability $P_{idle} = 1 - P_{local}$.

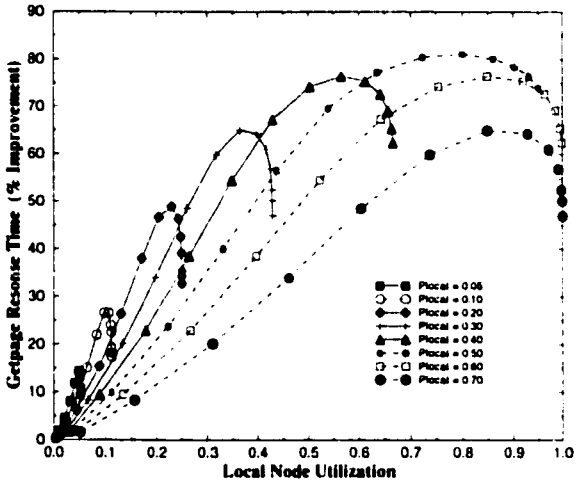


Figure 3.8: Percent improvement in the response time of getpage requests when a local node serves a fraction P_{local} of all global requests versus the utilization of the local node. Symbols are placed on the curves for each value of N_{global}, which ranges from 1 to 36 for each curve. The curves are solid when local nodes are given less precedence than idle nodes ($P_{local} < 0.50$), and are dashed when given equal or higher precedence ($P_{local} >= 0.50$).

In this experiment, I characterize the benefit of using local nodes to distribute load in terms of the improvement in response time of getpage requests, and I characterize the cost in terms of the utilizations on the local nodes. For a given $P_{local} = p$, I first solve the model with $P_{local} = 0$ and derive the response time of getpage requests. I then solve the model with $P_{local} = p$ and derive both the response time of getpage requests and the utilization of the local node. From the two response times, I then calculate the percent improvement in getpage response time at the cost of a particular utilization of the local node.

Figure 3.8 shows the results of this experiment when P_{local} is incrementally varied from 0.05 to 0.70. Symbols are placed on the curves for each value of N_{global}, which ranges from 1 to 36 for each curve in increments of 3 (except for the first increment, which is by 2). I again used the inputs derived from the OO7 benchmark to parameterize the request rates from global nodes.

From the figure I see that, for each P_{local}, improvement in response time is dramatic as N_{global} increases and the local node starts to shoulder the burden of global requests. However, improvement peaks at around 18 global nodes for our inputs, and then starts to decrease. Improvement decreases

after 18 global nodes because this is the region where the idle node response time curve for $P_{local} = p$ enters its heavy load region. Since the idle node response time curve for $P_{local} = 0$ is already in heavy load, the differences between the curves, and therefore the percent improvement, decreases as the number of global nodes increases.

The surprising result from this experiment is that the utilization of the local node approaches an asymptotic value as the number of global nodes in the network increases. In other words, for a given P_{local}, the utilization of the local node by global requests will never go above a constant value, no matter what the rate of requests might be.

The intuitive reason why the local node utilization asymptotes at a value less than 1.0 when $P_{local} < 0.5$ is due to the fact that the arrival rate at local nodes will always be less than the arrival rate at idle nodes. Therefore, when the arrival rate at the idle nodes reaches a maximum, the arrival rate to local nodes also reaches a maximum, but a maximum that is smaller than the one for idle nodes. Since idle and local nodes have the same service demand, the smaller arrival rate to local nodes is not enough to fully utilize the local nodes, and so the utilization maximizes at a value less than 1.0. Once $P_{local} >= 0.5$, though, the rate to local nodes is equal to or greater than the rate to idle nodes, and the local node utilization increases to 1.0.

3.6 A New Family of Global Page Replacement Algorithms

In this section I use the insights gained from the experiments in section 3.4 to design a new family of global page replacement algorithms that incorporate load considerations as well as age information. The goal is to achieve a close approximation to global LRU replacement while bounding the global memory traffic load on local nodes and balancing the global memory traffic across the idle nodes.

I begin by pointing out that the ideal target for balancing the load across the idle nodes is not necessarily equal global request traffic at each node; in order to fully utilize the available global memory, the target relative load for each idle node is instead proportional to the total amount of memory on the node. In many systems the amount of memory per node will not vary greatly, and in any case I assume that the higher relative load at the servers that have more memory is generally outweighed by the advantage of caching as many recently used pages as possible in global memory. The new algorithms can be easily modified if experience were to prove otherwise. For now, it is

Table 3.3: Notation used in discussion of new family of algorithms.

Symbol	Meaning
M_i	total memory available at node i, in pages
G_i	number of pages at node i that can be replaced without harming local jobs, if any
a_i	the age weight for node i, (i.e., the fraction of globally oldest pages at node i)
w_i	the target weight for node i, (i.e., node i share of the balanced load)
f_i	selection frequency for node i
b	load balancing rate
T_i	recent average time between global requests issued by node i
S	average time to serve one global memory request at a server node
N_g	current number global nodes
U_i^*	upper bound on the utilization of local node i by global memory requests

the additional unevenness in short term loads caused by (persistent) high concentrations of same-aged pages in one or more servers, as discussed in section 3.4.1, that the new class of global page replacement algorithms seeks to minimize.

I also point out that if the global replacement policy uses only the target relative loads (or weights) when selecting servers for global page replacement, then in the long-term the weights will reflect the relative proportion of pages of a given age at each node, and thus replacing the locally oldest page at the selected server node should be a close approximation of global LRU. On the other hand, the results of section 3.4.3 showed that strict use of target relative loads is not aggressive enough in taking advantage of temporary concentrations of very old global pages, such as the pages that become available when a node transitions from global or local to idle or when the overall load on a local node decreases. What's needed is an algorithm that uses age information to select those pages aggressively, and yet also uses target relative loads to balance the concentrations of same-age pages over time. As well, the weights for local and idle nodes should be continuously bounded to protect local jobs running on the local nodes and to avoid overloading any given idle node.

With these factors in mind, I propose a new family of global page replacement algorithms. I first describe the algorithms assuming that all required global system parameters are known and that all global nodes use an equal share of the global memory capacity. I then describe how global nodes

that have greater need for global memory can temporarily "borrow" server capacity from global nodes that have lower need. Finally I briefly discuss how the global parameters are communicated.

3.6.1 The Basic Global Page Replacement Algorithm

Let M_i denote the total amount of memory on node i, measured in pages, and let G_i denote the number of pages on node i that can be replaced without degrading the performance of the local jobs on i, if any. For an idle node, $G_i = M_i$; for a local node, $0 \leq G_i < M_i$. A global node has $G_i = 0$. The *target weight* for node i, w_i, equals $G_i / \sum G_j$, where the summation is over each node j in the system. This is the desired relative frequency of selecting node i for global page replacement based solely on load balancing considerations. The notation for the proposed page replacement algorithm is summarized in Table 3.3.

Regarding global page age information, two schemes have previously been proposed. In GMS, a dynamically selected *initiator* node periodically collects page age information from each node in the network and computes a set of weights that reflect the fraction of the M globally oldest pages that reside at each node. (M is an adjustable parameter of the GMS algorithm.) These weights, which I will call *age weights*, are then distributed to each node and used to select nodes for putpage requests, thus implementing approximate global LRU page replacement. Alternatively, in the global page replacement algorithm proposed by Dahlin and Sarkar [Sarkar et al. 96], each memory server piggybacks the age of its oldest page on its responses to global memory requests. In that algorithm, each global node maintains a list of the most recent page age information it has received from each server, sorted in order of decreasing age. The global node then selects the memory server at the top of the list for its next global page replacement. I choose to use age weights rather than simply the age of the oldest page on each server, as the weights are more easily adapted to accommodate load-balancing considerations. However, I leave open the possibility that the weights might be estimated using global page age information that can be piggybacked on responses to global memory requests. This is discussed further in section 3.6.3 below.

When a global node needs to replace a page in its local memory, it first chooses a memory server based upon its current age weights (a_i) and current target weights (w_i) for the memory servers, as follows. Let b denote a parameter of the algorithm called the desired ***load balancing rate*** for the

system, $0 \leq b \leq 1$. Node i is selected for the global page replacement with frequency

$$f_i = bw_i + (1 - b)a_i. \tag{3.1}$$

If $b = 0$ the algorithm approximates global LRU replacement without regard to load balance, as in GMS. If $b = 1$ the algorithm simply balances global requests across the available memory servers without regard to age information. For in between values of b the algorithm responds to new concentrations of globally oldest pages but gradually balances the number of similar age pages across the servers. The closer b is to zero, the more aggressively it will take advantage of newly idle memory. The closer b is to one, the more aggressively it will balance the global memory load.

Two additional constraints, in the form of upper bounds, are imposed on the memory server selection frequencies for global page replacement. First, for local nodes, I derive a simple bound to protect the local jobs running on the server from excessive interference by global memory requests. Let U_i^* denote an upper bound on utilization of local node i by global memory requests, and S denote the average time to serve a global memory request. Assuming each of the N_g global nodes uses an equal share of the total allowed server utilization, then the maximum rate that a global node can send global memory requests to local node i is $\frac{1}{S}\frac{U_i^*}{N_g}$. Thus, the following equation defines an upper bound on the selection frequency for local node i by global node j:

$$\frac{f_i}{T_j} \leq \frac{U_i^*}{N_g S}. \tag{3.2}$$

where T_j is the recent average time between the global memory requests issued by global node j. Multiplying each side of the equation by T_j gives the desired upper bound on f_i.

For each idle node, I derive a simple upper bound on the selection frequency to avoid overloading the node. In this case, I observe that the maximum utilization of the idle node by global memory requests is 100%. Plugging in 100% for U_i^* in equation 3.2 yields the following bound on selection frequency for idle nodes:

$$\frac{f_i}{T_j} \leq \frac{1}{N_g S}. \tag{3.3}$$

This assumes again that all global nodes use an equal share of the global memory capacity. The bounds can be modified for unequal use of memory capacity, as described below. Note that the above bound is simpler and somewhat more aggressive than the bound that would be derived by setting

estimated memory server capacity ($N^* \leq N_g$) and using standard asymptotic bounds techniques to estimate $N*$. [Lazowska et al. 84]

For each memory server, if the applicable upper bound on selection frequency is lower than the selection frequency originally computed from the target and age weights, then the selection frequency is set to the bound. Selection frequencies are then recursively recomputed for the other nodes using new target and age weights that are renormalized for the smaller set of server nodes that can be selected as well as for the remaining total selection frequency needed.

3.6.2 Borrowing Memory Capacity

The algorithm outlined above assumes that each global node uses an equal fraction of the total global memory capacity. In practice, however, this will rarely be the case. At any given point time, there is likely to be great variability in the total request rate for global memory among the global nodes. The basic algorithm approximates LRU while balancing server load even for this case, but the upper bounds on the selection frequencies will limit the throughput of the applications that have the greatest memory demands. I can improve the performance of those large applications by allowing them to "borrow" server capacity that would otherwise go unused by the global nodes that have lower global memory demands, as follows.

Define the unused capacity of a server node, U_e, i, to be the difference between its maximum possible utilization by global memory requests, U_i^* or 1.0 depending on the node type, and its recent utilization by global memory requests. Let each server node piggyback the maximum of this value and zero on each response it sends to a global memory request. Each global node can maintain the latest value of U_e, i that it has received from each server node i. The global node can then add the following term to the right hand side of equation 3.2 to increase the bound on its selection frequency for node i: $U_e, i/(N_g - 1)$. Note that this term assumes that the excess capacity will be shared equally with (at most) $N_g - 2$ other global nodes in the system. If one or more global nodes in the system continue to require less than their fair share of the global memory server capacity, the nodes that require more than their fair share will gradually grab the unused capacity. Note also that if the applications that haven't been using their fair share every suddenly require their fair share (or more), they are guaranteed to get at least their fair share immediately.

3.6.3 *Parameters of the New Page Replacement Algorithms*

Table 3.3 gives the parameters used in the new global page replacement algorithms. In this section I briefly discuss how the global nodes might determine the parameters that are needed to compute the memory server selection frequencies.

The total memory available at node i (M_i) and the average time to serve one global memory request (S) are system parameters that change infrequently and are easy to provide as inputs to the algorithm. Similarly, each global node j can easily measure the recent average time between the global requests it issues (T_i).

The current number of global nodes (N_g) can be determined by a dynamically selected (and possibly replicated) "status node", or can be estimated by each memory server and piggybacked on responses to global memory requests.

The desired load balancing rate (b) is a tunable parameter for the algorithm. Future research includes determining good values of this parameter for various workloads. It may even be useful to investigate the viability of dynamically tuning b at run-time using measured impact on global memory paging rates.

The upper bound on the utilization of local node i by global memory requests (U_i^*) might simply be specified by the system administrator, or might be adjusted according to the measured recent utilization of node i by its local jobs. Again this is the subject of future research.

The number of pages at node i that can be replaced without harming local jobs, if any, (G_i) is equal to M_i for idle nodes. For local nodes, further research is needed to determine precisely how this value should be estimated.

The age weights (a_i) can be determined by an initiator node, as in GMS, or might instead be estimated from global page age information that is piggybacked on responses from global memory servers. In this latter case, for example, the memory server might report the age of its xth oldest page, where x is a parameter of the algorithm. Each global node would maintain a table of the age reported by each memory server. The age weight for node i would then be estimated as the reported age of the xth oldest page on node i, divided by the sum of the reported ages of the xth oldest page on each other node.

3.7 Conclusions

High-speed switched networks have reduced the latency of network page transfers to below that of local disk. This technology change has made cooperative caching systems practical. A crucial issue in the implementation of cooperative caching systems is the selection of the target nodes to receive evicted pages. Existing systems have selected targets based on the ages of their memory pages, effectively selecting the oldest pages in the network for replacement. In this chapter, I study the impact of this decision, particularly on the nodes holding old pages; these nodes become global memory servers for highly-active nodes, which may interfere with their local application performance. I also examine (1) the impact of changing the target node selection criteria in order to reduce interference on local performance, and (2) the effect of deviating from the global LRU replacement algorithm used in some cooperative caching systems.

Using an analytic queueing network model, I studied how uneven request distributions produce memory server contention, even when the cluster has a large number of servers. Load balancing requests to produce an even distribution across servers minimizes server contention and improves request latency. With the model, I have also shown that cooperative caching systems will not scale if there are only a few memory servers available. However, when requests are load balanced across servers, I find that eight memory servers are sufficient to handle the load in the 70 node cluster I modeled.

Using a trace-driven simulator, I have demonstrated that the global LRU page replacement decision is relatively *insensitive* to exactly which of the oldest pages is replaced. Therefore, load balancing global page replacements should not degrade application performance if the deviation from strict LRU is moderate.

Overall, I conclude from these experiments with the model and simulator that load balancing global memory requests can significantly reduce contention at memory servers and substantially improve application performance. This conclusion led me to design of a new family of cooperative caching memory management algorithms that use a small set of measured system parameters, as well as two tunable parameters (i.e., the load balancing rate, b, and the stack location for exchange of global page age information, x) to balance the load across the servers while still implementing approximate LRU global page replacement.

Chapter 4

Cooperative Prefetching and Caching

This chapter presents *cooperative prefetching and caching* — the use of network-wide global resources (memories, CPUs, and disks) to support prefetching and caching in the presence of hints of future demands. Cooperative prefetching and caching effectively unites disk-latency reduction techniques from three lines of research: prefetching algorithms, cluster-wide memory management, and parallel I/O. When used together, these techniques greatly increase the power of prefetching relative to a conventional (non-global-memory) system.

In a cooperative caching system, a node has three opportunities for data prefetching: (1) from its disk into its local memory, (2) from disk into global memory (i.e., the disk and memory of *another* node), and (3) from global memory into local memory over the network. While these opportunities for using network memory for prefetching are conceptually straightforward, their use raises a number of interesting questions. For example, how do nodes *globally* choose the pages in which to prefetch from the global memory pool? When should data be prefetched and to what level of the storage hierarchy? When should pages be moved from global memory to local memory? How do we trade off the use of global memory pages for prefetching versus the use of those frames to hold evicted VM and file pages for non-prefetching applications? And finally, how do we value each page in the network, in order to best utilize each page frame?

To explore the answers to these questions, in this chapter I present the design of a cooperative prefetching and caching system whose goal is to minimize the total cost of all memory references in the cluster. For issuing prefetches, the system uses a hybrid prefetching algorithm that combines aggressive prefetching into global memory with more conservative prefetching into local memory. For making replacement decisions in the cooperative cache, the system uses a provably near-optimal variant of the classic Longest Forward Distance algorithm extended to a three-level memory hierarchy.

I also present an implementation of this design in a prototype cooperative prefetching and caching system called PGMS (Prefetching Global Memory System). I have implemented this prototype in the Digital Unix operating system, and evaluated it on a cluster of DEC Alpha workstations connected by a 1.28 Gb/sec Myrinet network.

My evaluation shows that by using available global resources, cooperative prefetching can obtain significant speedups for I/O-bound programs. For example, for a graphics rendering application, PGMS achieves a speedup of 4.9 over a non-prefetching version of the same program, and a 3.1-fold improvement over the same program using conventional, local-disk prefetching alone.

4.1 Introduction

Cooperative caching benefits memory-intensive applications, applications whose virtual memory and file buffer cache working sets are larger than the memory resources of a single node. By caching evicted pages in the remote memories of idle nodes instead of on local disk, cooperative caching transforms slow disk faults into much faster network faults. However, I/O-bound applications, either with large working sets or a small degree of reuse, only benefit to a limited degree from cooperative caching. For example, one application executing on the GMS cooperative caching system achieved a speedup of 1.7 when the system had enough idle nodes to contain its working set [Feeley et al. 95]. However, it still spent 80% of its execution time stalling on disk I/Os as it loaded its working set from disk into memory. For this application, adding more nodes to the system would not improve its performance because it was limited by the I/O time required to load all of its data into the system-wide cache. Instead, additional techniques are required to improve the performance of these kinds of I/O-bound applications running on cooperative caching systems.

Cooperative caching is just one approach for reducing disk stall time. Recent research has focused on two others. One approach is the development of algorithms for prefetching data from disk into memory to avoid stalls [Cao et al. 96, Patterson et al. 95, Kimbrel et al. 96b, Tomkins et al. 97, Rochberg et al. 97]. These prefetching algorithms use programmer annotations [Patterson et al. 95] or compiler-generated hints [Mowry et al. 96], speculative execution [Chang et al. 99], or markov prediction [Bartels et al. 99] to enable the system to predict future accesses and overlap disk I/O with execution. The second approach is the striping of files over multiple disks [Patterson et al. 88,

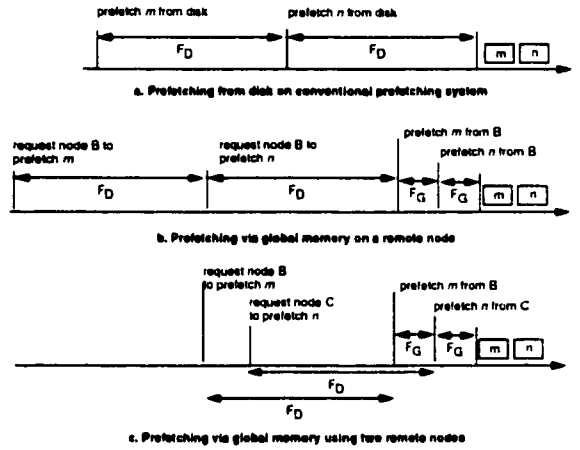


Figure 4.1: Prefetching in conventional and global-memory systems

Cao et al. 93] or the use of multiple network nodes in distributed file systems [Hartman et al. 95, Anderson et al. 96, Thekkath et al. 97, Gibson et al. 97] and distributed storage systems [Lee et al. 96, Hartman et al. 99, Gibson et al. 99] to access disks in parallel, dramatically increasing I/O throughput.

To further improve the performance of cooperative caching systems, this chapter proposes the synthesis of all three approaches for reducing disk stall time into a novel *cooperative prefetching and caching* system — the use of network-wide memory to support prefetching, caching, and parallel I/O in the presence of optional hints of future demands. The synthesis of these approaches results in a system that is significantly different than one using any single approach alone. In the presence of global memory, a node has three choices for data prefetching: (1) from disk into local memory, (2) from disk into global memory (i.e., the disk and memory of *another* node), and (3) from global memory into local memory. When used together, these options greatly increase the power of prefetching relative to a conventional, non-cooperative-caching system.

For example, Figure 4.1a shows a simplified view of a conventional prefetching system. Node A issues prefetch requests to missing blocks m and n in advance, so that both blocks are available in memory just in time for the data references. In this case, buffers must be freed on node A for blocks m and n about $2F_D$ and F_D in advance of their use, respectively, where F_D is the disk fetch time. There are two possible problems with this scheme. First, node A's disk may not be free in

time to prefetch these blocks without stalling. Second, if prefetched early enough to avoid stalling, blocks m and n may replace useful data, causing an increase in misses; whether or not this happens depends on how far in advance the data is prefetched (which depends on F_D) and the access pattern of the program.

In contrast, Figures 4.1b and 4.1c show two examples of prefetching in a global-memory system. These scenarios show that combining prefetching and global memory has several possible advantages:

- A prefetching node can greatly delay its final load request for data that resides in global memory, thereby reducing the chance of replacing useful local data. In Figure 4.1b, for example, node A requests that node B prefetch pages from disk into B's memory ahead of time. As a result, node A need not free a buffer for the prefetched data until F_G (the time for a page fetch from global memory) before its use. On a 1Gb/sec network, such as Myrinet, F_G may be up to 50 times smaller than F_D [Yocum et al. 97], so this difference is substantial.

- The I/O bandwidth available to a single node is ultimately limited by its I/O subsystem – in most cases, the disk subsystem. However, using idle nodes to prefetch data into global memory greatly increases the available I/O bandwidth by adding in parallel: (1) the bandwidth of the network, (2) the bandwidth of remote disk subsystems, and (3) the execution power of the remote CPUs. Use of this parallelism for the global prefetching shown in Figures 4.1b and 4.1c effectively reduces page prefetch time for I/O-bound processes from F_D to F_G.

- Figure 4.1c shows that distributing prefetch requests among multiple nodes in parallel allows those nodes to delay their own buffer replacement decisions (in this case, node B benefits relative to Figure 4.1b), thereby making more effective use of their memories.

- With the high ratio of disk latency to global memory latency, a highly-conservative process could choose to prefetch *only* into global memory; the process would fault on reference to a non-resident page, but would still benefit from the 50-fold reduction in fault time.

- Given that there is idle memory and CPU power in the network, a process could afford to prefetch *speculatively*, using idle global pages as the speculative prefetch cache.

While the idea of using network memory for prefetching is conceptually straightforward, it raises a number of questions. For example, how do nodes *globally* choose the pages in which to prefetch from the global memory pool? When should data be prefetched and to what level of the storage hierarchy? When should pages be moved from global memory to local memory? How do we trade off the use of global memory pages for prefetching versus the use of those frames to hold evicted VM and file pages for non-prefetching applications? And finally, how do we value each page in the network, in order to best utilize each page frame?

To answer these questions, I have developed an algorithm that combines cooperative caching with prefetching. I have implemented the algorithm in the DEC UNIX operating system, and evaluated it on a collection of DEC Alpha workstations connected by a Myrinet high-speed switched network. This system, called PGMS (Prefetching Global Memory System), integrates all cluster memory use, including VM pages, mapped files, and file system buffers, for both prefetching and non-prefetching applications. It effectively unites techniques from three previous lines of research: prefetching algorithms, including the work of [Patterson et al. 95], [Cao et al. 96], [Kimbrel et al. 96b], and [Tomkins et al. 97]; the global memory system (GMS) of [Feeley et al. 95]; and the use of multiple network nodes for parallel I/O, as in the Zebra [Hartman et al. 95], xFS [Anderson et al. 96], and Frangipani [Thekkath et al. 97] distributed file systems, and the Petal [Lee et al. 96], Swarm [Hartman et al. 99], and NASD [Gibson et al. 97] distributed storage systems.

My evaluation of PGMS executing on the Alpha-based cluster shows that prefetching in a global memory system can produce substantial performance advantages for I/O-bound programs. For example, a data-intensive graphics rendering application on a 5-node cluster achieves a speedup of 4.9 when running on PGMS compared to running on a single node alone. In addition, I show that:

- The use of prefetching significantly increases application performance beyond the use of cooperative caching alone. For example, the same rendering application using prefetching on PGMS achieves a speedup of 2.8 over a version of the same program that just uses cooperative caching.

- Global prefetching provides substantial speedup over local, conventional prefetching alone.

For example, the rendering application using global prefetching achieves a speedup of 3.1 over a version that only uses local prefetching.

- Parallel disk prefetching leverages the aggregate disk I/O bandwidth in the cluster, scaling application performance with cluster size.

- Prefetching over the network from global to local memory has much less impact than disk prefetching. For the rendering application, for example, prefetching over the network only improved performance by 5%.

- The system isolates prefetching and non-prefetching applications, allocating system resources among them fairly.

The remainder of the chapter is organized as follows. Section 4.2 presents the PGMS algorithm for prefetching in global memory. Section 4.3 describes the implementation of the algorithm in the DEC UNIX operating system. Section 4.4 presents performance results from the prototype. Finally, Section 4.5 concludes.

4.2 The Global Prefetching and Caching Algorithm

This section presents the idealized global prefetching and caching algorithm that is the basis of PGMS; Section 4.3 describes how the PGMS implementation efficiently approximates the algorithm. For the purposes of defining the algorithm, I make several simplifying assumptions. First, I assume a uniform cluster topology with network page transfer cost (F_G) independent of location. Second, I assume uniform availability of disk-resident data to all nodes (e.g., through a network-attached disk [Gibson et al. 97] or replicated file system [Lee et al. 96]) and uniform page transfer time from disk into a node's memory (F_D). For cluster systems using high-speed switched networks, F_G will be significantly smaller than F_D. Third, I assume a centralized algorithm with complete *a priori* knowledge of the reference streams of the applications running on all nodes, including the pages to be referenced, the relative order in which they are referenced, and the inter-reference times.

I begin with a description of the algorithm below. In Section 4.2.3, I discuss the theory motivating my design.

4.2.1 Design principles

The goal of my design is to minimize average memory reference time across all processes in the cluster. This goal requires that "optimal" prefetching and caching decisions be made both for individual processes and for the cluster as a whole. The algorithm I use in PGMS has two basic objectives:

- To reduce disk I/Os, maintain in the cluster's global memory the set of pages that will be referenced nearest in the future.
- To reduce stalls, bring each page in the cluster to the node that will reference it in advance of the access.

4.2.2 Detailed description

In this discussion, I use the term *local page* for a page that is resident on a node using that page, and the term *global page* for a page that is cached by one node on behalf of another. A reference to a global page thus requires a network transfer.

To meet its objectives PGMS must make decisions about both prefetching and cache replacement. Furthermore, the system must make (1) global decisions about which pages to keep in global memory rather than on disk, and (2) local decisions about which data to keep resident in a node's local memory rather than in global memory. The PGMS algorithm thus implements four interrelated policies:

- local cache replacement (transfer of pages from a node's local memory to global memory),
- global cache replacement (eviction from global memory),
- local prefetching (disk-to-local and global-to-local), and
- global prefetching (disk-to-global).

For cache replacement, I modify the algorithms used by GMS [Feeley et al. 95] to incorporate prefetching. For local cache replacement, I first choose to forward to global memory a global page

on the local node (i.e., a page held on behalf of another node); if there is no global page, I choose the local page whose next reference is furthest in the future. For global cache replacement in PGMS, I evict the page in the cluster whose next reference is furthest in the future.

For prefetching decisions I apply a hybrid algorithm, whose goal is to be conservative locally but aggressive with resources on idle nodes. For local prefetching, I adapt the Forestall algorithm of Kimbrel, et al. [Kimbrel et al. 96b, Tomkins et al. 97]. Forestall analyzes the future reference stream to determine whether the process is I/O constrained; if so, Forestall attempts to prefetch just early enough to avoid stalling. In my adaptation, I apply the Forestall algorithm to the node's local reference stream and take into account the different access times for network-resident (global) and disk-resident data. This analysis leads to a *prefetch predicate*; when the prefetch predicate is true, Forestall recommends that a page be prefetched either from global memory or from local disk. Whether the page is actually prefetched depends on whether a resident page can be found whose next reference is further in the future.

For prefetching into global memory (disk-to-global) PGMS uses the Aggressive algorithm of Cao et al. [Cao et al. 95]. If a page on disk will be referenced earlier than a page in cluster memory, then the disk page is prefetched. To make room, the global eviction policy chooses for replacement the page (in the cluster) whose next reference is furthest in the future.

Computing the local prefetch predicate

The local prefetch predicate indicates when prefetching is needed to avoid additional stalls. In the predicate computation, I assume that all prefetches into a node and all memory references at a node are serialized.

Consider the hinted future reference stream on a node P at a given time T, and let $b[i]$ be the i-th missing page in the hinted reference stream that will be accessed after time T. (Missing pages at time T are those pages that are not in P's local memory or in the process of being prefetched into P's local memory at time T.) Let $t_{b[i]}$ be the time between T and the next access to $b[i]$, assuming no stalls occur between T and this access. Let F_i be the time that will be required to fetch $b[i]$ into local memory: F_i equals F_G (the time to perform a network fetch) if $b[i]$ is currently in global memory and equals F_D (the time to fetch from disk) otherwise. Under these assumptions, the system can

readily calculate whether or not it needs to begin prefetching immediately in order to avoid stalling: prefetching is not yet required if for each j, the time to fetch the first j missing pages ($\sum_{1 \leq i \leq j} F_i$) is less than the time until the access to the j-th missing page ($t_{b[j]}$). Therefore, I define the local prefetch predicate to be true if there is some j for which $\sum_{1 \leq i \leq j} F_i \geq t_{b[j]}$. Whenever this prefetch predicate is true for some j, node P attempts to prefetch its first missing page.

4.2.3 Theoretical underpinnings

I begin with a discussion of cache replacement in a three-level memory hierarchy. Then I summarize theoretical results on prefetching as it pertains to PGMS. Finally, I touch upon the problem of buffer allocation among competing processes.

Cache replacement

The PGMS algorithm attempts to minimize the total cost of all memory references within the cluster. The cost of a memory reference depends on whether, at the time of reference, the data is in local memory, in global memory (on another node), or on disk. Typically, a local hit is more than three orders of magnitude faster than a global memory or disk access, while a global memory hit over a Gb/sec network is on the order of 50 times faster than a disk access.

It is well known that in a two-level memory hierarchy such as local memory and disk, the optimal replacement strategy is to replace the page whose next reference is furthest in the future [Belady 66]. The analogous replacement strategy for a three-level memory hierarchy (local memory, global memory, disk) such as PGMS is the Global Longest Forward Distance ($GLFD$) algorithm, defined formally as follows.

> On a reference by node A to page g in global memory on node G, bring g into A's memory, where it becomes a local page. In exchange, select a page on A for eviction: if A has a global page, send that page to G, where it remains a global page; otherwise if A has no global page, select the local page whose next reference is furthest in the future on A, and send that page to G, where it becomes a global page. On a reference by node A to page d on disk, read d into A's memory. In exchange, select (1) a page a on A for eviction to global memory, and (2) a page g in the cluster for eviction to disk. Select the page a on A for eviction using the same

method described above for a global memory reference. For the cluster-wide eviction, select page g (say on node G) whose next reference is furthest in the future, cluster-wide. Write g to disk, and send a to node G, where it becomes a global page.

The effect of this algorithm is to (1) maintain in the cluster as a whole the set of pages that will be accessed soonest and (2) maintain on each node the set of pages that will be accessed soonest by processes running on that node. While this algorithm is not always optimal, it is near optimal as shown by the following theorem:

Theorem

Consider a global memory system with local memory access cost F_L, global memory access cost F_G, and disk access cost F_D, where $F_L < F_G < F_D$. Let OPT be the offline page replacement algorithm minimizing total memory access cost. I denote by $C_{OPT}(R)$ the total memory access cost incurred by OPT on reference stream R, i.e.

$$C_{OPT}(R) = |R|F_L + O_G(R)F_G + O_D(R)F_D,$$

where $O_G(R)$ (resp. $O_D(R)$) denotes the number of global memory references (resp. disk references) made by OPT on R. Similarly, denote by $C_{GLFD}(R)$ the total memory access cost incurred by $GLFD$ on input R. Then for any R,

$$C_{GLFD}(R) \leq C_{OPT}(R)\,(1 + 3(F_G/F_D))\,.$$

The theorem implies that the $GLFD$ algorithm is near optimal whenever the ratio of network access time to disk access time is small. For example, in a fast network such as the Myrinet where $(F_G/F_D) \leq 0.02$, the total I/O overhead incurred by the $GLFD$ paging algorithm is within 6% of optimal. Therefore, in PGMS I use $GLFD$ as the cache replacement algorithm.

Prefetching strategy

Effective prefetching into local memory eliminates stall time while minimizing computational overhead. Previous studies of prefetching [Cao et al. 95, Cao et al. 96] have shown that for a fully-hinted

process with a single disk, the Aggressive prefetching algorithm achieves near-optimal reduction in stall time. Unfortunately, Aggressive's early prefetching may result in suboptimal replacements, which can increase the total number of I/Os performed. Although these I/Os are overlapped with computation, a significant overhead (the computational overhead of issuing fetches) can result. The Forestall algorithm has been shown in practice to match the reduction in I/O stall achieved by the Aggressive algorithm, while avoiding the computational overhead of performing unnecessary fetches [Kimbrel et al. 96b, Tomkins et al. 97]. Forestall is therefore the method of choice for local prefetching.

In contrast to local prefetching, disk stall time is much more important than computational overhead for disk-to-global prefetching, where the prefetching is performed by otherwise idle nodes. By analogy with the problem of prefetching from a single disk into a single memory [Cao et al. 95], the problem of prefetching from multiple disks into global memory, under the assumption that disk-resident data is available uniformly on all disks, can be shown to achieve near-optimal reduction in disk stall time[1]. Further, where the pages to be evicted will not be referenced until significantly later, if ever, aggressive prefetching's drawback of less accurate cache replacement decisions is relatively unimportant. Little harm results from displacing these pages aggressively in order to gain the benefits of prefetching[2].

There are two other important reasons to prefetch aggressively into global memory. First, pressure on *local* memory is significantly reduced through aggressive global prefetching. Indeed, the times at which the local prefetch predicate for a process is true depends directly on how many of the process' missing pages are in global memory (as opposed to disk); the greater the fraction of missing pages that are in global memory, the *later* the times at which the predicate will first be true. Delaying the times at which the local prefetch predicate is true allows better replacements to be made on a busy node running the hinted process, reducing unnecessary fetches and associated overhead on that node. Second, hinted processes cannot rely on access to the full CPU and disk bandwidth

[1]This is in sharp contrast to the case where different pages reside on different disks, in which case aggressive prefetching can be far from optimal.

[2]It should also be noted that in contrast to the results of [Tomkins et al. 97], a page cannot be prefetched into global memory and then evicted before it is referenced: a page chosen for prefetch into global memory is always the then soonest non-resident page to be referenced by any process in the cluster, so the next global fault will not occur until after that page is referenced.

of idle nodes, because of competition with other prefetching processes for these resources. Aggressive prefetching gives these processes some leeway for dealing with this uncertainty whereas more conservative global-memory prefetching could result in unnecessary stall.

Allocating buffers among competing processes

My assumption of complete advance knowledge of the combined reference streams of the applications in the cluster allows us to view each node as executing a single process. This simplifies the algorithm and conceptual framework. In practice, any prefetching system must allocate buffers among multiple independent processes with differing hint capabilities.

Policies for allocating buffers among competing processes on a single node have been extensively studied [Patterson et al. 95, Tomkins et al. 97, Cao et al. 96]. These studies show that proper buffer allocation among competing processes on a single node must consider working set sizes, hinted reference patterns, cache behavior of unhinted processes, variability of inter-reference CPU times between different processes, the prefetching and cache replacement policies used and processor scheduling. For example, the benefit of the prefetch recommendations made by hinted processes can be compared to the cost of LRU cache replacement decisions for unhinted processes [Patterson et al. 95, Tomkins et al. 97]. An interesting direction for future research is to analyze these algorithms in the context of a prefetching global memory system.

Processes on different nodes will also compete for global memory and prefetching resources. The prefetching system must similarly allocate resources among competing nodes. As noted above, the aggressive prefetching policy in PGMS reduces the impact of this competition on individual prefetching processes, both by reducing the likelihood of disk faults and by reducing the uncertainty resulting from independent competing prefetching requests.

4.2.4 Summary

This section has described an algorithm for prefetching and caching in global memory systems. PGMS prefetches into local memory conservatively (delaying as long as possible) and into global memory aggressively. The objective of this two-pronged scheme is to maintain valuable blocks in local memory, while sacrificing global blocks to speedup prefetching. I make this tradeoff because

it has been shown that in a global memory system performance is relatively insensitive to which of the oldest global pages are replaced [Voelker et al. 97]. Therefore, I replace the least valuable global pages in order to reduce stall time through prefetching, without risking local performance. *The ability to make this tradeoff is the key advantage of combining prefetching and global memory.*

4.3 Implementation

The previous section presented an idealized algorithm for prefetching in global memory systems. I now describe the prototype PGMS implementation that approximates the ideal algorithm for co-operative prefetching and caching. In brief, I implemented PGMS by starting with the Digital-UNIX-based GMS global memory system [Feeley et al. 95], adding prefetching support, and then implementing an approximation to the prefetching algorithm presented above. I begin by giving an overview of GMS for background.

4.3.1 Overview of GMS

GMS is a global memory system for a clustered network of workstations. The goal of GMS is to use global memory to increase the average performance of all cluster applications. Programs benefit from global memory in two ways. First, on a page replacement, the evicted page is sent to global memory rather than disk; a reload of that page may therefore occur much faster. Second, programs benefit from access to shared pages, which may be transferred over the network rather than from disk.

GMS is implemented as a low-level operating system layer, underneath both the file system and the virtual memory system. All *getpage* requests issued by both the VM and file systems to fetch pages from long-term storage, and all *putpage* requests issued to send pages to long-term storage, are intercepted by GMS. Each page in the GMS system has a network-wide unique ID, determined by its location on disk (i.e., the UID consists of the IP address, device number, inode number, and block offset). GMS maintains a distributed directory that when given the UID for a page, can locate that page in global memory, if it exists. The key structures of that database are: (1) a per-node *page-frame-directory* (PFD) that describes every page resident on the node, (2) a replicated *page-ownership-directory* (POD) that maps a page UID to a manager node responsible

for maintaining location information about that page, and (3) the *global-cache-directory* (GCD), a distributed cluster-wide data structure that maps a UID into the IP address of a node caching a particular page. On a *getpage* request, GMS finds the manager node for that page and sends a request to that node; the manager checks whether that page is cached in the network, and sends a message to the caching node if so, requesting that it transfer the page to the original requester.

4.3.2 Mechanisms for implementing prefetching

PGMS extends the GMS implementation in three key ways. First, PGMS adds operations for prefetching blocks into any node's memory from disk or global memory. Second, PGMS modifies GMS mechanisms for distributing and maintaining global page information. Third, the PGMS policy module follows a hinted application through its predicted reference stream and uses epoch-based prefetch information for scheduling prefetching operations. The remainder of this section describes these three key aspects of my implementation.

At the lowest level, prefetching in PGMS is handled by two operations. The *prefetch_to_local* operation prefetches a page into local memory – if that page is in global memory, the page is fetched over the network, otherwise it is read from local disk. The *prefetch_to_global* operation prefetches into global memory from the disk on the global node. In the current prototype, prefetched files are replicated on the disks of multiple cluster nodes, thus allowing each of these nodes to prefetch from the same file independently. (Alternatively, the files could be striped across the disks.)

PGMS stores file replication information in a distributed directory called the *file-alias-directory* (FAD). For each replicated file, the FAD contains an entry that lists the IP address and local file name for each node storing a replica. The FAD entry for a file is stored on the manager node for the file's blocks (the FAD entry for shared files is replicated on every manager node). The FAD serves two key purposes: (1) PGMS uses the FAD to pick the nodes used to prefetch a file, and (2) the FAD extends the GMS UID to permit on-disk replication. While pages in GMS are named by a UID, the UID does not allow a page to be replicated on multiple disk locations, because each copy would be given a different name. In PGMS, a replicated file is assigned a primary location that determines the UID assigned to each of its pages; the FAD is then used in conjunction with the UID to support aliasing of a file to multiple storage locations.

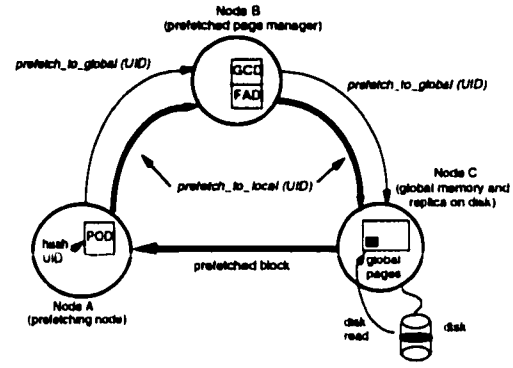


Figure 4.2: Communications for prefetch into global memory

Figure 4.2 shows the communications and data structures for the most general prefetch to global memory (a shared page). The thin arrows show the actions performed when node A issues a *prefetch_to_global* request. The request is first directed to the page's managing node, which knows if and where the page is cached in the network. If the page is not cached, PGMS picks the prefetching node from among the idle nodes that replicate the page's backing file. When a node receives a prefetch request, it reads the page from disk and caches the page in its own memory (as illustrated by node C in Figure 4.2).

The thick arrows show the actions performed later when the page is prefetched from global memory on node C to local memory on node A. Both actions must pass through the manager, because in the interim the global page may have moved.

4.3.3 Approximating the prefetching and caching algorithm

My goal in approximating the algorithm of Section 4.2 is to provide a reasonable tradeoff between accuracy and efficiency. The key issue is guaranteeing the validity of global knowledge used by the algorithm and deciding when it must be updated.

I give only a high-level description of the algorithm approximation here due to length considerations. My approach is similar to that used in GMS. The algorithm divides time into *epochs*, where each epoch has a maximum duration T (in practice, T is between 5 and 10 seconds). A coordinator node is responsible for collecting and distributing global information at the beginning of each

epoch; the coordinator for the next epoch is elected as the "least loaded" node at the start of the current epoch.

At the start of the epoch, each node sends to the coordinator its CPU load and a summary of its *buffer values*. The CPU load on a node is an estimate of the CPU utilization seen by locally running processes. The value of a buffer, or equivalently the value of a page, is an estimate of the time until the next reference to the page stored in that buffer. The time until the next reference to a page is estimated on a per-process basis as follows. Future inter-reference CPU time is estimated from inter-reference CPU times measured in the recent past scaled by the percentage of time that the process was scheduled on the processor. The estimated time until the next reference to a hinted page is then the number of hinted references preceding it multiplied by the estimated future inter-reference CPU time. For unhinted processes, the time until the next reference to a page is estimated to be the time since the previous reference.

Using the information collected and the recent rates of evictions and prefetches, the coordinator computes a weight w_i for each node, representing the number of buffers on node i that are candidates for replacement by global prefetch requests and putpages (evictions) from other nodes during the epoch. Nodes whose CPUs are fully utilized are assigned a w_i value of 0, regardless of whether or not they have buffers of low value. The coordinator also determines the maximum buffer value, *MaxValue*, that will be replaced in the new epoch. To start the epoch, the coordinator sends the weight vectors w_i, and the value *MaxValue*, to all nodes in the cluster. The epoch terminates when either (1) the duration of the epoch, T, has elapsed, (2) $\sum_i w_i$ global pages have been replaced, or (3) the buffer value information is detected to be inaccurate.

During the epoch, nodes perform replacement and prefetching as follows:

- **Replacements:** When a page on a node must be replaced, the node selects its least-valuable page, p, for eviction. The node then forwards p to node i, where p becomes a global page in i's memory, replacing i's least valuable page. The target node i is chosen with probability proportional to w_i/N ($N = \sum_i w_i$). (If p is a shared page and a copy exists in another node's local memory, then p is simply discarded.) Roughly then, over an epoch the system will replace the N least valuable pages in the network.

- **Prefetching into local memory:** For each node j, whenever the prefetch predicate for a

hinted process on j is true, and there are buffers on j of lower value than the ones that it wishes to consume for prefetching, the node issues prefetch requests, replacing its least-valuable pages.

- **Prefetching into global memory:** Processes running on each node, j, issue requests in round-robin order to other nodes k with $w_k > 0$ and request prefetches into global memory on their behalf. The request includes an estimate of the value of each page to be prefetched. To avoid global-memory thrashing, the system limits each node to B simultaneous outstanding disk prefetch requests, where B is a system parameter (I set $B = 8$ in the prototype). A node with fewer than B outstanding requests may issue a new batch of global prefetch requests.

  A given node k may receive several competing disk-to-global prefetch requests from other nodes. It initiates the prefetch request for the page p of highest value and, upon initiation of the prefetch, sends an acknowledgement to the requesting processor, say node j. This enables node j to update its hinted reference stream to indicate that p will soon be stored in global memory. The acknowledgement is also used to inform node j how many of the w_k low-value buffers have not yet been replaced.

The w_i values determine the rate at which global prefetch requests and evictions to global memory arrive at node i. I choose these values to meet two goals. First, I wish to minimize the overhead on each node due to global memory requests and to balance this load across the various nodes in the network. Load balance is important for global prefetching in order to achieve a high degree of I/O parallelism. Second, I wish to ensure that buffers supplied for eviction to global memory and global prefetching are of sufficiently low value, so that the system does not replace useful data. This is particularly important for limiting any negative impact from aggressive or speculative prefetching into global memory.

4.4 Performance Measurements and Analysis

This section presents the measured performance of the PGMS system. I begin with low-level measurements of the implementation, and then present the performance of the prefetching mechanisms

for several workloads. I then examine the performance characteristics of various aspects of PGMS in detail, using a rendering application as an example.

4.4.1 Experimental testbed

All measurements are from 266 MHz DEC AlphaStation 500 (Alcor) systems running Digital Unix 4.0, connected by a 1.28 Gb/s Myrinet network [Boden et al. 95]. All nodes in each experiment use M2F-PCI32 (LANai-4) Myrinet adapters attached to a full-crossbar SW8 Myrinet switch. The disks from which all experimentes are performed are 7200 RPM ST32171W Seagate Barricuda drives. Pages and file blocks are 8KB, and reading a random 8KB page from disk takes an average of 13ms.

For optimum network performance I used Trapeze [Yocum et al. 97, Anderson et al. 98] firmware for the Myrinet adapters. Trapeze uses an adaptive message pipelining technique called cut-through delivery [Yocum et al. 97] to minimize transfer latencies on the network in a manner similar to GMS subpages [Jamrozik et al. 96]. Using Trapeze, GMS can perform an 8KB page fault from remote memory in 165μs on platforms capable of delivering the full bandwidth of the 33 MHz 32-bit PCI bus. The Alcor is limited to 66 MB/s in the receiving direction, which increases raw page transfer times to 187μs. Including the PGMS overhead of generating a request and processing the reply, a global-to-local prefetch takes 370μs.

4.4.2 Microbenchmarks

Tables 4.1 and 4.2 detail the costs of the PGMS prefetching operations for the common case where the requester and manager are the same node. For a *prefetch_to_local* that returns a page from global memory, the times are divided into four components: issuing the prefetch request to the manager and target nodes (*Request*), processing the message and sending the prefetched page to the requester (*Prefetch*), receiving the page (*Receive*), and *Access*, the time to install the page in the local page map (as is normal for any read). In my PGMS experiments, all *putpage* and *getpage* operations (including *prefetch_to_local*) copy the page once on the client; these copies can be eliminated with an optimization [Anderson et al. 98]. The *prefetch_to_global* operation has two components: the time to generate the prefetch request (*Request*), and the time to initiate the disk request into remote

Table 4.1: Micro measurement of *prefetch_to_local()* operation (median times from 50 iterations). Note that, due to pipelining effects, the requester and prefetcher overlap part of their overheads.

Operation	Node	Time (μs)		
		CPU	Net	Total
Request	Requester	67.9	—	67.9
	Manager	53.9	35	88.9
Prefetch	Prefetcher	46.3	187	233.3
Receive	Requester	22.3	—	22.3
Access	Requester	153	—	153

Table 4.2: Micro measurement of the *prefetch_to_global()* operation (median times from 50 iterations). Note that the prefetch request sent to the prefetcher is an asynchronous operation.

Operation	Node	Time (μs)		
		CPU	Net	Total
Request	Requester	7.6	—	7.6
	Manager	40	35	75
Prefetch	Prefetcher	146	—	146

host memory and process it when it completes (*Prefetch*). Where appropriate, the values are broken into the CPU overhead for each node (*CPU*), the overhead and latency of communication (*Net*), and the total overhead and latency (*Total*).

4.4.3 Application-level performance of PGMS

To characterize the application-level performance of PGMS under a wide range of workloads, I hinted a number of synthetic benchmarks and real applications and evaluated their performance on the PGMS prototype:

OO7 is an object-oriented database benchmark that builds and traverses a parts-assembly database [Carey et al. 93]. The experiments traverse an existing 100MB database mapped into mem-

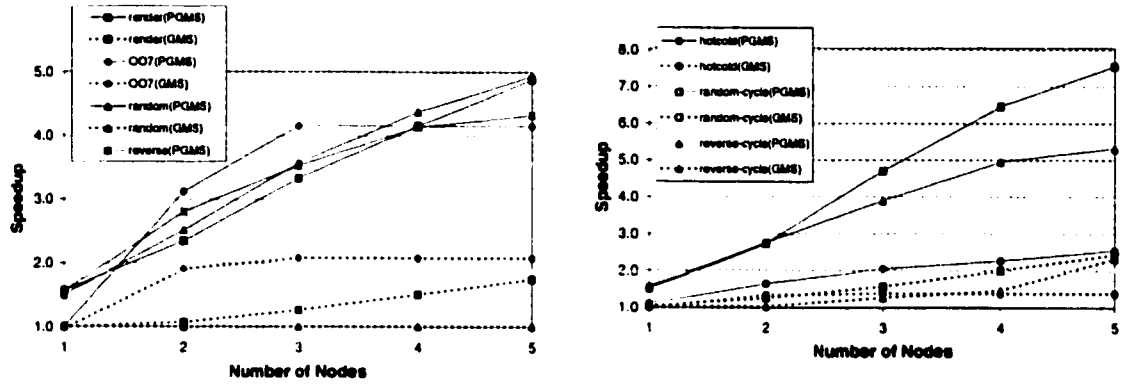


Figure 4.3: Application speedup on GMS and PGMS

ory, accessing approximately 65MB of data.

Render is a display engine that renders a computer-generated scene from a pre-computed 178MB database of tracing data [Chamberlain et al. 95]. The experiments perform a sequence of operations to move the viewpoint progressively closer to the scene without changing the viewpoint angle, accessing approximately 100MB of data.

Hotcold accesses 512 random pages from a 20MB "hot" mapped file region, then 256 random pages from a 80MB "cold" region, and repeats for 20 phases. Every group of four pages read are accessed sequentially.

Random randomly accesses all of the pages of a 100MB file.

Reverse sequentially reads a 100MB file in reverse.

The synthetic benchmarks first issue hints to inform the kernel of the pages or blocks they will access. In the experiments presented here, each benchmark computes for 250μs between each page or block access. To explore the effects of data reuse in the synthetic file benchmarks, I also measured "cyclic" variants that access their data sets three times before terminating.

Figure 4.3 shows the speedup of the synthetic benchmarks and applications on clusters ranging from 1 to 5 nodes. The top graph shows the speedup of the applications and synthetic benchmarks;

the bottom graph shows the speedup of the benchmarks run cyclically. The speedups are relative to performance on a single node with no global memory and no prefetching other than the Digital Unix readahead mechanism for files accessed sequentially. All programs run on a single node with 64MB of memory, about 32MB of which is available for prefetching and caching. The other nodes in the cluster are idle and act as memory and disk servers, each again with about 32MB of available global memory and a single disk. For comparison, I also show performance under GMS (dashed curves).

For these applications, PGMS achieves speedups of 4 to 7 on 5 nodes. Using the Render application as an example, the Y intercept on the Render curve (Number of Nodes = 1) is the speedup for a single isolated node that is prefetching from its local disk only. In this case, local-disk prefetching achieves a speedup of 1.6 over the non-prefetching version of the application. The 2-, 3-, 4-, and 5-node measurements show what happens as I add 1, 2, 3, and 4 idle nodes to the network, respectively. The curve demonstrates that PGMS is able to achieve nearly linear speedup when up to 4 idle nodes are used to service a memory-intensive application. With 4 idle nodes, PGMS achieves a speedup of 4.9 over a non-prefetching version of Render and 3.1 over a single node with local-disk prefetching. Although I was not able to perform the bulk of the experiments on a cluster larger than 5 nodes, I did investigate Render on larger clusters and found that its performance levels off at a speedup of 5.6 on 7 nodes.

Comparing the GMS and PGMS lines, GMS achieves moderate speedups when the benchmarks access their data cyclically, yielding hits in global memory as pages are reused. However, parallel prefetching allows PGMS to achieve significant speedup for both non-cyclic and cyclic data access.

A common workload absent from these graphs is a sequential benchmark. Digital Unix and the Alpha I/O hardware is heavily optimized for sequential access, doing clustered readahead prefetches that provide user-level bandwidths of up to 8.3MB/s on my hardware. The PGMS prototype currently does not take advantage of these optimizations, and as a result performs no better than Digital Unix when doing sequential accesses.

The following sections focus on the behavior of a single application, Render, to examine the performance characteristics of various aspects of PGMS in detail. I chose Render because of its data size and relatively complex data access pattern. Render is a difficult application for global memory systems because of its large number of cold misses and its poor data locality, which reduces the

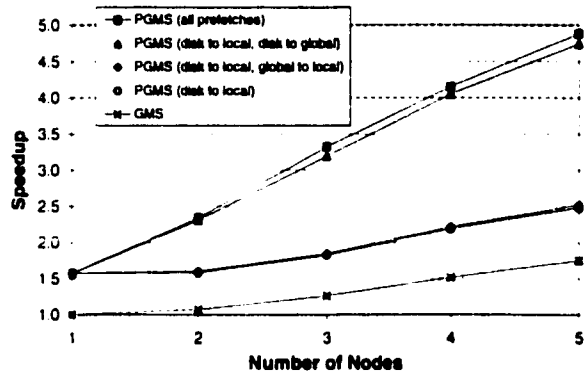


Figure 4.4: Speedup of Render application on PGMS

value of both local and global data caching.

4.4.4 Performance breakdown of prefetch types

Figure 4.4 shows the performance of the hinted Render application with various prefetching options. The top curve shows the speedup of Render under PGMS relative to its performance on a single node with neither prefetching nor global memory, as described in the previous section.

The bottom curve of Figure 4.4 shows the performance of Render under raw GMS, i.e., global memory is used only to hold pages evicted from the active node's memory, without PGMS prefetching. No prefetching of any type is performed. With 4 idle nodes, full PGMS outperforms GMS by a factor of 2.8 for this application.

The middle curves show the effect of selectively enabling both types of PGMS prefetching in order to quantify the performance contribution of each type. The second curve from the bottom shows that adding local-disk prefetching to the basic GMS system improves performance by approximately 57% on a single node. As idle nodes are added to cache evicted pages, the speedup increases linearly. This second curve uses no global-to-local prefetching: page fetches from global memory are demand faulted, as in the original GMS. The third curve from the bottom, which nearly overlaps the second, shows that enabling global-to-local prefetching provides only slight benefit. This is for two reasons. First, global memory holds only evicted pages, and the GMS curve (bottom curve) shows that the benefit of caching evicted pages for Render is modest. Second, global memory

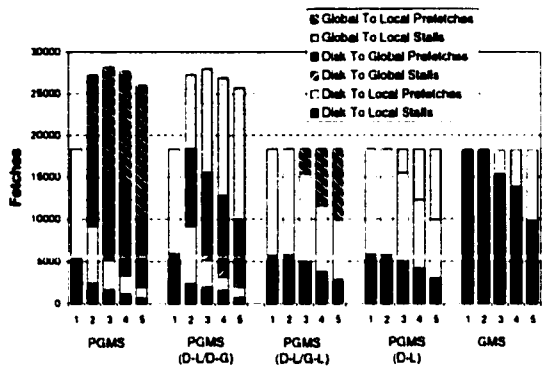


Figure 4.5: Prefetch request and stall breakdown for Render

faults using Trapeze/Myrinet are very fast relative to a disk access, reducing the I/O stall time to be saved by prefetching from global memory to local memory.

Finally, the top two curves show the large gains from adding disk-to-global prefetching in PGMS. The second curve from the top uses disk-to-global prefetching, but without global-to-local prefetching. This configuration suffers the cost of demand faults from global memory; however, global memory is used more effectively as a buffer for parallel prefetching of disk data ahead of its access. Thus, the benefit relative to the previous curve (with global-to-local prefetching but without disk-to-global prefetching) comes from replacing disk stalls with global memory stalls. The top curve shows the final performance improvement from eliminating most of these global stalls with global-to-local prefetching. Again, the incremental gains from global-to-local prefetching are much smaller than the gains from using global memory on a fast network to begin with.

4.4.5 Detailed breakdown of prefetch types

Figure 4.5 shows a detailed breakdown of the total number of fetches performed in the 1- through 5-node cases for each of the configurations shown in Figure 4.4, in order from fastest (on the left) to slowest (on the right). For each prefetch type I show counts of both the prefetch requests that arrived in time to avoid a stall (*prefetches*) and those that did not (*stalls*).

Note that the height of each bar shows the number of fetches, and not performance. In particular, the different bar components have different performance costs, and some blocks are fetched twice in

the faster PGMS configurations: once from disk into global memory, and once from global memory to local memory. The three rightmost sets of bars show the slower configurations with no disk-to-global prefetching, counting only fetches into the active node's memory. Since the size of the active node's memory is the same across all experiments, the total number of fetches remains constant. The two leftmost sets of bars show additional fetches resulting from PGMS disk-to-global prefetching.

The rightmost set of bars shows Render's performance on raw GMS. Render's working set is sufficiently large and its locality sufficiently poor that the addition of the first idle node does not yield any hits in the global cache; i.e., none of the pages evicted from local memory and cached in global memory are accessed again before being evicted from global memory. With the addition of the second idle node, the size of the global cache grows enough to eliminate some disk stalls.

The next two sets of bars from the right show the effects of adding disk-to-local and global-to-local prefetching to GMS, respectively. The second set of bars from the right show that the addition of local disk prefetching to GMS eliminates many of the local disk stalls not already eliminated by the global cache. This is in part because of the large number of cold misses in the Render experiment. The middle set of bars show that the addition of global-to-local prefetching eliminates all of the remaining stalls on global memory fetches, improving performance further.

The leftmost two sets of bars show the effect of adding disk-to-global prefetching, in order to benefit from the combined effects of global memory and I/O parallelism from the use of disks on the added nodes. As nodes are added, disk-to-local prefetches and stalls are reduced to their lowest levels, replaced by disk-to-global and global-to-local fetches. In the leftmost set, for full PGMS, the global-to-local fetches are prefetched, eliminating all but a few I/O stalls occurring at the start of the experiment. With 4 nodes, global memory is large enough to store Render's entire data set, and the number of disk-to-global prefetches drops significantly, leading to a reduction in the total number of fetches performed.

4.4.6 Elapsed time breakdown

Figure 4.6 shows a breakdown of the execution time for Render on the active node under full PGMS as a function of the number of nodes cooperating. The user CPU time component is constant since the program executes the same computation in all configurations; my prefetching techniques only

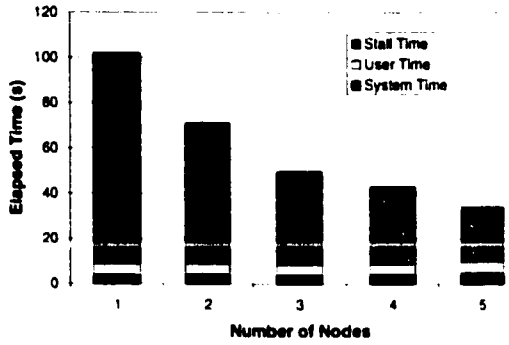


Figure 4.6: Execution time detail for Render

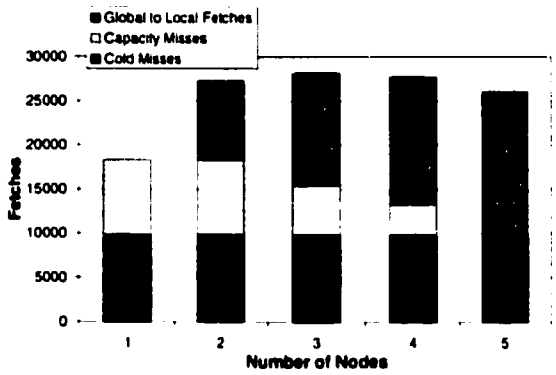


Figure 4.7: Breakdown of cold and capacity misses for Render

reduce I/O stall time. In the first bar, despite prefetching from the local disk, stall time is substantial on a single node. Adding additional nodes reduces stall time enormously, in exchange for a small increase in system time that includes the time spent in the PGMS prefetching code and the network driver.

Figure 4.7 gives more insight into Render's behavior by showing the causes of the PGMS prefetch requests. For a single node, disk fetches occur for two reasons: the cold misses are the unavoidable disk reads for the first access to each block in the dataset, while capacity misses are caused by insufficient memory to cache the entire dataset. Capacity misses are unchanged when a second node is added, since (as previously noted) the second node provides insufficient global memory to deliver global cache hits given Render's working set size and locality. However, with the

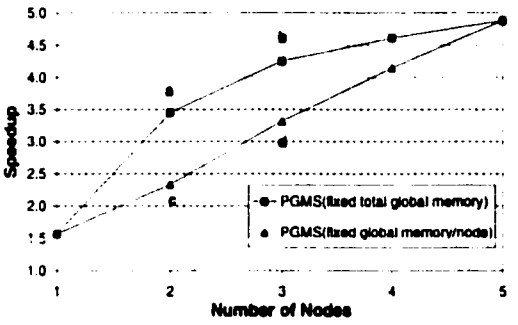


Figure 4.8: Fixed total global memory size vs. fixed per-node global memory size

second node these capacity misses now result in disk-to-global rather than disk-to-local prefetches, followed by global-to-local prefetches as shown in the top portion of the second bar. As additional nodes are added there is sufficient global memory to reduce capacity misses, and the count of global-to-local prefetches rises due to effective GMS caching of evicted pages in the idle cluster memory.

4.4.7 The impact of global memory size

A key question in global memory systems is the tradeoff between adding more global memory to an existing node vs. adding more nodes. To explore this tradeoff, I ran an additional set of experiments, shown in Figure 4.8. The lower curve shows the PGMS data previously presented; in these measurements, each idle node added an additional 32MB of global memory to the cluster. In the new experiment shown by the upper curve in Figure 4.8, I kept global memory size constant, independent of the number of idle nodes (i.e., total global memory size was always 128MB, divided evenly among the idle nodes). Comparison of points *a* and *c* shows that increasing memory on the single idle node (by 96MB) had a substantial performance impact, improving speedup by 48%. With 2 idle nodes (points *b* and *d*), increasing global memory size from 64MB to 96MB improved speedup by 28%. These improvements are not surprising, given what we've seen of Render's locality. Comparing points *a* and *b* can isolate the impact of an additional CPU and disk, independent of global memory size. In this case, performance increases by 23% due to the additional parallelism and bandwidth added by that CPU and disk resource. Thus, speedup in PGMS is caused both by

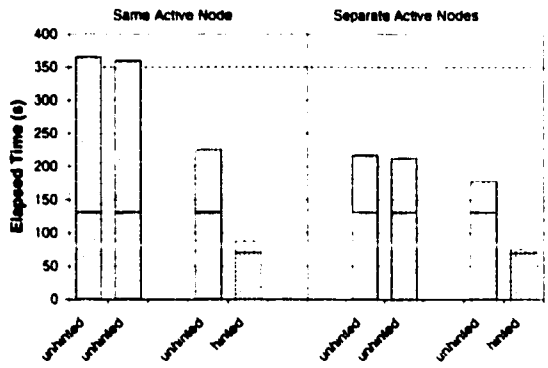


Figure 4.9: Elapsed times for two Render processes executing simultaneously.

the addition of global memory and by the additional parallelism provided by idle nodes. The exact benefit of either will of course depend on the nature of the applications.

4.4.8 Interaction of competing processes

It is interesting to examine the effect of prefetching when a prefetching process competes for resources. The left side of Figure 4.9 shows the effect of running two competing processes simultaneously on the same node. In one case I run two unhinted Renders; in the other case I run one hinted and one unhinted Render. There is one idle node in the system, and thus the two processes in each case are competing for both local and global resources. The line on the bars shows the performance for each application when running standalone in the same configuration. (While I showed two instances of the same program in Figure 4.9, experiments using different programs show similar results.)

Running the two unhinted applications together takes longer than running them sequentially, i.e., they suffer from the competition for memory and disk, causing each to run slower than 2 times its single-node time. Running the unhinted application along with a hinted application reduces its execution time substantially (compared to the previous example); in this case the hinted process is essentially unaffected by the unhinted process and completes quickly, leaving all of local and global memory for the unhinted process to use. The result is nearly optimal, as in shortest-job-first scheduling.

The right side of Figure 4.9 shows two similar experiments, however in this case each application runs on a different active node; they again share a single idle node, and thus they compete for global memory only. For the two unhinted applications running simultaneously, the elapsed time of each of the processes is within 1% of the elapsed time of an unhinted process running by itself in a system with one idle node with half the global memory, i.e., the sharing of the idle node is nearly perfect. For the hinted/unhinted experiment the results are similar to the single-node case; the hinted application finishes quickly, leaving the global resources to the unhinted application, which finishes relatively quickly afterwards.

4.4.9 Summary

In this section I presented measurements of Render in various system configurations. From these measurements, I showed the following results:

1. Global prefetching and caching can provide significant speedup over local-disk prefetching alone.

2. Disk-to-global prefetches can substantially improve performance by adding additional disk parallelism, and by turning disk misses into global memory fetches, which have much lower latency.

3. Compared to prefetches from disk, global-to-local prefetches are less significant because global fetch time is small (and shrinking with newer network technologies).

4. For some applications, increasing global memory size may have more impact than adding processors.

5. Running hinted and unhinted applications together does no harm to the unhinted application (when compared to running with another unhinted program). Hinted applications make more effective use of resources.

6. Applications experience both cold misses and capacity misses, depending on the memory requirements of the application and the configuration of the system. Caching in global memory

reduces the cost of capacity misses; prefetching reduces the cost of both cold and capacity misses. The Render application that I used has a large number of cold misses, and thus is aided more by prefetching than global memory caching.

Taken together, these results show the advantages of combining prefetching and global memory in a cluster.

4.5 Conclusions

This chapter presented PGMS, a system using cooperative prefetching and caching in a network-wide global memory system. Cooperative prefetching permits multiple network nodes with idle CPU cycles and memory pages to cooperate in prefetching on behalf of active nodes. This prefetching to global memory can reduce stall time without the risks of aggressive prefetching on the active nodes.

I have developed a hybrid algorithm for PGMS that combines aggressive prefetching into global memory with more conservative prefetching into local memory. I have designed and implemented an approximation to that algorithm on the Digital UNIX operating system running on a small cluster of Myrinet-connected DEC Alpha workstations. My results show that significant speedups can be achieved using cooperative prefetching for memory-bound applications. In addition, I quantify the impact of various types of prefetching: disk-to-local, disk-to-global, and global-to-local.

As network technology advances, the ratio between disk and network transfer times will continue to increase, and therefore using network resources to reduce the I/O bottleneck will be even more crucial in future generations. The PGMS experiments show how the integration of prefetching and global memory technologies permits multiple idle network resources to be used in parallel to benefit memory-bound jobs on active nodes in the cluster.

Chapter 5

Cooperative Caching in Wide-Area Networks

This chapter explores the fundamental question of scaling cooperative caching to wide-area distributed systems. In particular, it examines the potential of using cooperative caching among World Wide Web proxies deployed in the Internet. Internet proxy caching has become a commonplace approach for improving the performance of Web browsers. Typically, the proxy sits in front of an entire company or organization. By caching requests for a group of users, a proxy can quickly return documents previously accessed by other clients. Ultimately, though, the effectiveness of the proxy is a function of the size of the population it serves – a size often dictated by political, organizational, or geographic considerations. An obvious question, then, is whether multiple proxies should cooperate with each other in order to increase total client population, improve hit ratios, and reduce document-access latency.

Several cooperative caching techniques have been proposed for use in the Web. These proposals include hierarchical schemes like Harvest and Squid [Chankhunthod et al. 96, Squid et], hash-based schemes [Karger et al. 99, Valloppillil et al. 98], directory-based schemes [Fan et al. 98, Menaud et al. 98, Tewari et al. 99] and multicast-based schemes [Michel et al. 98, Touch 98]. Although each of these research efforts includes a performance evaluation of the protocols proposed and a discussion of algorithm scalability, only [Krishnan et al. 98] presents empirical evaluations of cooperation (and only for small populations), and none present empirical or analytical evaluations of the effectiveness of their schemes for the large client populations found in a wide-area setting.

As a result, several questions remain unanswered regarding the potential performance benefits of cooperative caching in large-scale Web environments. For what range of client populations can cooperative Web caching perform effectively in terms of hit rate, latency reduction, and bandwidth utilization? Previous studies have just looked at performance from the basis of a fixed number of requests. What is the effect on the performance of cooperative caching if the systems were running

for a month, a year, five years? In other words, how will cooperative caching schemes perform in steady-state operation over long-term time scales? Furthermore, many characteristics of the Web change constantly, such as client request rate, the rate of change of documents, and the size of the Web (in terms of clients, servers, and documents). How do the results of cooperative caching apply in the face of future trends of Web characteristics? Lastly, various cooperative caching architectures have been proposed, but not compared at large-scales. What arrangement of cooperating caches provides the best performance, and/or the most efficient use of storage resources?

In this chapter, I develop an analytic model of Web access behavior to explore these questions. The model determines the steady-state performance of Web caches as a function of the size of client populations and other workload parameters, such as document rate of change. With this model, I evaluate the effectiveness of cooperative Web proxy caching at population sizes that range from a single organization to 100s of millions of clients, explore the tradeoffs among various cooperative caching schemes, and speculate on the performance implications of future trends in Web workloads.

Overall, I demonstrate that the utility of cooperative Web proxy caching fundamentally depends upon the scale at which it is applied. It is most effective among groups of small organizations, incrementally beneficial among large organizations within a medium-sized metropolitan area, and only marginally useful for larger client populations.

5.1 Introduction

Cooperative caching has been shown to improve the performance of file and virtual memory systems in a high-speed, local-area network environment. Unfortunately, the algorithms, mechanisms, and policies used in cooperative caching systems for workstation clusters do not scale beyond hundreds of nodes due to assumptions inherent in local-area network performance. For example, GMS uses periodic broadcasts to synchronize the global state of all nodes in the cluster [Feeley et al. 95], an operation not practical in wide-area systems. As a result, it is not known to what degree cooperative caching techniques might apply to large-scale distributed systems comprising millions of nodes deployed across wide-area networks. This chapter explores the fundamental question of scaling cooperative caching to wide-area distributed systems. In particular, it examines the potential of using cooperative caching among World Wide Web proxies deployed in the Internet.

Internet proxy caching has become a commonplace approach for improving the performance of Web browsers. Typically, the proxy sits in front of an entire company or organization. By caching requests for a group of users, a proxy can quickly return documents previously accessed by other clients. Ultimately, though, the effectiveness of the proxy is a function of the size of the population it serves – a size often dictated by political, organizational, or geographic considerations. An obvious question, then, is whether multiple proxies should cooperate with each other in order to increase total client population, improve hit ratios, and reduce document-access latency.

Whether cooperative proxy caching is useful for improving performance depends on a number of issues. These include the client access patterns of Web documents and the cacheability properties of those documents, the performance of proxy caches in light of client access patterns and document characteristics, and the specific organization of proxy caches used to mitigate the long latencies and limited bandwidth of wide-area networks.

Web tracing and caching are highly active research areas. Recent studies of Web traffic include analyses of Web access traces from the perspective of browsers [Cunha et al. 95, Mah 97], proxies [Almeida et al. 96b, Breslau et al. 99, Caceres et al. 98, Cao 98, Crovella et al. 96, Douglis et al. 97, Duska et al. 97, Feldmann et al. 99, Gribble et al. 97, Kroeger et al. 97, Rabinovich et al. 98], and servers [Almeida et al. 96a, Arlitt et al. 96, Mogul 95]. Earlier tracing studies were limited in request rate, number of requests, and diversity of population; more recent tracing studies have been larger in scale and more diverse in characteristics. In addition to static analysis, some studies have also used trace-driven cache simulation to characterize the locality and sharing properties of large traces [Almeida et al. 96b, Caceres et al. 98, Duska et al. 97, Feldmann et al. 99, Gribble et al. 97, Kroeger et al. 97], and to study the effects of cookies, aborted connections, and persistent connections on the performance of proxy caching [Caceres et al. 98, Feldmann et al. 99].

Researchers have studied the temporal locality of Web proxy traces and examined how hit-ratio depends, asymptotically, on cache size and the number of requests. Several interesting findings have been identified. First, it has been repeatedly found that, for most traces, the relative frequency with which Web pages are requested follows a Zipf-like distribution, where the number of requests to the i^{th} most popular document is proportional to $1/i^{\alpha}$ for some constant α [Almeida et al. 96b, Breslau et al. 99, Cao et al. 97, Cunha et al. 95, Glassman 94, Kroeger et al. 96]. Second, for infinite-sized caches, it has been shown empirically and analytically that the hit ratio for a Web proxy grows

logarithmically with the client population of the proxy and the number of requests seen by the proxy [Breslau et al. 99, Cao et al. 97, Duska et al. 97, Gribble et al. 97, Kroeger et al. 96].

Finally, several cooperative caching techniques have been proposed for use in the Web. These proposals include hierarchical schemes like Harvest and Squid [Chankhunthod et al. 96, Squid et], hash-based schemes [Karger et al. 99, Valloppillil et al. 98], directory-based schemes [Fan et al. 98, Menaud et al. 98, Tewari et al. 99] and multicast-based schemes [Michel et al. 98, Touch 98]. Although each of these research efforts includes a performance evaluation of the protocols proposed and a discussion of algorithm scalability, only [Krishnan et al. 98] presents empirical evaluations of cooperation (and only for small populations), and none present empirical or analytical evaluations of the effectiveness of their schemes for the large client populations found in a wide-area setting.

As a result, several questions remain unanswered regarding the potential performance benefits of cooperative caching in large-scale Web environments. For what range of client populations can cooperative Web caching perform effectively in terms of hit rate, latency reduction, and bandwidth utilization? Previous studies have just looked at performance from the basis of a fixed number of requests. What is the effect on the performance of cooperative caching if the systems were running for a month, a year, five years? In other words, how will cooperative caching schemes perform in steady-state operation over long-term time scales? Furthermore, many characteristics of the Web change constantly, such as client request rate, the rate of change of documents, and the size of the Web (in terms of clients, servers, and documents). How do the results of cooperative caching apply in the face of future trends of Web characteristics? Lastly, various cooperative caching architectures have been proposed, but not compared at large-scales. What arrangement of cooperating caches provides the best performance, and/or the most efficient use of storage resources?

In this chapter, I develop an analytic model of Web access behavior to explore these questions. The model determines the performance of Web caches as a function of the size of client populations and other workload parameters. It is inspired by the model of [Breslau et al. 99], but differs from it in two key ways: (1) I consider the *steady-state* behavior of caching systems rather than caching behavior based on a finite request sequence, and (2) I incorporate document rate of change into the model rather than assuming that documents are static.

With this model, I evaluate the effectiveness of cooperative Web proxy caching at population sizes that range from a single organization to 100s of millions of clients, explore the tradeoffs among

various cooperative caching schemes, and speculate on the performance implications of future trends in Web workloads. In particular, results from the model show that:

- Cooperative Web proxy caching is effective at organizational scales, but it is not necessary; a monolithic proxy cache provides performance at least as good, but with less complexity.

- Cooperation among organization-sized caches within a medium to large city will provide additional performance benefit, although only marginal, over cooperative caching at small scales. Assuming that bandwidth within a city is plentiful and latencies are small, the overhead of cooperation would be low and therefore worth the secondary benefits that such caching provides.

- Extrapolating to larger scales, cooperative caching provides limited or no additional benefit, particularly in the face of increased latencies among caches.

- Performance at the population level at which cooperative caching works most effectively is fundamentally limited by document *cacheability*. Thus, the crucial problem that must be solved to improve Web performance is how to increase document cacheability.

- The three common architectures for Web proxy cooperation, hierarchical, directory-based, and hash-based, all perform well in the region where cooperative caching remains advantageous (e.g., at the level of a medium-sized city). In principle, the organizational caches within a city can use a hash-based scheme to maximize storage efficiency. In practice, however, given the cheap cost of disks, using a hash-based scheme to spread load is more important than storage efficiency.

- The effectiveness of large-scale proxy caches is very sensitive to the change rates of popular and unpopular documents. For popular documents, hit rate varies moderately when documents change faster than once a day. In contrast, hit rate is sensitive to the rate of change of unpopular documents on an entirely different time scale. For unpopular documents, hit rate varies considerably when documents change *slower* than once a day.

- The populations at which large-scale caching systems experience diminishing returns will decrease over time. Trends indicate that client request rate is increasing significantly over time, faster than increasing document rates of change. Since the population needed to achieve a given hit rate varies inversely with the client request rate, faster client request rates will decrease the population needed to approach maximum hit rates.

- Growth in the number of documents in the Web increases the populations at which large-scale proxy cache systems remain effective. Web growth and population scale increase proportionally; an order of magnitude growth in the size of Web makes caching systems effective at scales an order of magnitude larger.

The rest of this chapter is organized as follows. Section 5.2 develops an analytic model of steady-state Web proxy caching and uses the model to study the performance of large-scale proxy caches. Section 5.3 then uses the model to compare cooperative caching schemes, and Section 5.4 summarizes and concludes.

5.2 An analytic model of Web accesses

In this section, I develop an analytic model of Web accesses. I then use the model to explore the steady-state performance of cooperative caching for large client populations, and to speculate on caching performance in light of future trends. I then examine the tradeoffs between various cooperative-caching schemes in Section 5.3.

5.2.1 Steady-state performance

Previous trace-based and analytic studies of proxy cache performance bound the performance of caching schemes in the short term – over a period of one week. How would these results change if the cooperative caching systems were up and running for a month, a year, five years? In other words, how will cooperative caching schemes perform over the long run? In this section I examine the *steady-state* or *limiting* performance of caching schemes.

The model of steady-state performance assumes that a cache can store all cacheable documents

in the Web and that there are no capacity misses in the workload[1]. With this assumption, it is conceivable that in the long term the hit rate for cacheable documents would approach 100% since the relative importance of cold-cache effects, such as compulsory misses, would diminish. For example, this would happen if the Web were static, i.e., documents were not changing and new documents were not being generated. In this case, *all* documents would eventually be requested, and no further misses would be incurred thereafter. On the other hand, if new documents are constantly being created and old documents are changing, the hit rate for cacheable documents might remain low even over the long run.

The ultimate performance of a cache will therefore depend upon the rate at which documents change compared to the rate at which documents are requested. If the request rate dominates document rate of change, then the cache will still achieve near optimal hit rates. One request will miss in the cache whenever a document changes, but all subsequent requests to that document will be hits. However, if the request rate does *not* dominate document rate of change, then the cache will perform poorly. Repeated requests to a document will often find that the document has changed, so most of those requests will be misses. Since increasing the population served by a cache also increases the request rate, cooperative caching increases the likelihood that document request rate dominates document rate of change.

Similarly, the creation of new documents in the Web introduces cold misses into the workload when those documents are requested. As with document rate of change, if the Web grows slowly compared to the request rate, then caches will perform well: only a small fraction of requests will result in cold misses. However, if the Web grows significantly faster than the request rate, then the cache will be dominated by cold misses and will perform poorly.

I use the steady-state model to explore these effects in detail. I begin by introducing the model in Section 5.2.2, and then describe its parameterization in Section 5.2.3. Section 5.2.4 explores the steady-state performance of cooperative caching for large client populations and speculates on caching performance in light of future trends.

---

[1] I contend that the storage needed to do this is not intractable, and that capacity misses are therefore not an important aspect of the workload. Storage for the cacheable fraction of all documents currently estimated to be on the Web would require a large contemporary disk array, such as the 9TB EMC Symmetrix 5930 storage system [EMC 99].

5.2.2 *The model*

The goal of the model is to determine the performance of Web caches as a function of the size of client populations and other workload parameters. It is inspired by the model of [Breslau et al. 99], but differs from it in two key ways: (1) I consider the *steady-state* behavior of caching systems rather than caching behavior based on a finite request sequence, and (2) I incorporate document rate of change into the model rather than assuming that documents are static. My goal in building the model also differs from the goals pursued by Breslau et al. They used their model to study proxy cache replacement algorithms; I use my model to understand the performance of large-scale, cooperative-caching schemes in terms of hit rate, latency, bandwidth savings, and storage consumed.

To begin, I make the following assumptions about clients and documents in the model:

- There are N clients in the population. Clients are indistinguishable and act independently of one another.

- The total number of documents is n. For simplicity, I model documents as indivisible, rather than as compound, and assume that accesses to objects are independent.

- The fraction of all requests that are for the i-th most popular document, or the "popularity" of this document, is denoted by p_i. I assume that documents follow a Zipf-like distribution [Breslau et al. 99], i.e., that p_i is proportional to $1/i^{\alpha}$ for some constant α. The important characteristic of a Zipf-like distribution is that it is *heavy-tailed* – a significant fraction of the probability mass is concentrated in the tail, which in this case means that a significant fraction of requests go to the relatively unpopular documents. As α increases, the distribution becomes less heavy-tailed, and a larger fraction of the probability mass is concentrated on the most popular documents.

- The distribution of time between requests made by the client population is exponential with parameter λN, where λ is the average client request rate.

- The distribution of time between changes to a document is exponential with parameter μ, independent of document size and latency, but not independent of popularity. I use two sep-

arate document change distributions, one for popular documents with mean μ_p and another for unpopular documents with mean μ_u. The number of popular documents is n_p. Document change can be used to model either expiration or actual change.

- The probability that a requested document is cacheable is p_c.

- The average document size is $E(S)$. Document size is independent of document popularity, latency, and rate of change.

- The last-byte latency to the server that houses that document has average value $E(L)$. Last-byte latency is independent of document popularity and document rate of change.

I justify the independence assumptions I make by the fact that all the correlation coefficients (between each pair of document size, document popularity, first-byte latency, and document cacheability) computed from a large client trace of the University of Washington (see Section 5.2.3 below) are close to 0; these correlations also agree with those computed by [Breslau et al. 99] for six other client traces. Analyses of the UW trace show that cacheable objects are more likely to have low latencies, making the model conservative with respect to reality. The data from the UW trace strongly also suggests that the rate-of-change distribution is heavy-tailed. However, to make the model tractable to solve analytically, I assume that the rates of change for popular and unpopular documents are distributed exponentially. Since the steady-state performance of a proxy cache improves as document inter-modification times increase, this approximation underestimates cache performance and again makes it conservative.

With these assumptions, I can compute a number of performance characteristics. In steady state, it is easily shown that for a single proxy cache serving a population of size N:

- The steady state hit rate is

$$H_N = p_c C_N,$$

where C_N is the probability that a request is a hit given that it is cacheable. The steady-state cacheable hit rate to cacheable documents C_N is

$$C_N = \sum_{1 \le i \le n} p_i \frac{\lambda N p_i}{\lambda N p_i + \mu}.$$

Taking p_i proportional to $1/i^{\alpha}$, a very close approximation to this sum is given by

$$\int_1^n \frac{1}{Cx^{\alpha}} \left(\frac{1}{1 + \frac{\mu x^{\alpha} C}{\lambda N}} \right) dx,$$

where

$$C = \int_{1 \le x \le n} \frac{1}{x^{\alpha}} dx.$$

The first integral can be evaluated exactly for $\alpha = 1$ and numerically for other values of α.

- The expected last-byte latency to serve a request is given by

$$lat_{req} = (1 - H_N)E(L) + H_N * lat_{hit},$$

where lat_{hit} is the latency for a cache hit.

- The average bandwidth savings per request due to proxy caching, measured in kilobytes *not* transferred, is denoted B_N and is given by

$$B_N = H_N E(S).$$

- The expected amount of storage required in a proxy cache for a population of this size is $nH_N E(S)$. This is actually optimistic, as it assumes that only objects that are cacheable and have not expired are cached.

The goal of this section is *not* to exactly model empirical Web caching results. Rather, it is to examine at a high level the impact of changes to, or the sensitivity of, various workload parameters in light of future trends.

5.2.3 Model parameters

I parameterize the model using values computed from a large, week-long client trace of the University of Washington (UW) [Wolman et al. 99a]. These values are summarized in Table 5.1.

I estimate the number of objects in the Web, n, using results from a study by [Lawrence et al. 99]. Based upon February 1999 data, they estimate that the Web has 800 million compound Web documents. I use an estimate of 3.2 billion objects in the Web, since each compound document in the UW trace contained an average of four objects.

Table 5.1: Default model parameters from the UW trace.

Parameter	Value	Parameter	Value
α	0.8	p_c	0.6
n	3.2 billion	$E(S)$	7.7 KB
n_p	10.4 million	$E(L)$	1.9 seconds
λ	590 reqs/day	lat_{hit}	10 ms
Slow		*Fast*	
μ_p	1/14 days	μ_p	1/5 minutes
μ_u	1/186 days	μ_u	1/85 days
Mid Slow		*Mid Fast*	
μ_p	1/1 day	μ_p	1/1 hour
μ_u	1/186 days	μ_u	1/85 days

I assume that the time to serve documents from a proxy cache lat_{hit} is 10ms. Although this may be an optimistic value, particularly when caches are under heavy load and requests experience queueing delays, increasing lat_{hit} merely offsets the latency results by a similar amount until it reaches a second or more.

Table 5.2 summarizes the rate-of-change values observed in the UW trace for three policies. I found that document rate of change is correlated with document popularity, so the table contains results separated into popular and unpopular documents. This finding is consistent with previous rate of change studies. For example, [Douglis et al. 97] also found that the popular pages changed more often than the less popular pages. I define popular documents as the most frequently requested documents that account for 40% of all requests. Due to the Zipf popularity distribution, the popular documents comprise only 0.3% of all documents. I account for this in the model by using two separate rate-of-change distributions, one for popular documents (mean μ_p) and another for unpopular documents (mean μ_u). In the model results, I use four parameterizations of these distributions. The "slow" parameterization uses the *mean* rates of change for popular and unpopular documents computed from the UW trace as the means μ_p and μ_u of the rate-of-change distributions in the model.

Table 5.2: Document rate of change (in days between changes) for three different policies (Normal, Always Change, and Never Change), and then broken down by Cacheable and Uncacheable documents using the Always Change policy.

Scenario	Popular		Unpopular	
	Mean	Median	Mean	Median
Normal	14	< 1	186	85
Always Change	3	< 1	129	23
Never Change	27	< 1	763	180
Cacheable	5	< 1	168	65
Uncacheable	< 1	< 1	22	< 1

The "fast" parameterization uses the *median* rates of change as computed from the UW trace as the means for the rate-of-change distributions in the model. And "mid slow" and "mid fast" represent intermediary values for the rate-of-change parameters.

Since the UW trace does not record the data transferred during a connection, I must rely on the "Last-Modified" HTTP header to detect document changes. However, this header is not always present in Web server responses. Other studies on rate of change, where the full document content was available, have determined that relying solely on HTTP header information has some pitfalls, the most common being when the headers signal a change when none has occurred [Douglis et al. 97, Wills et al. 99]. Wills et al. [Wills et al. 99] also quantify how often documents change when the "Last-Modified" field is missing. The top three lines of Table 5.2 show results that differ only in how I treat requests to documents with incomplete header information (36% of the requests). For the "normal" results, I simply ignore those documents. For the "always change" results, I calculate an upper bound by assuming that those documents change between each access. For the "never change" results, I calculate a lower bound by assuming that no document missing this header field changes.

I also investigated whether the rate of change for cacheable documents was different than that of uncacheable documents. The "cacheable" and "uncacheable" lines in Table 5.2, generated using the "always change" policy, show that uncacheable documents change much more rapidly than

cacheable documents. It is important to note that since the model is restricted to cacheable documents, the rate of change results for cacheable documents are more relevant as input parameters to the model.

5.2.4 *Performance of large scale proxy caching*

I begin by examining basic performance results from the model using parameters extracted from the trace data. I then consider the effects of possible future changes in the fundamental parameters of the Web. In particular, I examine the impact on performance of (a) the rate of change of Web documents μ, (b) the client request rate λ and population size N, (c) the Zipf parameter α of the popularity distribution, and (d) the rate of growth of the Web, measured in the number of accessible documents n.

Hit rate, latency, and bandwidth

Figure 5.1 shows the steady-state hit rate for cacheable documents C_N as a function of the population size N, graphed on a log scale. The figure shows four curves corresponding to four possible values for the rate of change parameters presented in Table 5.1. A key question to address is: given the client request rate, document rate of change, and document popularity distributions, how large a client population is needed to achieve a cache hit rate approaching p_c, the fraction of cacheable Web documents.

All these curves can be viewed as consisting of three regions: (1) an initial region in which hit rates grow slowly, (2) a large middle region in which hit rates grow linearly (as population grows exponentially), and (3) a final region in which hit rates grow slowly again, ultimately converging to 100%. In the initial region, the request rate is too low to dominate the rate of change for unpopular documents. As a result, the hit rate for unpopular documents remains close to zero and almost all of the hit rate improvement is accounted for by hits on popular documents. The transition to the middle region marks the beginning of a significant increase in the hit rate to unpopular documents. The heavy-tailed nature of the popularity distribution implies that an exponential increase in request rate is needed to obtain a linear increase in the fraction of requests to unpopular documents that are hits, i.e., requests to documents that have been requested more recently than they have changed. The

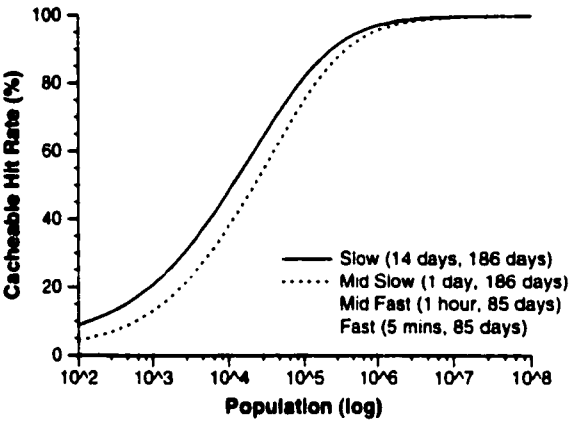


Figure 5.1: Cacheable request hit rate as a function of client population (log scale).

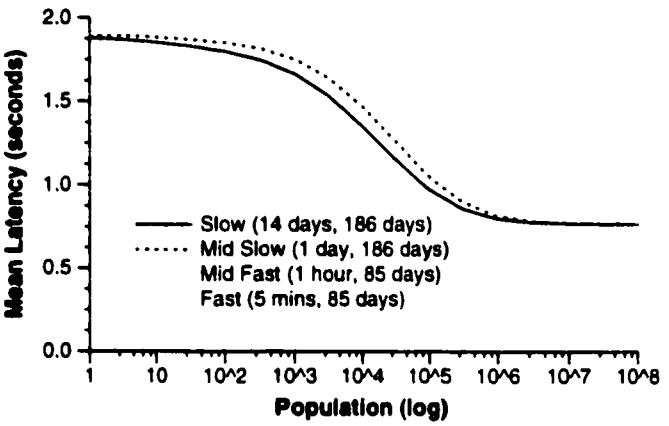


Figure 5.2: Mean request latency as a function of client population.

transition to the final region occurs when the hit rate for unpopular documents approaches 100%. Behavior in this region is once again accounted for by improvements in the hit rate for popular documents. Here, the request rates are high enough to dominate those inter-document modification times that are much smaller than the mean of the exponential distribution.

Figure 5.1 shows that proxy cache hit rate is very sensitive to document rate-of-change parameters. The client population required to achieve 90% of the cacheable hit rate p_c is only 250,000 for the slowest parameters but nearly 20 million for the fastest parameters.

Finally, I would like to understand how hit rate translates into actual performance improvement. Hit rate fundamentally determines the improvement in object access latency and the reduction in

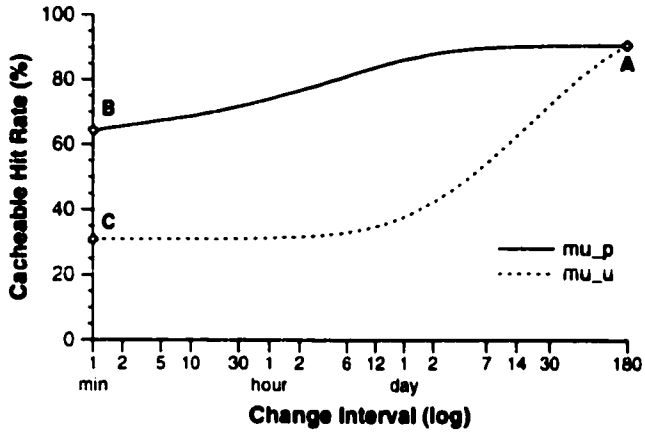


Figure 5.3: Sensitivity of hit rate to the rate of change of popular and unpopular documents, μ_p and μ_u. The hit rates were calculated for a population of 250,000.

network bandwidth consumed by transmitting Web objects. The effect on object access latency is shown in Figure 5.2. This figure graphs mean latency as a function of N, assuming a single cooperative cache for the entire population that serves cache hits with an average latency of 10ms. The curves asymptote at $(1 - p_c)E(L)$, the mean latency of uncacheable documents; recall that p_c=0.6 and $E(L)$=1.9 seconds, so the curves asymptote at 0.76 seconds. For the slower rates of change, most of the benefit is achieved at medium-sized populations: 95% of the maximum benefit is achieved at a population of 500,000. For the fast rates of change, 68% of the maximum benefit is achieved at a population of 500,000, and it decreases thereafter. Since the impact on latency and bandwidth is directly a function of hit rate, the curves for the effect on bandwidth are identical to those for latency.

Document rate of change

Two key issues regarding document rate of change concern: (1) how sensitive the hit rate is to document change rate, and (2) which parameter has the greatest impact on hit rate – the rate of change of popular documents μ_p or unpopular documents μ_u. Figure 5.3 shows the sensitivity of the proxy cache hit rate to the rate of change of popular and unpopular documents for a population of 500,000 clients. The x-axis is the mean interval between changes to a document (the inverse of μ) on a log scale, and the y-axis is the hit rate of cacheable documents. The top curve shows the

effect on hit rate of varying the mean rate of change of popular documents μ_p from a very slow rate, viz., 1 change every 180 days (point A), to a very fast rate, viz., 1 change every minute (point B). For this curve, the rate of change of unpopular documents μ_u is held constant at the slow rate of 1 change every 180 days to minimize its impact. Similarly, the bottom curve shows the effect on hit rate of varying the mean rate of change of unpopular documents μ_u between the same extremes of slow (point A) and fast (point C) rates of change. For this curve, the rate of change of popular documents μ_p is held constant at the slow change rate.

Figure 5.3 shows that the proxy cache hit rate is very sensitive to the change rates of popular and unpopular documents. For popular documents, hit rate varies moderately when documents change faster than once a day. When popular documents change slower than once a day, there is little impact on hit rate. In contrast, hit rate is sensitive to the rate of change of unpopular documents on an entirely different time scale. For unpopular documents, hit rate varies considerably when documents change *slower* than once a day. Once unpopular documents change at least once a day, they are already changing faster than they are being requested. At this point, hit rate reaches a minimum and does not decrease, even at higher rates of change. Thus, for popular documents the issue is whether they change on the scale of minutes to hours; for unpopular documents it is whether they change on the scale of days to weeks to months.

Figure 5.3 also shows that the rate of change of unpopular documents μ_u has a more significant impact on proxy cache hit rate than the rate of change of popular documents μ_p. This behavior is a result of the dynamics between document popularity and rates of change. Requests are heavily skewed to popular documents; therefore, even with high change rates, the request rate to popular documents dominates. While each document change makes the next request to that document miss in the cache, the popularity of the document is such that it will have many requests to the document before it changes again, and these requests hit in the cache. However, the request rate to unpopular documents is so low that the change rate dominates. Even if unpopular documents change at a moderate rate, requests to those documents always find out-of-date versions in the cache.

Figure 5.4 shows results similar to Figure 5.3, except for a much larger population of 20 million users. This population corresponds to a hit rate of 90% for the fast rates of change. Comparing the figures, the time scales at which hit rate is sensitive to rates of change decrease. For popular documents the time scale is now an hour or less, and for unpopular documents most of the variation

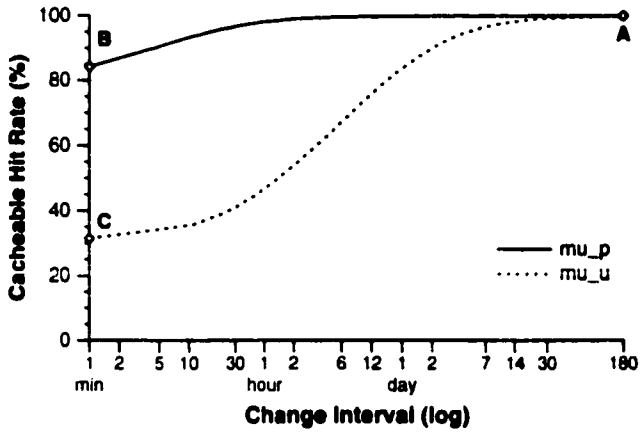


Figure 5.4: Sensitivity of hit rate to the rate of change of popular and unpopular documents, μ_p and μ_u. The hit rates were calculated for a population of 20 million.

in hit rate is between rates of change of 10 minutes to a month. At these very large population sizes, document request rates are significantly higher than with the smaller population in Figure 5.3. For both popular and unpopular documents, these high request rates dominate even higher rates of change.

Client request rate

The population needed to achieve a given hit rate varies inversely with the client request rate λ. When λ is very low, even large populations cannot dominate the object rate of change and therefore cannot keep the cache filled with up-to-date objects. On the other hand, when λ is extremely large, even small populations can maintain a filled, up-to-date cache. Trends indicate that λ is increasing significantly over time; the per client request rate in the UW trace is eight times that of the 1996 DEC trace [Kroeger et al. 96]. Unfortunately, there is minimal current or historical data on document rate of change. The rates of change in the UW trace seem well within a factor of two of those presented in [Douglis et al. 97]. Based upon just these two points of reference, it does appear that λ is growing much faster than μ_u. As a result, the populations at which large-scale caching systems experience diminishing returns will decrease over time.

Document popularity and size of the Web

I also examined the sensitivity of the model to variations in the Zipf parameter α and the number of documents in the Web n. Increasing α skews the distribution even further towards popular documents. This greater skew towards popular documents significantly increases hit rates for slower rates of change, but only slightly increases hit rates for faster rates of change. Increasing the number of documents n simply shifts the curves for slow and fast rates of change to larger populations; it does not significantly change the shapes of the curves. This population shift is roughly in proportion to the increase in n: for example, for n=3.2 billion, the slow curve reaches a 90% hit rate at a population of 250,000; for n=32 billion, the slow curve reaches a 90% hit rate at a population of 25 million; and for n=320 billion, the slow curve reaches a 90% hit rate at a population of 250 million.

5.2.5 Summary

In this section, I developed an analytic model and used it to examine the steady-state performance of cooperative-caching schemes at various population scales. I found that cooperative Web proxy caching is effective at organizational scales, but it is not necessary; a monolithic proxy cache provides performance at least as good, but with less complexity. Cooperation among organization-sized caches within a medium to large city will provide additional performance benefit, although only marginal, over cooperative caching at small scales. Assuming that bandwidth within a city is plentiful and latencies are small, the overhead of cooperation would be low and therefore worth the secondary benefits that such caching provides. Extrapolating to larger scales, cooperative caching provides limited or no additional benefit, particularly in the face of increased latencies among caches.

I also used the model to speculate on the performance implications of future trends in Web workloads. I found that the effectiveness of large-scale proxy caches is very sensitive to the change rates of popular and unpopular documents. For popular documents, hit rate varies moderately when documents change faster than once a day. In contrast, hit rate is sensitive to the rate of change of unpopular documents on an entirely different time scale. For unpopular documents, hit rate varies considerably when documents change *slower* than once a day.

I also found that the populations at which large-scale caching systems experience diminishing returns will decrease over time, and growth in the number of documents in the Web increases the

populations at which large-scale proxy cache systems remain effective.

5.3 Comparing cooperative caching schemes

The previous section explored the performance of large Web proxy-caching systems. This section examines the question of how these systems are designed and organized. Proxy caching starts at the level of individual organizations, small and large. To achieve the performance of caching systems with large populations, some form of cooperative caching among these organizational proxies will have to be used. In this section, I extend the calculations from the model to understand the differences in performance between various cooperative-caching systems as a function of scale. I compare three basic schemes: a hierarchical caching system inspired by Squid [Squid et]; a flat hash-based caching system inspired by [Karger et al. 99, Valloppillil et al. 98]; and a directory-based scheme inspired by Summary Cache [Fan et al. 98]. I evaluate these schemes at the scale of a City, a State, and a large region (the West Coast of the U.S.). The results of these evaluations will show why there is little motivation to scale to a region larger than a medium-sized city.

The *hierarchical caching system* assumes a hierarchy of k levels of caches, with a fanout of d at each level, where the bottom-level caches serve as proxy caches for populations of size N_k. A client's request is forwarded up the hierarchy until a cache hit occurs; if none occurs, the request is forwarded to the server. A copy of the requested object is then stored in all caches along the request path. In what follows, I will assume that the top-level cache serves an overall population of size $N_1 = N$, each second-level cache serves a disjoint subpopulation of size N_2, and on down to a set of k-th-level caches, each serving subpopulations of size N_k. (Concisely, $N_i = N/d^{i-1}$.) For simplicity, I assume fixed latencies between the caches in the hierarchy, where L_i is the latency between level $i+1$ and level i caches.

The *hash-based caching system* assumes a total of m caches cumulatively serving a population of size N. (I will assume $m = d^k$ from the hierarchical scheme.) I assume a fixed average latency of L_c to transmit data from a random cache to a client in the population, and a hash function that randomly maps URLs to one of the m caches uniformly. Upon a request by a client, the client hashes the URL and forwards the request to the corresponding cache C. If the cache stores the document, it is forwarded to the client. Otherwise, the request is forwarded to the server, and a copy is returned

to the client and to the cache C. The advantages of such a scheme are: (1) that load is balanced across the proxy caches, and (2) only one copy of each document is stored in the entire cooperative caching system. Such schemes have been proposed primarily for use in a local-area setting, since for large populations L_c may be sizeable.

Finally, in the *directory-based system*, I assume a total of m caches cumulatively serving a population of size N. In this system, the population is partitioned into subpopulations of size N/m, and each subpopulation is served by a single proxy cache. Each proxy cache maintains a directory that summarizes the set of documents stored at each of the other proxy caches in the system. When a client issues a request for a document, it is forwarded to its proxy. If the proxy has the object, it returns it directly to the client. Otherwise, the proxy cache checks its directory to see if another proxy cache in the system stores a copy of the document. If so, the request is sent to a random cache storing a copy, which then returns a copy of the document to the proxy cache and to the requesting client. I assume a latency of L_l to transmit data from a proxy cache to a client it serves, and a latency of L_c between proxy caches in the system.

To maintain the directories, each proxy cache periodically (every t time units) sends out an update about the contents of its cache. In particular, it multicasts the set of changes to its document set since the last request. I ignore the overhead of these messages in my analyses, as well as the extra misses caused by directory entries that become stale between updates.

These descriptions of cooperative caching systems are stated in general terms that emphasize the structure and operation of the systems, glossing over potential implementation details. For example, the description of the hierarchical system is in terms of multiple levels of caches. In practice, these caches may be special caches maintained at ISPs, or existing organizational caches that serve the role of first, second, and third level caches for different portions of the Web name space, as with [Tewari et al. 99]. The issue of whether it is better to use separate dedicated caches or to overload existing individual caches with multiple responsibilities is a detailed design issue that is beyond the scope of this thesis.

Table 5.3 summarizes the performance of the three cooperative caching schemes, based on the model. Recall from Section 5.2 that $E(L)$ is the average latency to a server and $E(S)$ is the average document size. I compare the performance of these schemes at three different scales:

Table 5.3: Cooperative caching performance parameters.

	Hierarchical	**Hash-based**	**Directory-based**
Request Arrival Rate	$\lambda N_i(1 - H_{N_{i+1}})$ to level i ($H_{N_{k+1}} = 0$)	$\lambda N/m$	$\lambda N/m$ $+\frac{\lambda N}{m}(1 - H_{\frac{N}{m}})H_{N(1-\frac{1}{m})}$ (2nd term – requests from other proxies)
Average Request Latency	$(1 - H_{N_1})E(L)$ $+\sum_{1 \le i \le k} L_i(H_{N_i} - H_{N_{i+1}})$	$(1 - H_N)E(L) + H_N L_c$.	$(1 - H_N)E(L)$ $+(1 - H_{\frac{N}{m}})H_{N(1-\frac{1}{m})}2L_c$ $+H_{\frac{N}{m}}L_l$
Storage Per Proxy	$\sum_{1 \le i \le k} nd^{i-1}H_{N_i}E(S)/m$	$nH_N E(S)/m$	$nH_{\frac{N}{m}}E(S)$ (lower bound)

1. A medium-sized City (N=0.5 million users)

2. A small State (N=5 million users)

3. The west coast of the U.S. (N=50 million users)

From the UW trace, the average client issues just under 600 requests. Based on this, I assume that there are 50,000 clients behind each lowest-level proxy cache, which results in an average request rate of about 350 requests per second. This request rate is well within the load a single host can handle as a proxy cache (e.g., [Danzig 98] reports 500 requests per second, and various single host proxies from the Web Caching Bake-off report throughputs ranging from 96–690 requests per second [Rousskov et al. 99]). Based on the populations at different scales, this results in a total of $m = 10$ organizational proxy caches for the City, $m = 100$ caches for the State and $m = 1000$ caches for the west coast. For the hierarchical scheme, I assume that $d = 10$, giving us a two-level hierarchy for the city (a single top-level proxy cache on top of the 10 organizational caches), a three-level hierarchy for the state, and a four-level hierarchy for the west coast.

I used the values in Table 5.1 from the UW trace to parameterize the model. For rates of change, I used the "mid slow" values, where μ_p is one change per day and μ_u is one change per 186 days. These rates of change most closely matched the performance observed in the UW trace. I further

Table 5.4: City cooperative caching performance.

	Hierarchical	Hashed	Directory
Arrival Rate	r_1 = 150M/day r_2 = 30M/day	30M/day	37M/day
Latency	0.86 secs	0.88 secs	0.89 secs
Storage	11 TB	1.5 TB	9.5 TB

assume that the average last-byte latency to transfer a document: (1) from an organizational proxy cache to a client is 10ms, (2) between two random caches in the city is 50ms, (3) between two random caches in the state is 100ms, (4) and between two random caches on the west coast is 500ms. I derived these numbers by multiplying a basic latency by the number of round trips required to download an average document. For the basic latency, I used ping latencies from the University of Washington to popular Web sites whose distances correspond to the three levels of the cache hierarchy. Since the average document size in the trace is 7.7KB, I estimate that five round trips between the sender and the receiver of the document are required to complete the transfer (due to TCP/IP protocol overhead).

Because these latencies are based upon a simple model of the network (e.g., persistent connections might reduce the number of round trips if the connection has been ramped up to a high congestion window) and pings from a single network source, I also evaluated the sensitivity of the model results to these parameters. I did this evaluation indirectly by varying the average latency for downloading documents from servers, thereby changing the ratio of cache latency to server latency. In addition to the trace value of $E(L)$=1.9 seconds, I also evaluated the schemes using average documents latencies ranging from 250ms to 10 seconds. In each case, their relative performance was qualitatively similar when $E(L)$=1.9 seconds.

Tables 5.4, 5.5 and 5.6 present model results for the three cooperative-caching schemes using the parameterizations above. From these tables I can make a number of conclusions. First, the bulk of the achievable benefit, in terms of latency savings, is already achieved at the scale of city-level cooperative caching. Indeed, the minimum possible average latency is $(1 - p_c)E(L)$, which

Table 5.5: State cooperative caching performance.

	Hierarchical	**Hashed**	**Directory**
Arrival Rate	r_1 = 1.37B/day r_2 = 147M/day r_3 = 29.5M/day	29.5M/day	37.7M/day
Latency	0.79 secs	0.83 secs	0.85 secs
Storage	11 TB	150 G B	9.5 TB

Table 5.6: West Coast cooperative caching performance.

	Hierarchical	**Hashed**	**Directory**
Arrival Rate	r_1 = 13B/day r_2 = 1.37B/day r_3 = 147M/day r_4 = 29.5M/day	29.5M/day	37.9M/day
Latency	0.78 secs	1.1 secs	1.13 secs
Storage	11 TB	15 GB	9.5 TB

for my parameters is 0.76 seconds. All three schemes already achieve a value close to this in the city. Second, broadening the region to increase population also increases inter-proxy latencies. As a result, a flat cooperative-caching scheme is no longer effective. For the west coast, the average latency between proxy caches is sufficiently large that the performance of the flat hash-based and directory-based schemes is worse, in terms of document latency, than their performance at the level of the state, despite the ten-fold increase in population. Obviously, this problem could be solved by designing hierarchical variants of these two schemes.

In terms of request rate, all schemes the lowest-level proxy caches have similar request rates (though the directory scheme has a slightly higher value), and are dominated by the requests from clients served directly by that proxy. However, even at the scale of city-wide cooperative caching, the top-level cache in the hierarchical scheme is a bottleneck. Since request rate will be directly correlated with queueing, the average latency that will be observed in the hierarchical scheme will

be significantly higher than shown here, particularly as the system scales up to the state or west coast level. Therefore, if scaling up to these levels is desirable (which is itself a questionable proposition at best), the load at the higher levels of the hierarchy must be distributed across multiple proxy caches. There are a number of fairly obvious and natural ways to do this. Finally, the hash-based scheme has the advantages that each document is stored only in one proxy cache and the load is balanced across the caches.

In summary, all three schemes perform well in the region where cooperative caching is advantageous (e.g., at the level of a medium-sized city). Since documents are stored only in one cache, a hash-based scheme achieves the best storage efficiency. In a broader area (e.g., the size of the West Coast of the U.S.), the increased latency of inter-proxy communication eclipses the very limited benefits of increased population.

5.4 Conclusions

This chapter explored the effectiveness of cooperative Web proxy caching using an analytic model of the steady-state behavior of Web proxy caches. Using the model, I evaluated the effectiveness of cooperative Web proxy caching at population sizes that range from a single organization to 100s of millions of clients, explored the tradeoffs among various cooperative caching schemes, and speculated on the performance implications of future trends in Web workloads.

Fundamentally, the usefulness of cooperative Web proxy caching depends upon the scale at which it is being applied. From the model results, cooperative Web proxy caching is most effective for small individual caches that together comprise user populations in the tens of thousands. At such small scales, any reasonable cooperative caching scheme will serve. But cooperative caching is not required for user populations of this size. If it is administratively and politically feasible, a single proxy cache can provide the same benefits with fewer resources and less overhead.

Whether or not they use cooperative caching locally, large organizations should use proxy caching for their user populations. A key issue is whether these large organizational caches benefit from cooperating. Experiments with the steady-state model indicate that cooperation among the organizational caches within a medium to large city will still provide benefit, although an incremental benefit, over cooperative caching at small scales. Assuming that bandwidth within a city is

plentiful and latencies are small, the overhead of cooperative caching would be low and therefore worth the secondary benefits that such caching provides. In principle, the organizational caches within a city can use a hash-based scheme to maximize storage efficiency. In practice, however, given the cheap cost of disks, using a hash-based scheme to spread load is more important than storage efficiency. Extrapolating to yet larger scales, such as the state level and even the west coast of the U.S., the model results indicate that cooperative caching among cities would provide very limited additional benefit, particularly in the face of increased latencies among caches.

Finally, note that these results on cooperative caching are based upon Web workload behavior currently observed. Fundamental shifts in Web workloads might change these results. For example, the trace I have used to parameterize the model consists primarily of static documents. [Wolman et al. 99a] has also observed a growing presence in Web workloads of streaming multimedia traffic, and streaming multimedia objects have different characteristics than static objects. Their average size is orders of magnitude larger, so cooperative caching for storage efficiency becomes more appealing. Furthermore, last-byte latency is not a critical performance metric for streaming data. Instead, reducing jitter and making more effective use of the network become more important. Lastly, given the sizes of streaming objects, and the relatively long period of time over which they are transferred over the network, transport optimizations like multicast might prove more effective.

Chapter 6

Conclusions

This dissertation uses both analytic techniques and experimental measurements to explore cooperative caching system performance in three new domains:

- I use a queueing network model and trace-driven simulation to explore the performance tradeoffs of using load balancing in cooperative caching systems.

- I present the design, implementation, and evaluation of a combined cooperative prefetching and caching system.

- I introduce a novel analytic model of Web accesses to explore the effectiveness of cooperative caching among World Wide Web proxies at scales ranging from hundreds to millions of clients.

The following sections summarize the contributions of this dissertation in each of these new domains.

6.1 Load balancing in cooperative caching systems

A crucial issue in the implementation of cooperative caching systems is the selection of the target nodes to receive evicted pages. Existing systems have selected targets based on the ages of their memory pages, effectively selecting the oldest pages in the network for replacement. However, using age information alone to make page replacement decisions can lead to significant server contention when there is a concentration of old pages at one or a few nodes. This contention increases the latency of remote page requests, thereby decreasing overall application performance. To reduce contention, cooperative caching systems can use load balancing to reduce server contention and

limit the impact of request traffic on local jobs on memory server nodes. Using an analytic queueing network model, I show the extent to which server load can degrade remote memory request latency and how load balancing alleviates this problem.

There are tradeoffs to the use of load balancing, however. Replacements made to balance load can cause the system to deviate from its original replacement algorithm. This deviation from policy can increase the overall number of disk accesses when load balancing is used, mitigating the benefits of load balancing. To evaluate this tradeoff, I use trace-driven simulation to measure the impact on application performance of deviating from the cooperative cache replacement policy. I find that deviating from the strict replacement policy, even substantially for some applications, does not significantly degrade overall application performance. Based upon these results, I conclude that cooperative caching systems can benefit considerably from load balancing requests with little harm from suboptimal replacement decisions. Finally, I use the intuition gained from the experiments to propose a new family of algorithms that incorporate load considerations as well as age information in global memory replacement decisions.

6.2 Cooperative prefetching and caching

Cooperative caching benefits memory-intensive applications, applications whose virtual memory and file buffer cache working sets are larger than the memory resources of a single node. By caching evicted pages in the remote memories of idle nodes instead of on local disk, cooperative caching transforms slow disk faults into much faster network faults. However, I/O-bound applications, either with large working sets or a small degree of reuse, only benefit to a limited degree from cooperative caching.

Cooperative caching is just one approach for reducing disk stall time; recent research has focused on two others, prefetching data from disk into memory, and striping data over multiple disks or nodes in distributed file and storage systems. To further improve the performance of cooperative caching systems, I propose a system that synthesizes of all three approaches for reducing disk stall time into a novel *cooperative prefetching and caching* system — the use of network-wide memory to support prefetching, caching, and parallel I/O in the presence of optional hints of future demands.

In this dissertation, I present the design, implementation, and evaluation of a prototype coop-

erative prefetching and caching system called PGMS (Prefetching Global Memory System). The goal of the system is to minimize the total cost of all memory references in the cluster. For issuing prefetches, the system uses a novel hybrid prefetching algorithm that combines aggressive prefetching into the cooperative cache with more conservative prefetching into local memory. For making replacement decisions in the cooperative cache, the system uses a provably near-optimal variant of the classic Longest Forward Distance algorithm extended to a three-level memory hierarchy.

I evaluate the PGMS prototype on a cluster of DEC Alpha workstations connected by a 1.28 Gb/sec Myrinet network. My measurements and analysis show that by using available global resources, cooperative prefetching can obtain significant speedups for I/O-bound programs. For example, for a graphics rendering application, PGMS achieves a speedup of 4.9 over a non-prefetching version of the same program. Furthermore, I show that: (1) the use of prefetching significantly increases application performance beyond the use of cooperative caching alone; (2) global prefetching provides substantial speedup over local, conventional prefetching alone; (3) parallel disk prefetching leverages the aggregate disk I/O bandwidth in the cluster, scaling application performance with cluster size; (4) prefetching over the network from global to local memory has much less impact than disk prefetching; and (5) the system isolates prefetching and non-prefetching applications, allocating system resources among them fairly.

6.3 Cooperative caching in wide-area networks

Previous work has shown that cooperative caching can be used to improve the performance of file and virtual memory systems in a high-speed, local-area network environment, and this dissertation has shown that load balancing and prefetching are two useful techniques for further improving the performance of these systems. Unfortunately, the algorithms, mechanisms, and policies used in cooperative caching systems for workstation clusters, both in previous work and in the initial parts of this dissertation, do not scale beyond hundreds of nodes due to assumptions inherent in local-area network performance. The last part of this dissertation explores the fundamental question of scaling cooperative caching to very large, wide-area distributed systems. In particular, it examines the potential of using cooperative caching among World Wide Web proxies deployed in the Internet.

Although several cooperative caching techniques have been proposed for use in the Web, the po-

tential performance benefits of cooperative caching in large-scale Web environments has not been previously evaluated. In this dissertation, I use a novel analytic model of Web accesses to explore the effectiveness of cooperative caching among World Wide Web proxies. This model evaluates the steady-state performance of proxy caches based upon client request, sharing, and document rate of change distributions for small-scale organizational populations to large-scale regional populations. With this model, I demonstrate that the utility of cooperative Web proxy caching fundamentally depends upon the scale at which it is applied. It is most effective among groups of small organizations, incrementally beneficial among large organizations within a medium-sized metropolitan area, and only marginally useful for larger client populations.

6.4 Future Research Directions

This dissertation has demonstrated that local-area cooperative caching systems can benefit considerably from incorporating load balancing and prefetching techniques into the system. Although cooperative caching was originally designed to globally manage the memory resources of a cluster, I have shown that managing the CPU and disk resources as well can further increase application performance. This dissertation has also explored the potential of using cooperative caching techniques in wide-area networks, showing that cooperative Web proxy caching is effective within limited population bounds. These results suggest a number of potential directions for future work:

Combining load balancing with prefetching and caching

This dissertation studied load balancing in isolation from prefetching in local-area cooperative caching systems. The proposed load balancing algorithm could be implemented as an enhancement to the prefetching and caching algorithm implemented in the PGMS prototype.

Modeling document rate of change

Experiments with the model of Web accesses showed that the steady-state performance of the caching system depends heavily on the rate of change of documents in the Web. I modeled document rate of change conservatively using an exponential distribution, although the data strongly suggests that it is heavy-tailed. In addition, the trace analyses used to determine the rate of change of Web documents was only approximate. Improving the model to use a

more accurate distribution for document rate of change and using server traces in addition to client traces to derive the distribution would further improve the analyses and conclusions of cooperative caching for the Web.

Streaming media

The analyses and conclusions regarding cooperative caching for the Web were made for workloads dominated by relatively static content downloaded using RPC transactions. However, trace analysis suggests that streaming media is becoming a significant part of Web workloads [Wolman et al. 99a]. The characteristics of streaming media and the manner in which streaming media is downloaded are fundamentally different than the characteristics and download mechanisms of static content. For example, the average size of streaming media objects is orders of magnitude larger than static content, so cooperative caching for storage efficiency becomes more appealing. Furthermore, last-byte latency is not a critical performance metric for streaming data. Instead, reducing jitter and making more effective use of the network become more important. Lastly, given the sizes of streaming objects, and the relatively long period of time over which they are transferred over the network, transport optimizations like multicast might prove more effective.

6.5 Final Remarks

This dissertation has shown that local-area cooperative caching systems can benefit considerably by managing all of the resources in clusters in concert. Previous work in cooperative caching systems naturally focused on managing cluster memory resources, enabling sharing of cache contents and efficient coordination of cache resources. In this dissertation I show that cooperative caching systems can gain significant performance benefits by managing cluster CPU and disk resources as well. Using load balancing techniques, cooperative caching systems can reduce CPU contention of remote page requests at memory servers. Using prefetching techniques, cooperative caching systems can leverage idle disk resources in the cluster to increase I/O bandwidth between cluster disk and memory and further reduce disk stall time for I/O-bound applications. The performance benefits of managing cluster memory, CPU, and disk resources together further demonstrates the advantages of managing workstation clusters more as loosely-coupled multicomputers than individual machines.

This dissertation has also shown that Web proxy caching in wide-area networks can benefit from the use of cooperative caching techniques, but only to a limited degree. Large-scale cooperative Web caching systems achieve most of their performance gains at the scale of a medium to large city. Although there have been a number of various large-scale cooperative Web caching algorithms proposed, this evaluation suggests that Web caching algorithms do not need to scale beyond the bounds of a metropolitan area. Instead, further efforts on Web caching should, for example, focus more on techniques for improving the cacheability of Web objects, particularly with the advent of new types of content such as streaming media and XML databases.

Bibliography

[Almeida et al. 96a] J. Almeida, V. Almeida, and D. Yates. Measuring the behavior of a World Wide Web server. Technical Report BU-CS-96-025, Boston University, October 1996.

[Almeida et al. 96b] V. Almeida, A. Bestavros, M. Crovella, and A. deOliveira. Characterizing reference locality in the WWW. Technical Report 96-011, Boston University, June 1996.

[Anderson et al. 96] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[Anderson et al. 98] D. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrin et. In *Proceedings of the 1998 USENIX Technical Conference*, June 1998.

[Arlitt et al. 96] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Proc. of the ACM SIGMETRICS '96 Conf.*, pages 126–137, May 1996.

[Bartels et al. 99] G. Bartels, D. Anderson, J. Chase, A. Karlin, H. Levy, and G. Voelker. Potentials and limitations of fault-based markov prefetching for virtual memory pages. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99), Extended Abstract*, Atlanta, GA, May 1999.

[Belady 66] L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 1966.

[Boden et al. 95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su. Myrinet - a gigabit-per-second local area network. *IEEE Micro*, February 1995.

[Breslau et al. 99] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. of IEEE INFOCOM '99*, pages 126–134, March 1999.

[Caceres et al. 98] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: The devil is in the details. In *Workshop on Internet Server Performance*, pages 111–118, June 1998.

[Cao 98] P. Cao. Characterization of Web proxy traffic and Wisconsin proxy benchmark 2.0. http://www.cs.wisc.edu/ ~cao/w3c-webchar-position, November 1998.

[Cao et al. 93] P. Cao, S. B. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. In *Proceedings of the 20th Annual International Symposium of Computer Architecture*, pages 52–63, May 1993.

[Cao et al. 95] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of intergrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 188–197, May 1995.

[Cao et al. 96] P. Cao, E. W. Felten, A. Karlin, and K. Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems (TOCS*, November 1996.

[Cao et al. 97] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. of the 1st USENIX Symp. on Internet Technologies and Systems*, pages 193–206, Dec. 1997.

[Carey et al. 93] M. J. Carey, D. J. Dewitt, and J. F. Naughton. The OO7 benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 12–21, May 1993.

[Chamberlain et al. 95] B. Chamberlain, T. DeRose, D. Salesin, J. Snyder, and D. Lischinski. Fast rendering of complex environments using a spatial hierarchy. Technical Report 95-05-02, Department of Computer Science, University of Washington, May 1995.

[Chang et al. 99] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 1–14, New Orleans, LA, USA, February 1999.

[Chankhunthod et al. 96] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proc. of the 1996 USENIX Technical Conf.*, pages 153–163, San Diego, CA, January 1996.

[Comer et al. 90] D. E. Comer and J. Griffoen. A new design for distributed systems: The remote memory model. In *Proceedings of the 1990 Summer USENIX Technical Conference*, pages 127–135, June 1990.

[Crovella et al. 96] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. In *Proc. of the ACM SIGMETRICS '96 Conf.*, pages 160–169, May 1996.

[Cunha et al. 95] C. R. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, Boston University, July 1995.

[Dahlin 99] M. D. Dahlin. Interpreting stale load information. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, May 1999.

[Dahlin et al. 94] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of the Conf. on Operating Systems Design and Implementation*. USENIX, November 1994.

[Danzig 98] P. Danzig. NetCache architecture and deployment. In *Proc. of the 3rd Int. WWW Caching Workshop*, http://wwwcache.ja.net/events/workshop/01/NetCache-3_2.pdf, June 1998.

[Douglis et al. 97] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proc. of the 1st USENIX Symp. on Internet Technologies and Systems*, pages 147–158, Dec. 1997.

[Duska et al. 97] B. Duska, D. Marwood, and M. J. Feeley. The measured access characteristics of World Wide Web client proxy caches. In *Proc. of the 1st USENIX Symp. on Internet Technologies and Systems*, pages 23–36, Dec. 1997.

[Eager et al. 86a] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adapative load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12:662–675, 1986.

[Eager et al. 86b] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation Review*, 16:53–68, 1986.

[EMC 99] EMC Corporation, http://www.emc.com/products/enterprise_storage_systems/systems.htm. *Symmetrix 3000 and 5000 Enterprise Storage Systems Product Description Guide*, 1999.

[Fan et al. 98] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A scalable wide-area web cache sharing protocol. In *Proc. of ACM SIGCOMM '98*, Vancouver, BC, August 1998.

[Feeley et al. 95] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 201–212, Dec. 1995.

[Feldmann et al. 99] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proc. of IEEE INFOCOM '99*, March 1999.

[Felten et al. 91] E. W. Felten and J. 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.

[Flouris et al. 98] M. D. Flouris and E. P. Markatos. The network ramdisk: Using remote memory on heterogeneous NOWs. Technical Report 226, ICS-FORTH, August 1998.

[Fox et al. 97] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, San Malo, France, October 1997.

[Franklin et al. 92] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *Proceedings of the 18th VLDB Conference*, pages 596–609, August 1992.

[Gibson et al. 97] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Reidel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '97)*, June 1997.

[Gibson et al. 99] G. A. Gibson, D. F. Nagle, W. C. II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD scalable storage systems. In *Extreme Linux Workshop, USENIX '99*, Monterey, CA, June 1999.

[Glass et al. 97] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of ACM SIGMETRICS 1997*, June 1997.

[Glassman 94] S. Glassman. A caching relay for the World Wide Web. In *Proc. First Int. World Wide Web Conf.*, pages 60–76, May 1994.

[Gribble et al. 97] S. D. Gribble and E. A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proc. of the 1st USENIX Symp. on Internet Technologies and Systems*, pages 207–218, Montery, CA, December 1997.

[Harchol-Balter et al. 97] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, August 1997.

[Hartman et al. 95] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):279–310, August 1995.

[Hartman et al. 99] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm scalable storage system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, June 1999.

[Ioannidis et al. 98] S. Ioannidis, E. P. Markatos, and J. Sevaslidou. On using network memory to improve the performance of transaction-based systems. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 363–370, Las Vegas, NV, USA, July 1998.

[Jamrozik et al. 96] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. E. II, A. R. Karlin, H. M. Levy, and M. K. Vernon. Reducing network latency using subpages in a global memory environment. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[Karger et al. 99] D. Karger, T. Leighton, D. Lewin, and A. Sherman. Web caching with consistent hashing. In *Proc. of the 8th Int. World Wide Web Conf.*, May 1999.

[Kimbrel et al. 96a] T. Kimbrel and A. Karlin. Near-optimal parallel prefetching and caching. In *Proceedings of the 37th IEEE Annual Symposium on Foundations of Computer Science*, pages 540–549, October 1996.

[Kimbrel et al. 96b] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. Gibson, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 19–34, October 1996.

[Koussih et al. 99] S. Koussih, A. Acharya, and S. Setia. Dodo: A user-level system for exploiting idle memory in workstation clusters. In *Proceedings of the 8th IEEE International Symposium on High-Performance Distributed Computing (HPDC-8)*, August 1999.

[Krishnan et al. 98] P. Krishnan and B. Sugla. Utility of co-operating Web proxy caches. In *Proc. Seventh Int. World Wide Web Conf.*, April 1998.

[Kroeger et al. 96] T. M. Kroeger, J. C. Mogul, and C. Maltzahn. Digital's Web proxy traces. ftp://ftp.digital.com/pub/DEC/ traces/proxy/webtraces.html, August 1996.

[Kroeger et al. 97] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proc. of the 1st USENIX Symp. on Internet Technologies and Systems*, pages 13–22, Dec. 1997.

[Lawrence et al. 99] S. R. Lawrence and C. L. Giles. Accessibility of information on the Web. *Nature*, 400(6740):107–109, July 1999.

[Lazowska et al. 84] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[Leach et al. 83] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1(5), November 1983. 842–857.

[Lee et al. 96] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[Litzkow et al. 88] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computer Systems,*, pages 104–111, San Jose, CA, June 1988.

[Mah 97] B. A. Mah. An empirical model of HTTP network traffic. In *Proc. of IEEE INFOCOM '97*, pages 592–600, April 1997.

[Markatos 96] E. P. Markatos. Using remote memory to avoid disk thrashing: A simulation study. In *Proceedings of the ACM International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '96)*, pages 69–73, San Jose, CA, USA, February 1996.

[Markatos et al. 96] E. P. Markatos and G. Dramatinos. Implementation of a reliable remote memory pager. In *Proceedings of the 1996 USENIX Technical Conference*, pages 177–190, San Diego, CA, USA, January 1996.

[Menaud et al. 98] J.-M. Menaud, V. Issarny, and M. Banatre. A new protocol for efficient transversal Web caching. In *Proc. of the 12th Int. Symp. on Distributed Computing*, pages 288–302, September 1998.

[Michel et al. 98] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching: Towards a New Global Caching Architecture. *Computer Networks and ISDN Systems*, 30(22–23):2169–2177, November 1998.

[Mitzenmacher 97] M. Mitzenmacher. How useful is old information. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC '97)*, pages 83–91, 1997.

[Mogul 95] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report 95/5, DEC Western Research Laboratory, October 1995.

[Mowry et al. 96] T. Mowry, A. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '97)*, pages 3–17. USENIX, October 1996.

[Pai et al. 98] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network service. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[Patterson et al. 88] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data*, pages 109–116, June 1988.

[Patterson et al. 95] R. H. Patterson, G. A. Gibson, E. Gintnig, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[Rabinovich et al. 98] M. Rabinovich, J. Chase, and S. Gadde. Not all hits are created equal: Cooperative proxy caching over a wide area network. In *Proc. of the 3rd Int. WWW Caching Workshop*, June 1998.

[Rochberg et al. 97] D. Rochberg and G. Gibson. Prefetching over a network: Early experiences with ctip. *Performance Evaluation Review*, 25(3):29–36, December 1997.

[Rosenblum et al. 91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[Rousskov et al. 99] A. Rousskov, D. Wessels, and G. Chisholm. The first ircache web cache bake-off. Technical report, National Laboratory for Applied Network Research, April 1999.

[Sarkar et al. 96] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*. USENIX, October 1996.

[Schilit et al. 91] B. N. Schilit and D. Duchamp. Adaptive remote paging for mobile computers. Technical Report CUCS-004-91, Columbia University Computer Science Department, February 1991.

[Squid et] Squid internet object cache, http://squid.nlanr.net.

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

[Tewari et al. 99] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design considerations for distributed caching on the Internet. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, May 1999.

[Thekkath et al. 97] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP '97)*, October 1997.

[Tomkins et al. 97] A. Tomkins, R. H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '97)*, June 1997.

[Touch 98] J. Touch. The LSAM proxy cache - a multicast distributed virtual cache. In *Proc. of the 3rd Int. WWW Caching Workshop*, June 1998.

[Valloppillil et al. 98] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. ftp://ftp.isi.edu/internet-drafts/draft-vinod-carp-v1-03.txt, February 1998.

[Voelker et al. 97] G. Voelker, H. Jamrozik, M. Vernon, H. Levy, and E. Lazowska. Managing server load in global memory systems. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, June 1997.

[Voelker et al. 98] G. M. Voelker, E. Anderson, T. Kimbrel, M. J. Feeley, J. Chase, A. Karlin, and H. M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1998.

[vonEicken et al. 95] T. vonEicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 303–316, Copper Mountain Resort, CO, December 1995.

[Wills et al. 99] C. E. Wills and M. Mikhailov. Towards a better understanding of Web resources and server responses for improved caching. In *Proc. of the Eighth Int. World Wide Web Conf.*, pages 153–165, May 1999.

[Wilson et al. 94] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, C.-W. Tseng, M. W. Hall, M. S. Lam, , and J. L. Hennesy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[Wolman et al. 99a] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of Web-object sharing and caching. In *Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems*, Oct. 1999.

[Wolman et al. 99b] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP '99)*, pages 16–31, December 1999.

[Yocum et al. 97] K. G. Yocum, J. S. Chase, A. J. Gallatin, and A. R. Lebeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–252, August 1997.

Vita

Geoffrey Michael Voelker was born on September 12, 1970 in Riverdale, Maryland, and was raised in Livermore, California. In 1988 he attended the University of California at Berkeley, where he received his B.S. degree with High Honors in Electrical Engineering and Computer Science in 1992. He then attended the University of Washington, where he received his M.S. degree in Computer Science and Engineering in 1995 and his Ph.D. degree in Computer Science and Engineering in 2000.