Architectures and Algorithms for
Scalable Wide-area Information Systems

by

Renu Tewari, B.E.,M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1998

Architectures and Algorithms for

Scalable Wide-area Information Systems

Approved by
Dissertation Committee:

To My Family

Acknowledgments

I owe a debt of gratitude to many who, in varying but substantial ways, have contributed in making this endeavor possible. The journey may have seemed long and daunting at first, but with the guidance and support that I was privileged to receive, I leave with only the fondest of memories.

Foremost, I would like to thank my advisor, Prof. Harrick Vin, for his guidance, patience, and most of all, his example. His contributions to this work are too many to enumerate. His constant striving for perfection and his emphasis on detail have been invaluable and inspiring. A large part of what I have learnt in graduate school is due to him.

My sincere thanks to Prof. J. C. Browne, Prof. Mike Dahlin, Prof. Don Fussell, Prof. Al Mok, and Dr. Dan Dias for being on my doctoral committee. They have provided insights and ideas that have contributed greatly to my research. Dr. Dan Dias, despite his many responsibilities, found time to guide my research while I was at IBM. His cheerful spirit, constant encouragement, and keen perception made even the "real world" seem accessible. Prof. Mike Dahlin played a crucial role in shaping and guiding my work on distributed caching. He not only influenced my approach to problem solving, but also brought attention to zero-errors in graph plotting! I thank Prof. Browne for his support and assistance at the beginning of my research work. His upbeat spirit has always been a morale booster. Thanks also to Prof. Greg Plaxton for several illuminating discussions on the theoretical aspects of caching problems.

Nothing could have been achieved without the help and contributions of several colleagues and friends. I owe them all my sincerest thanks. A large part of the work on clustered servers was done with Rajat Mukherjee at IBM. Asit Dan and Dinkar Sitaram were instrumental in the work on resource-based caching. Jon Kay helped in the implementation and late-night debugging of the Tsquid cache code. The theoretical analysis of push caching was done by Madhukar Korupolu, who was a valued member of the Friday afternoon cache club. Pawan Goyal contributed, both directly and indirectly, with his critical comments and insightful suggestions. Prashant Shenoy, who effortlessly and without much complaint bore the burden of system administration, was of invaluable help. He was always there to make the right software/hardware coexist when it was most critically needed, and patiently tolerated my expanding disk space usage. My sincere thanks to Gloria Ramirez, our Graduate Coordinator, whose phenomenal efficiency and patience made all the bureaucratic hassles of UT seem effortless.

I have benefited from many fun discussions, wise or otherwise, with my colleagues at the Distributed Multimedia Computing Laboratory (DMCL)—Balaji Balasubramanyan, Xingang Guo, Amir Husain, Jon Kay, Scott Page, Ed Posnak, Sriram Rao, Jasleen Sahni and T R. Vishwanath. Those in the lower lab have also been kind to ignore my increasing entropy and the gradual encroachment of space on tables and chairs.

My stay in Austin was made memorable by a number of friends and ex-friends! Being with Ajita, Anu, Archana, Deepa, and Nina was always a comfort and a joy. The long discussions on feminism with Ajita, the wonderful time with Anu as roommates, and the enjoyable tea sessions with Nina were undoubtedly the best moments I had in Austin. The numerous arguments and puzzle sessions with Shashidhar, Nandu and Rajan were a special treat. For the passionate defense of the New Yorker, the art movies at village cinema, the wonderful culinary experiments, and for just being there when needed, a heartfelt thanks to Rajan. Discussions with Rajmohan and Pawan on the technical and the Times were both fun and enlightening. For the 4 p.m. coffee tradition, the pointed and pointless discussions on life and philosophy, the debates over academia and industry, Prashant gets all the blame and some thanks. For her warmth, genuine concern, and all the mouth-watering treats, a special thanks to Neha.

Lastly, but most importantly, I would like to thank my family who were with me through all the ups and downs and who, in the end, made all of this worthwhile. My parents have provided love, encouragement, and inspiration throughout the course of my graduate studies and my entire life. This dissertation would not have been possible without the support and strength of my husband, Anant, who put up with long years of separation without a grudge. I could not have come this far without him.

RENU TEWARI

The University of Texas at Austin
August 1998

Architectures and Algorithms for Scalable Wide-area Information Systems

Publication No. ____________________

Renu Tewari, Ph.D.
The University of Texas at Austin, 1998

Supervisor: Harrick M. Vin

In recent years, the World Wide Web has become the dominant wide-area information system that enables seamless access to multimedia information over the Internet. The simplicity and global reach of the web has enhanced the ease of exchanging, accessing, and viewing different types of data; it has provided the impetus for Internet connectivity to become truly world-wide. This has resulted in the proliferation of new applications for wide-area information systems. These applications require the underlying infrastructure to scale to a large number of users, substantially reduce response times, support the requirements of heterogeneous data types, and guarantee data availability to the users. The current web infrastructure built using monolithic servers, best-effort service, and isolated proxy servers is inadequate for meeting these requirements.

The focus of this dissertation is to design an infrastructure for a scalable wide-area information system with support for managing a large number of heterogeneous objects, maintaining low response times, and guaranteeing high availability. To achieve these goals, we focus on the design and implementation of scalable servers and scalable delivery architectures.

The scalable server design consists of a clustered server that eliminates hot-spots by load balancing, supports requirements of heterogeneous data types, and stores redundant data

to mask disk and node failure. Conventional techniques were designed for load balancing on a disk array in a single node system and are not suited for multi-node clusters. For load balancing, we propose: (i) data layout techniques for storing data across disks and nodes, and (ii) mechanisms for balancing the load of client connections across the nodes. Next, we design an analytical model of the clustered server that is used to determine the optimal data layout parameters (e.g., stripe unit size, striping width, etc.) for heterogeneous data types. The model also provides insight into the interaction of the interconnection network with the storage disks and its combined effect on the data layout parameters. For masking failures, existing redundant storage techniques neither scale to large number of users nor support the requirements of heterogeneous data types. We design redundant data placement techniques that have the following features: (i) balance the load during normal and failure mode operations, (ii) minimize the storage and computation overhead, (iii) support optimizations for different data types, and (iv) scale to a large number of users.

Since effective caching techniques are the cornerstone for handling exponential growth and improving performance, the scalable delivery architecture consists of a distributed caching and replication system. We design and implement a distributed cache architecture to achieve the following goals: (i) improve performance, (ii) increase scalability, and (iii) support different data types. To address the goal of improving performance, each cache in the distributed cache architecture stores location hints to reduce the delay in locating and accessing information. Further, we develop dynamic push caching algorithms to move the data closer to clients, thereby, improving performance. For scalability, we design a self-configuring meta-data hierarchy that is used to efficiently propagate locations hints among an increasingly large number of caches without incurring high space and bandwidth overheads. To support heterogeneous data types, we design a cache replacement policy that manages the cache resources of space and bandwidth among multiple data types. Experimental results show that using these techniques results in a factor of about 2.5 speedup in average client response times. We provide details of a prototype implementation that incorporates these techniques and is deployed on the Internet.

Contents

Chapter 1

Introduction

I think there is a world market for maybe five computers.

— Thomas Watson, Chairman of IBM, 1943

640KB ought to be enough for everybody.

— Bill Gates, Chairman of Microsoft, 1981

1.1 Motivation

In recent years, the World Wide Web has become the dominant wide-area information system that enables easy access to multimedia information over the Internet. The Web, as it currently exists, is used mostly as an information dissemination system. Information or applications residing at a server are down-loaded and executed at a client using a request-response protocol. The ease-of-use and global reach of the Web has resulted in two beneficial effects. First, it has triggered the connectivity of the Internet to become truly world-wide. Second, it has enhanced the scope and ease of exchanging, accessing, and viewing different types of data. This has resulted in the proliferation of new applications that exploit the global scope of the Web. Unfortunately, the requirements of these applications have exposed the limitations of the Web infrastructure. To illustrate, consider the following requirements of some current and emerging applications.

- *Scale*: A Web site hosting information on a popular sporting event such as the Olympics requires support for millions of users and around 10 to 100 million accesses/day. E-commerce applications that enable online trading, purchasing, and auctioning of products support concurrent accesses from millions of users. It is envisioned that in the near future the number of users and the demand per user will grow by two orders of magnitude [86].

- *Heterogeneity*: Numerous applications require integrating 3D graphics, real-time audio/video, text, and images. For example, consider the Virtual Emergency Task Force, a virtual meeting environment that enables experts to meet virtually and handle a crisis situation; or a lecture-on-demand system that facilitates distance learning and self paced education. The different types of data that these applications support impose different requirements for access and delivery. While best-effort delivery suffices for images and text, audio and video data require real-time delivery. Along with managing the requirements of heterogeneous data types, applications need to support different end-station capacities and connectivity. The end-station capacities could range from mobile PalmPilots to an enterprise server, while connectivities could range from low bandwidth dial-up lines to a high bandwidth connection of an organization.

- *Performance Guarantee*: Interactive applications such as search engines, online purchasing and trading require low response times (order of 200 ms). Financial trading environments and video-conferencing applications require real-time updates and delivery guarantees.

- *Availability*: Information services (flight information, stock quotes, etc.), e-commerce, and collaborative applications require transparent failure detection and recovery. The transient or partial failure of the system should be masked from the user.

As we approach the next millennium, it is becoming evident that a new information environment is evolving— one that consists of millions of machines and users managing billions of globally distributed objects; one that allows users to interact, collaborate, and exchange information in any form easily, securely, and efficiently.

The current Web infrastructure built using monolithic servers, best-effort networks, and isolated caches and browsers is not designed support the requirements of this emerging environment.

1. Current servers are not designed to support the exponential growth in the number of users and the request rates. Sudden increases in the popularity of a server cause "hot spots" of server overload and network congestion. Since overload conditions develop and disappear in unanticipated ways, the brute force approach of over-provisioning the server or network is not an effective solution.

2. Servers and networks do not differentiate between the requirements of heterogeneous data types. A large fraction of the data stored at servers consists of non-textual data such as audio, video, animations, and images. Audio and video (referred to as continuous media) data has fundamentally different storage and access characteristics compared to textual data; it requires orders of magnitude more space on average, and requires real-time delivery.

3. The response times observed by users are an order of magnitude more than those acceptable for interactive tasks. There are no performance guarantees for real-time applications.

4. Failures are commonplace in the current system. Users are susceptible to both server and network failures.

To address the above limitations, it is widely recognized that the current Web infrastructure has to be redesigned.

The goal of this dissertation is to design an infrastructure for a wide-area information system with a primary focus on scalability; with support for the management of a large number of heterogeneous objects, maintenance of low response times, and high availability guarantees. The remainder of this chapter is organized as follows. Section 1.2 presents a brief survey on the history of information systems and related efforts on designing wide-area information systems. Section 1.3 discusses how the design of a wide-area information system is affected by considerations of scale. Section 1.4 summarizes the contributions and scope of this dissertation. Finally, Section 1.5 provides a brief outline of the rest of this dissertation.

1.2 Information Systems: A Historical Perspective

The advent of the information age began with the invention of the telegraph more than 150 years ago. The evolution of information systems can be traced with the growth of storage and communication technologies. The circuit-switched telephone system, that became intercontinental in the 1920's, was the first wide-area system for exchanging analog information. Pioneered by ARPA, the mid-sixties marked the beginning of computer networks— enabling computers to communicate and exchange information. Later, the growth of the Internet, the global network of computers[1], began with ARPANET in the 70's, to NSFNET, CSNET, BITNET in the 80's, and finally resulted in its commercialization in 1995. In what follows, we will illustrate with examples the different stages in the evolution of information systems.

The deployment of the UNIX operating system and the Ethernet in the early 80's triggered the growth of smaller, more powerful computers connected to local area-networks (LAN). While the Ethernet enabled fast communication over a LAN, UNIX provided a simple file abstraction for a container of information. The Network file system (NFS), using the file abstraction, enabled transparent access to information stored on nodes in a LAN. NFS adopted a client server model; the client was the machine that accessed the file and the server was the machine that stored and maintained the file.

With the expansion of the Internet, connectivity increased across wide geographic and organizational domains commonly called wide area networks (WAN). Communication across domains required a standard protocol for exchanging information. The Internet protocol (IP) became widely accepted as the base for simple, unreliable, connectionless delivery. Different applications used special purpose protocols over IP to enable different kinds of information exchange. The USENET News designed the Network News Transfer Protocol (NNTP) over IP for a distributed bulletin board system; news servers archived information and propagated new information (called postings) to participating servers using NNTP.

[1] The term Internet was officially defined only in 1995.

The adoption of the TCP/IP protocol for reliable, controlled, end-to-end communication resulted in a number of applications for accessing and exchanging information on the Internet. The File Transfer Protocol (FTP) allowed down-loading of entire files from/to servers across the Internet. The Andrew File System (AFS) extended the UNIX file access semantics to clients and servers across the Internet [63]. Unix file semantics required that any modifications to a file were reflected in a subsequent reads and files used a uniform namespace. AFS used less restrictive semantics for consistency. It used effective caching to maintain good performance while supporting thousands of clients. The Alex FTP filesystem merged the FTP protocol with the UNIX uniform namespace to enable transparent access to files from FTP servers around the world [15]. Apart from the client-server model of exchanging information, a client-client model allows direct exchange of information between clients. The E-mail application, based on this model, used the Simple Mail Transfer Protocol (SMTP) over TCP/IP, to exchange ASCII messages between pairs or groups of users across the Internet. This was extended to allow exchanging structured and non-ASCII information using the Multipurpose Internet Mail Extensions.

With the global reach of the Internet, faster transmission bandwidths, and faster disks with large capacities more information could be stored in computers around the world that could be accessed easily with low delay. With the standardization of compression algorithms, images, audio, and video information could easily be stored and exchanged. Global information systems had become a reality by the beginning of the 90's. Quoting from the Federal Networking Council resolution defining the Internet in 1995:

> *The "Internet" refers to the global information system that – (i) is logically linked together by a globally unique address space based on the Internet Protocol (IP) or its subsequent extensions/follow-ons; (ii) is able to support communications using the Transmission Control Protocol/Internet Protocol (TCP/IP) suite or its subsequent extensions/follow-ons, and/or other IP-compatible protocols; and (iii) provides, uses or makes accessible, either publicly or privately, high level services layered on the communications and related infrastructure described herein.*

A number of applications were developed in the early 1990's to enable easy access of globally distributed information. The Gopher system, built at the University of Minnesota, was one such information service that allowed users to explore, search, and retrieve information residing on servers across the world using a simple menu driven interface and the gopher protocol. Information accessed from gopher servers consists of text, binaries, directories, audio and images. Another example is the Wide Area Information Servers (WAIS) project that was developed jointly by Thinking Machines Corporation and others. WAIS is an electronic publishing system that allows querying, searching, and retrieving multimedia information from databases around the world [121]. It enables automatic indexing of databases and allows information from multiple databases to be accessible regardless of the format (e.g., text, images, spreadsheets, graphics, audio, video).

The WWW, developed initially at CERN, has become the most successful and widely used wide-area information system. The WWW offers a mechanism of accessing globally distributed documents connected together by hyper-links using a collection of protocols, Hypertext Transfer Protocol (HTTP), FTP, etc. Web servers store multimedia information which could be accessed by clients and displayed using a graphical user interface. To allow access to files stored at Web servers, using the UNIX file semantics, the WebNFS filesystem, an enhanced version of NFS, has recently been developed [13]. The original Web protocols were effective in locating and browsing multimedia data but lacked the ability to perform computation. The advent of the portable Java programming language helped address this limitation.

For the future, there is a thrust toward ubiquitous computing frameworks (or meta-systems) to make global computing as easy as global information access. Legion [79], Globus [66], Globe [117], WebOS [114], Bayou [95], etc., are a few such efforts. Discussion of these systems is beyond the scope of this dissertation.

1.3 Influence of Scale on Information System Design

The scale of a system has a fundamental influence on its design [106]. Mechanisms that work effectively in a small system become inadequate in the context of large systems. This, however, leaves open the question: Why is it necessary to focus on scale? To understand the need to scale one needs to understand the economy of growth. As a system grows enabling more users to communicate and share information, it becomes increasingly indispensable. At a critical point in the evolution of the system, the ability to use it is not a choice but a necessity. Growth, therefore, builds a momentum of its own. The economic impact of growth motivates the need to scale. Before instantiating the design principles of a scalable system, we formally define scalability in the context of wide-area information systems.

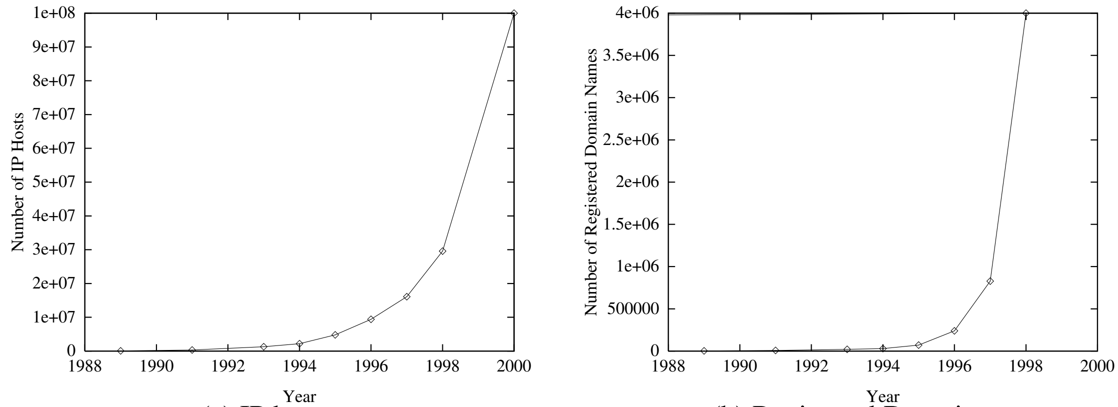
1.3.1 What is Scalability?

The term scale has been loosely used and interpreted in different contexts. A system is scalable if it can handle the addition of users and resources without suffering noticeable loss of performance or increase in administrative complexity [117]. Scale can be defined along three dimensions: (i) the number of users and resources (processing nodes, storage nodes, etc.), (ii) the diameter of the system (i.e., the distance between the farthest nodes), and (iii) the number of organizations that administer the system. Asymptotically, a scalable system is defined as having $O(n)$ resources for n users. In other words, the number of resources required to provide the same performance increases linearly with the number of users. Since increasing the number of resources may result in performance degradation (e.g., the response time or latency increase with the diameter of the system), a scalable system is defined as one that has an $O(log\ n)$ upper bound on performance degradation. The class of systems based on hierarchical architectures have the above property.

Any system built using centralized components is considered unscalable. Other examples of unscalable systems include those with single points of failure or with system-wide synchronization requirements.

1.3.2 Magnitude of Scale

Informally, in the current context, a system is said to be scalable if it can support millions of users or thousands of transactions per second. The statistics on the growth of the Internet and the Web serve as a good measure to quantify the magnitude of scale of a wide-area information system. While in 1985, NSFNET connected 5 supercomputing centers with a 56 Kbps network, today, the Internet consists of over 29 million hosts connected by backbone bandwidths of 45 to 622 Mbps. The Internet has grown at an exponential rate, doubling every year. The number of host IP addresses, as shown in Figure 1.1(a), has risen from 213 in 1981 to 29.6 million in mid 1998. The number of registered domains has grown from 2000 in 1989 to over 4 million in mid 1998 (as shown in Figure 1.1(b)).



(a) IP hosts (b) Registered Domains

Figure 1.1: The growth of the Internet. Source: Network Wizards Survey [125].

With the advent of the WWW in 1993, the number of Web sites have grown exponentially, doubling every 6 months. Figure 1.2 shows the growth of Web sites, from 623 in 1993 to 2.5 million by mid 1998. The Web traffic over the Internet has increased from 1 terabyte per day in 1995 to 133 terabytes per day in 1998.

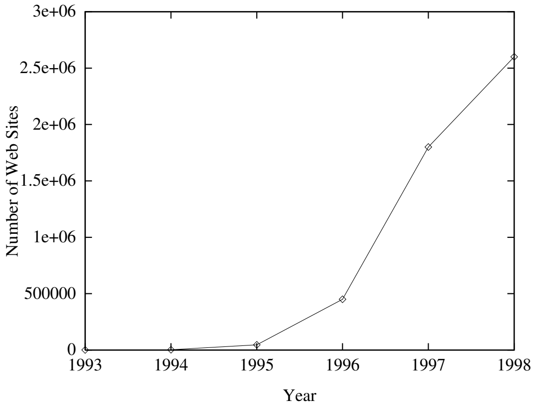


Figure 1.2: The Growth of the Web. Source: NetCraft Survey [86].

The growth of the Web and the Internet is not expected to plateau anytime in the near future. This suggests that there will an order of magnitude increase in the number of users and the amount of data by the year 2000. The current infrastructure is not designed to handle this magnitude of scale.

1.3.3 Design Principles for Scalability

The components of a wide-area information system are the (i) servers, (ii) interconnection networks, (iii) delivery architectures consisting of proxy servers or intermediary nodes, and (iv) clients. In a wide-area information system such as the Web the interconnection network is the Internet. Each component is composed of multiple resources (e.g., nodes, disks, memory etc.). To make a system scalable, it is necessary to make the individual components scalable and the interactions among them scalable as well. Based on the insights gained from examining existing wide-area systems, we enumerate the basic design principles for building a scalable wide-area information system.

- *Decentralize*: Any monolithic component becomes unwieldy and complex as the number of users increase. A single, centralized component has only a finite growth potential. To achieve scalability it is essential to have no centralized components.

- *Cache if possible*: Caching is a known technique to handle exponential growth [129]. Caching reduces load on servers and enables the the data to be available closer to the access location.

- *Move data and computation to the clients*: Clients, in general, have spare computation power. Thus, moving computation to the client reduces the server load. Similarly, data at the client reduces the network usage and improves the quality of service perceived by the client.

- *Reduce hot-spots*: Understanding the usage patterns enables identifying potential hot-spots that reduce scalability. It is beneficial to build strategies for data placement and replication based on knowledge of usage.

- *Minimize global information:* Any approach that requires global information, system-wide updates, shared state, and global co-ordination is inherently limited in scale.
- *Reconfigure dynamically and guarantee availability:* Scalable systems need to handle both resource failures and incremental growth. The system should automatically adapt to the increase or decrease in available resources or data. Failures should be masked from the user.
- *Minimize Trust:* Security can be maintained if the integrity of only a small number of resources has to be guaranteed. The more the number of resources to be trusted the less likely that they will remain secure.

1.4 Summary of Contributions

The focus of this dissertation is on the design of an infrastructure for a scalable wide-area information system that can manage a large number of heterogeneous objects, maintain low response times, and guarantee high availability. To achieve the above goal, we develop, design, and implement: (i) scalable servers and (ii) a scalable delivery architecture. The design of the other components of a wide-area information system, namely the Internet and the client systems is beyond the scope of this dissertation.

1.4.1 Scalable Server Design

Current Web servers are largely built using single node systems provide best-effort service for different types of objects with no availability guarantees to the user. This approach violates the basic design principles for scalable systems. Our scalable server design, based on *clustered servers*—a group of independent nodes connected by a scalable interconnect, has the following properties: (i) it has no central resource, (ii) it requires no global information or shared state, (iii) it supports incremental growth, (iv) it balances load across various nodes/disks, (v) it supports requirements of heterogeneous data types, and (vi) it guarantees data availability.

To address the design principles for scalability we build techniques for data placement that: (i) network-stripe the data across nodes for load balancing, and (ii) use a balanced random data placement to support incremental growth. Further, we design an analytical queuing model of the clustered server. The model has two benefits. First, it is used to analyze architectural choices for the cluster and determine the optimal striping parameters (e.g., stripe unit size, striping width, etc.) for the heterogeneous data types. Second, it provides an insight into the interactions of the interconnection network with the storage disks and its combined effect on the placement parameters.

To balance the load of client connections across the nodes, the existence of multiple nodes is made transparent to the clients. Conventional approaches balance the number of client connections across nodes, either by: (i) providing a list of node names to the client, or (ii) resolving a single name to multiple addresses. These approaches have two limitations: (i) they are susceptible to node failures, and (ii) provide poor load balancing. To achieve fine-grained load balancing, our approach provides a single name/address to the client and transparently routes the data to different nodes. The novel aspect of this technique is that it is limited to control-path routing (from the client to server) while the data path (from the server to the clients) circumvents the router. This eliminates router bottlenecks.

Increasing the number of nodes and disks in a cluster increases the overall probability of failure. To guarantee availability of data, redundant information is stored; either a replica of each block or parity information for a group of blocks (e.g., RAID). Existing parity placement techniques do not scale to large number of disks. We design parity placement schemes— balanced random parity and multi-stride parity placement, that: (i) balance the load during normal and failure mode operations, (ii) minimize the storage and computation overhead, (iii) support optimizations for different data types, and (iv) scale to a large number of users. To determine the optimal parity group size, we extend our analytical model to include failure mode operations.

1.4.2 Scalable Information Delivery Architecture

Geographically distributed servers and bursty accesses increase the network delays and congestion. This limits the system scalability and the performance observed by clients. Caching at intermediate nodes (called proxies) improves the scalability and performance by reducing:

the network/server traffic, and the distance traversed on the network. We design and implement a distributed cache architecture that has the the following goals: (i) improve performance, (ii) increase scalability, and (iii) support different data types. To address these goals, our design consists of distributed caches with location-hints that are stored locally, a self-configuring hierarchy for meta-data propagation, and cache replacement policies for multiple data types.

Based on measurements of several caches on the Internet and analysis of several traces of Web traffic, we derive four basic design principles for large-scale caches: (1) minimize the number of hops to locate and access data, (2) do not slow down misses, (3) share data among many caches, (4) cache data close to clients. Although these principles may seem obvious in retrospect, we find that current cache architectures routinely violate them at a significant performance cost. Using these principles, we design, simulate, and implement three strategies for organizing and managing large-scale, distributed caches that:

- Separate data paths from meta-data paths and maintain a hierarchy of meta-data that tracks where copies of data are stored.
- Maintain location hints so that caches can locate nearby copies of data without suffering network latencies. Use direct cache-to-cache data transfers to avoid inherent delays in conventional cache hierarchies.
- Use push caching to move data closer to the clients that have not yet referenced the data but are likely to do so in the near future.

Experimental results show that by using the above strategies, the response times improves by a factor of 1.3 to 2.3 compared to conventional cache organizations.

The goal of the self-configuring meta-data distribution hierarchy is to provide a fast and scalable way for a node to determine where to go for data that is not locally cached. Four aspects of the design improve scalability and performance: hierarchical propagation of hints, pruning the propagation of location hints, limiting the number and size of local hints, and automatically configuring the hierarchy. The design and implementation of the algorithm for self-configuring the hierarchy improves scalability by: (i) automatic reconfiguration on failure,

(ii) distribution of load among the caches by creating different hierarchies for different object groups, (iii) maintenance of locality of nodes within subtrees, and (iv) automatic configuration on incremental growth.

There are two factors that affect proxy cache resource management. First, objects stored in the cache have different sizes and occupy different amounts of cache space. Second, continuous-media objects are accessed in streaming mode instead of traditional down-load-and-play, and thereby require disk bandwidth reservations. Current proxy cache replacement algorithms, based on extensions of traditional memory caching, do not adapt to the resource requirements of heterogeneous data types. We design the *Resource-based Caching* (RBC) algorithm that makes replacement decisions based on the resource usage of different objects, and the benefit they add toward improving cache performance. The key aspects of RBC are selecting the unit of object to cache and determining what to replace based on the available cache resource. The idea is to model the cache as a two-constraint knapsack; use heuristics to convert the two-constraint knapsack to multiple single constraint knapsacks; and use the greedy approximation to solve a single constraint knapsack. We show that the performance of RBC is always an envelope of various other cache algorithms, ranging from 10-200% improvement for varying access parameters and cache capacities.

1.5 Outline of Dissertation

The remainder of this dissertation is organized as follows. In the next chapter, we present the overview of the clustered server design and its properties. We then present the design of our analytical multi-queue model of clustered servers. We also state the assumptions of the model and the parameters used for evaluation.

In Chapter 3, we describe the placement techniques for load balancing in clustered servers. We also present the results of the analytical model to derive optimal placement parameters and evaluate the architecture choices. We briefly describe a simulation model that is used to verify various analytical results.

Chapter 4 describes the placement schemes to guarantee data availability in clustered servers. We evaluate the performance of our parity placement schemes. We then describe our analytical cost model and use it to determine optimal parity group sizes. Finally, we present the implementation of a clustered server prototype and highlight the various design choices.

Chapter 5 motivates the need for large-scale distributed caches and presents the drawbacks of the current cache systems. We then describe our distributed cache architecture and present the algorithms used for hint cache propagations and the self-configuring meta-data hierarchy.

Chapter 6 evaluates the performance and scalability of our distributed cache architecture. We describe the implementation details of our prototype cache architecture and present the experimental results.

Chapter 7 discusses the state-of-the-art of cache replacement algorithms and describes the RBC algorithm. We then present results from extensive evaluation of the algorithm under various workloads and cache characteristics.

Finally, in Chapter 8 we summarize our work, and present an outline for future research.

Chapter 2

Scalable Server Design

Divide et impera—Divide and Conquer.

—Latin saying

In a wide-area information system, the function of the server is to store, process, and deliver information. Emerging applications require the storage and dissemination of massive amounts of information consisting of different types of data, that is accessed by millions of users across the Internet. To enable such applications requires the development of servers that support: (i) scalability—support the increasing number of users and requests, (ii) heterogeneity—meet the requirements of different data types for storage and delivery, and (iii) high availability—guarantee that data will be available to the users despite transient or partial failures.

Most current Web servers, built using a single machine, provide best-effort service and do not realize the requirements of emerging applications. First, monolithic servers cannot support the exponential growth of users and request rates. Second, best-effort service does not differentiate between the requirements of heterogeneous data types. A large fraction of the data stored at servers consists of non-textual data such as audio, video, animations, images. Audio and video data have fundamentally different storage and access characteristics compared to textual data; it requires an order of magnitude more space on average, and requires real-time delivery. Finally, the failure of a server node or disk results in a failure of service.

Our design of a scalable server, built using a multi-node cluster, has some fundamental properties that address the above requirements. The cluster architecture, however, has two main challenges: (i) load balancing across various components, and (ii) guaranteeing availability of data, while meeting the requirements of heterogeneous data types. Addressing these two challenges form the basis of our discussion in the following two chapters.

In this chapter, we first study the characteristics of the data and the accesses using statistics gathered from various existing web servers. Based on the observed characteristics, we enumerate the design principles of a scalable server in the following section. In Section 2.2 we describe the properties of a multi-node cluster; outline the challenges in building a multi-node cluster; and provide an overview of our server architecture. To design and evaluate algorithms for server design we build an analytical model of the cluster in Section 2.3, and show the use of the model for data-type specific quality-of-service requirements. Finally, section 2.4 summarizes the design. In the following chapter we use the analytical model to compare and evaluate various design choices and parameters of a clustered system.

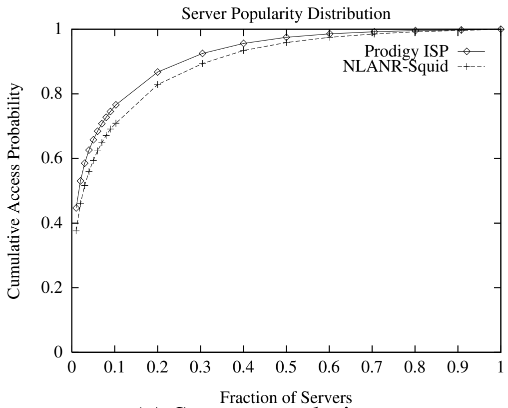
2.1 Data and Access Characteristics

We analyze the logs from various existing web servers and proxy servers to determine the characteristics of the data stored at these servers and the access patterns seen by them. The analysis shows the following access characteristics: (i) some servers are much more popular than others and the popularity varies with time, (ii) at each server, a large number of requests go to a small number of objects, and (iii) failures are common and visible to the users.

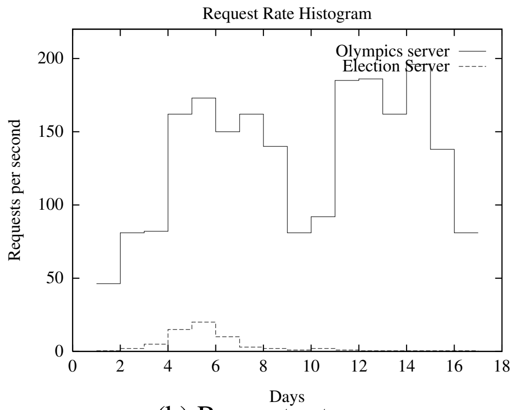
Figure 2.1(a) shows the popularity of different servers based on the access requests seen by the Prodigy ISP and the NLANR proxy servers. These show that the accesses are not evenly distributed across servers; 60% of the requests are for 5% of the servers. The demand at a particular servers also varies with time. Figure 2.1(b) shows that the request rate at the Olympics and the California election servers varying across multiple days; a maximum of 20 million accesses per day is observed for the Olympics server. Since capacity provisioning to meet transient overloads cannot be done optimally at design time, it is necessary that servers support incremental growth. Given the magnitude of users and requests, the servers need to scale to an increasingly large user population. This motivates two of the design principles for scalability: (i) do not have any central resource that will have a finite capacity, and (ii) support incremental growth to meet the demands of additional users without a complete reconfiguration.

Figure 2.2(a) illustrates the object access skew for the NCSA and NLANR servers; it shows that 75% of the requests are for 25% of the objects, i.e., a 75/25 skew. Higher skews of 80/20 and 90/10 are not uncommon. Since the request rate to the "hot" objects will be much higher that others, partitioning the set of objects among different nodes of a server will result in load imbalances. This motivates another design principle for scalability— a scalable server should not have hot-spots or load imbalances.

The analysis of the logs shows the following data characteristics: (i) different types of objects are stored and accessed at severs, and (ii) the mean size and access requirements depend on the data type. Figure 2.2(b) shows that the data requested at servers is a mix of data types consisting of text, images, audio, and video. Although audio/video account for 2 to 3% of the requests they have much higher sizes and have more stringent delivery requirements. Table 2.1 summarizes the size and delivery requirements of different data types. This motivates our second goal of supporting heterogeneous data types along with data-type specific policies.

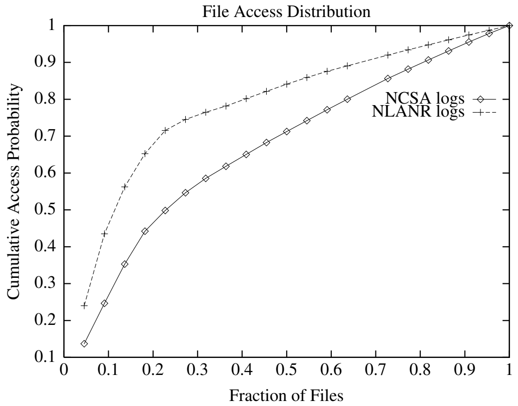


(a) Server popularity

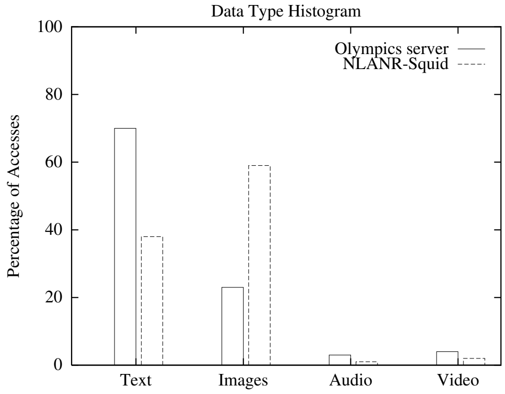


(b) Request rates

Figure 2.1: Skew of the server popularity and the request rate over a set of days. Figure (a) shows the fraction of servers and the probability of the accesses they received from data available at two large proxy caches. Figure (b) shows the request rates per second for each day for two heavily popular servers. The request rates correspond to the peak values in 1 hour intervals.



(a) Skew of object popularity



(b) Data type distribution

Figure 2.2: Object popularity skew and distribution of the data types requested. Figure (a) shows the popularity of a particular object (or file) at two large servers. Figure (b) shows the fractions of objects that belong to a particular data type.

Type	Mean Size	Delivery Requirements
Text	4 KB	Best-effort
Images	32 KB	Best-effort
Audio	500 KB	152 Kb/sec
Video	10 MB	1.5 Mbps (MPEG-1)

Table 2.1: Average size and delivery requirements of different data types

The analysis of the access logs at the Prodigy ISP and NLANR proxies shows that around 4% of the requests are some form of client or server errors (i.e., contain HTTP return codes of 400 and 500 series). A total of 1% of the requests are server errors corresponding to either server unavailability, connection time-outs, or server overload. Since failures are commonplace, availability is an important goal of the server design.

2.1.1 Design Principles for Wide-area Information Servers

The statistics on data and access characteristics motivates the three server design goals—scalability, heterogeneity, and data availability. We enumerate the design principles of a server to realize the above goals. Some of these principles are derived from the analysis of the data, while some are general principles for the scalability of large systems.

For scalability, the server design should be based on the following principles: (i) decentralize, (ii) reduce hot-spots, (iii) minimize global information, and (iv) support incremental growth.

Heterogeneity requires that the server differentiates on the service provided to different data types based on their inherent requirements. To achieve this the server should: (i) support data-type specific policies—the policies for storage and delivery of data should meet the requirement of different data types, and (ii) exploit data-type specific access characteristics—knowledge of access characteristics can be used for data-type specific optimizations. For example, the sequential nature of continuous media access can be used for placement and availability decisions.

Partial and transient failures of resources should be masked from the user to guarantee high availability. The design of a highly available server should: (i) reduce redundant resource overhead—the number of resources (disk, nodes, memory) used for redundant storage and access should be minimized, (ii) remove hot-spots after failure—the failure of a resource should not cause load imbalances among the functioning resources, and (iii) degrade gracefully—the occurrence of partial or transient failures should not result in complete loss of service.

2.2 Clustered Servers

Traditionally, a large scale server is built using a monolithic, symmetric multiprocessor (SMP) architecture. Such an architecture has a finite capacity, does not handle incremental growth, and is expensive. Our design of scalable servers, for wide-area information systems, consists of a *clustered server*—a group of independent nodes connected by a scalable interconnect. In a cluster, a group of nodes collectively service a request and handle failure. The inherent properties of clustered servers make them an ideal choice for wide-area information systems.

Before proceeding further on the discussion on clusters, it is necessary to comment on the current state-of-the-art and put this work in perspective. In the last few years, clusters have become the industry standard for scalability and are used in numerous commercially available Web servers [42]. Rather than motivating the need for clusters, in this dissertation we will primarily focus on the support for heterogeneous data types, continuous media in particular, and its effect on cluster design.

Observe that clusters differ from distributed servers that partition the system-wide object name-space among a group of nodes (e.g., AFS file server). In such servers the collection of objects is partitioned among a set of independent nodes, with each node managing the objects in its partition. This approach has two shortcomings that violate the design principles for scalability. First, when the accesses to objects are highly skewed, nodes storing the hot objects become the bottlenecks. Second, each node is the central controller for objects in its partition, thereby, becoming a single point of failure.

2.2.1 Properties of clustered servers

The four main properties of a clustered server that are beneficial toward its use as a wide-area information server are:

1. No central resource: By design, there is no central resource in the cluster. A group of resources collective provide a service. For example, the storage and control functions are distributed across a set of nodes.

2. Ease of incremental growth: Incremental growth of clusters typically consists of adding additional nodes, or additional disks/memory to a node. This is cheaper, simpler, and non-disruptive compared to upgrading or replacing a monolithic server with a higher capacity one.

3. Redundancy: Clusters are inherently redundant by having multiple independent resources. This redundancy can be exploited for masking failure.

4. No global information or shared state: Unlike large multi-processor machines, clusters have no global memory or shared state. Thus scalable applications map naturally to clusters.

Along with the above beneficial properties, there is also a economic advantage to building clusters. They can be built from commodity components with better cost/performance than special purpose high-end components. Clustered servers, however, pose a new set of challenges that have to addressed.

2.2.2 Challenges of Clustered Server Design

The design of clustered servers has the following challenges:

1. *Load balancing*: To guarantee that the load is balanced among the nodes, placement and replication techniques should be aware of the access characteristics of the workload and the requirements of heterogeneous data types.

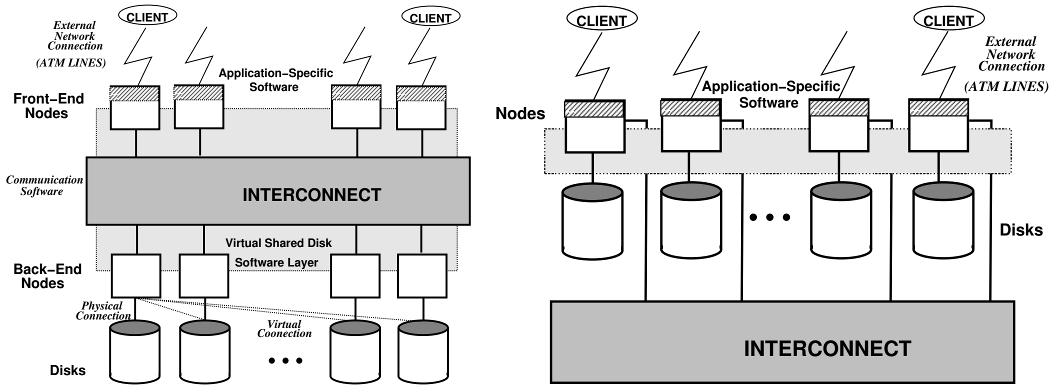
2. *Availability*: Clusters are susceptible to partial failures. The probability of partial failure increases with the number of resources in the system. Monolithic systems, on the other hand, have only total failures.

Another practical challenge for cluster design is that it should provide a single system abstraction to the applications and users. The existence of multiple nodes should be masked from the user. Software abstractions are required to make this possible without performance degradations.

In the following chapters, we focus on addressing the above challenges of a clustered server design. Some other management challenges of clusters are administrative complexity, failure detection, performance monitoring, etc., [42]. These, however, are not the focus of this dissertation.

2.2.3 Architecture Overview

The function of a wide-area information server is to service the user requests and store/retrieve the data. Based on their functionality, the nodes of a cluster can be divided into two logical categories: (i) front-end (delivery) nodes, and (ii) back-end (storage) nodes. Front-end nodes accept user requests and establish user connections. Back-end nodes have the attached local disks that store the data. Multiple back-end nodes may store the fractions or replicas of the data belonging to a particular object. To service a request, front-end nodes request for the blocks of data from the particular back-end nodes that store the data. Each front-end node functions independently. There is no global information maintained or shared among front-end nodes. The front-end nodes do not communicate or co-ordinate with each other. The communication path is only between the front-end and back-end nodes. This configuration addresses the design principle of eliminating shared state and global information exchange.



(a) Two-Tier (b) Flat

Figure 2.3: Server Architectures

The classification of nodes as front-end and back-end gives rise to two possible architecture configurations: (i) two-tier, and (ii) flat (see Figures 2.3(a) and 2.3(b), respectively). In a two-tier architecture, the logical front-end and back-end nodes are mapped to different physical nodes of the cluster and are distinct. The set of front-end (delivery) nodes are connected to back-end (storage) nodes via the interconnect.

In a flat architecture, on the other hand, each physical node can serve as a logical front-end or back-end. All nodes are identical, performing both storage and delivery functions. The node that manages the connection and initiates the request for a client is called the front-end node (for that request) while the one that stores and retrieves the data is the back-end node.

Given a cluster, we need to determine: which architecture configuration is suitable for a given system; what are the optimal parameters and policies for data layout and load balancing;

and what is the best policy for redundant data storage. To address these issues and analyze the clustered server design, we build an analytical model.

2.3 Modeling a Clustered Server

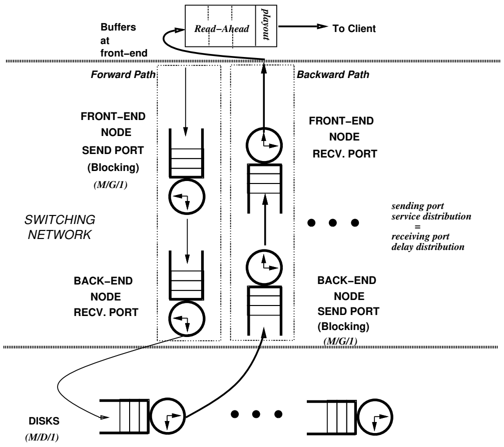


Figure 2.4: The Analytical Model Queues

The analytical model of a clustered server is used to: (i) evaluate the two architecture configurations (flat and two-tier), (ii) determine optimal placement parameters for load balancing with heterogeneous data types, and (iii) provide an insight in to the interactions of the interconnect and the disk on the placement parameters.

The cluster consists of three components— nodes, disks, and the interconnect. To service a client request, the front-end node servicing the request sends a request message to the back-end node (the request path is also called the forward path). The back-end node retrieves the data from the local disk. The data blocks are then sent from the back-end node to the front-end (the data path is also called the backward path). Given the above communication paths, the cluster can be modeled as three logical queues: i) the forward path queues, ii) the disk queues, and iii) the backward path queues (as shown in Figure 2.4(b)). Each disk and node is modeled as a single server queue.

To formalize the model, consider a cluster consisting of N nodes and D disks. Let S be the total number of clients in the system; each client has an average request rate λ. Let $1/\mu_{block}$

be the time required to transmit a data block across the interconnect, while $1/\mu_{request}$ be the the transmission time for a small request. The transmission time includes a connection set up time and a transfer time based on block size.

To instantiate the model, we first outline the various assumptions. We assume that the interconnect can be considered as a cross-point switch. Each node is connected to the switch through a sending port and a receiving port. To model the queuing delays in the data transfer across the switch, we assume that the message is queued at the source sending port and the destination receive port. After the sending port is acquired, it is *blocked* until the receiving port is also acquired, and the data transfer is completed in a FCFS order.

To evaluate the performance of the cluster, we use the delay distribution as a basis to derive other necessary statistics (mean, variance, tails of distributions, throughput etc.). In describing the components of the cluster model, we will instantiate the procedure for determining the delay distribution for that component. The overall delay distribution can be determined by a convolution of the individual delays.

2.3.1 The Disk Queue

The disk component consists of D identical disks attached to N identical nodes (on an average, each node is connected to D/N disks directly[1].)

Under the assumption that requests are independent, have random arrival times, and the number of requests is large, we can approximate the arrival process for read requests seen by the disks to be Poisson, with mean λS. Since the read request could be directed to any random disk, we approximate the arrival process at a particular disk to be Poisson, with a mean of $\lambda_d = \lambda S/D$. The disk service time consists of the seek latency for the disk head, the rotational latency, and the transfer time. For the disk service time we assume a general distribution that depends the seek time function for the disk and the disk block size distribution. Using the arrival and service distributions, each disk is modeled as an M/G/1 queue. This model can be simplified to an M/D/1 queue under the following assumptions: the disk block size is fixed, and the seek delays are relatively small compared to the total service time. Under these assumptions, the

[1]The model can be easily extended to a non-uniform configuration with additional book-keeping.

service time at the disk is deterministic and equals the mean service time $1/\mu_d$. From the results for an M/D/1 queue [56], we know the queue length distribution, p_n, which is the probability that there are n requests at a disk (Equations 2.1, 2.2, 2.3). From the queue length distribution, the delay distribution, $W_d(t)$ (the probability that a request has a disk delay equal to t) can be derived for different values of t.

$$
\begin{aligned}
p_0 &= (1-\rho)\ ;\ \rho = \lambda_d/\mu_d && (2.1)\\
p_1 &= (1-\rho)(e^{\rho}-1) && (2.2)\\
p_n &= (1-\rho)\Sigma_{j=1}^{n}(-1)^{n-j}e^{j\rho}[\frac{(j\rho)^{n-j}}{(n-j)!}+\frac{(j\rho)^{n-j-1}}{(n-j-1)!}]\ \ n \geq 2 && (2.3)
\end{aligned}
$$

2.3.2 The Forward Path

We had assumed earlier that the interconnection network can be modeled as a cross-point switch connecting N nodes. For the switch interconnect, the forward path consists of a small read request sent from a front-end node's sending port to a back-end node's receiving port.

The forward path is modeled as two queues—a blocking sending queue and a non-blocking receiving queue. The receiving queue has a deterministic service time. Since the service time at the sending queue depends on the queuing delays at the receiving end, we use the *delay distribution* of the receiving queue to determine the *service time distribution* of the sending queue.

Each back-end receiving port is modeled as an M/D/1 queue with one server. The service time at the receiving port is, $1/\mu_{request}$. The arrival rate at the back-end receiving queue is, $\lambda_r^{be} = \lambda S/(N_{be})$, where N_{be} is the number of back-end nodes. The queue length distribution p_i, and the mean delay for an M/D/1 queue is known (Equation 2.1) [56]. From the queue length distribution p_i, we get the delay distribution $W_r^{be}(t)$, at the back-end receiving queue for discrete values. The front-end sending port is modeled as an M/G/1 queue. The service time distribution at the front-end sending queue is set to equal, $W_r^{be}(t)$. The arrival rate λ_s^{fe}, at the front-end sending queue is $\lambda S/(N_{fe})$, where N_{fe} is the number of front-end nodes. From the service time distribution, the mean service time at the front-end sending queue is derived to be

$1/\mu_s^{fe} = 1/\mu_{request} + \frac{\rho_r^{be}}{2(1-\rho_r^{be})\mu_{request}}$, where ρ_r^{be} is the utilization of the back-end receiving port. Note that mean service time of the front-end sending port equals the mean delay at the back-end receiving queue. The delay distribution at the front-end sending port $W_s^{fe}(t)$, derived from the results for an M/G/1 queue [56] is given below.

$$W_s^{fe}(t) = (1 - \rho_s^{fe})\Sigma_{n=0}^{\infty}(\rho_s^{fe})^n[R^{(n)}(t)] \tag{2.4}$$

where,

$$R(t) = \mu_s^{fe}\int_0^t [1 - W_r^{be}(x)]dx \;\; \rho_s^{fe} = \lambda_s^{fe}/\mu_s^{fe} \tag{2.5}$$

and $R^{(n)}(t)$ is the n^{th} convolution of $R(t)$. Simplifying, we get

$$R(t) = \mu_s^{fe}\Sigma_{x=0}^{t}(1/\mu_r^{be})[1 - p_{\lfloor x\mu_r^{be}\rfloor}] \tag{2.6}$$

where p_i is the probability of the queue length being i at the back-end receiving port. Plugging the above simplification in Equation 2.4, gives the delay distribution of the forward path.

2.3.3 The Backward Path

The backward path consists of the transfer of a large data block from the back-end node's sending port to the requesting front-end node's receiving port. The backward path can be modeled in a similar fashion as the forward path, with the roles of front-end and back-end nodes reversed. Also the transfer time for a large block is much larger than that for the small request in the forward path. Each front-end receiving port is modeled as an M/D/1 queue and the back-end sending port is modeled as an M/G/1 queue. The delay distribution at the back-end sending port $W_s^{be}(t)$, is derived similar to that of the forward path.

2.3.4 Mapping Different Cluster Configurations

Having described the multi-queue model and derivations of the delay distribution, we instantiate the various parameters for the two cluster configurations: the two-tier architecture, and (ii) the flat architecture.

2.3.4.1 Two-tier Architecture

The delay distributions obtained from the above multi-queue model can be directly applied to the two-tier architecture. The logical front-end and back-end nodes can be mapped to physical front-end and back-end nodes. For simplicity, we assume in the analysis that the number of front-end and back-end nodes are equal (i.e., there are $N/2$ front-end nodes and back-end nodes). The forward and backward paths can be mapped to two separate links on the interconnect, and are distinct. For the two-tier architecture, the utilization at the back-end receiving port is $\rho_r^{be} = \lambda_r^{be}/\mu_{request}$, where λ_r^{be} is the arrival rate at the port. Assuming that the mean arrival rates at all ports are equal, and using the queuing assumptions of the ports, we can estimate the utilization at the front-end sending port, $\rho_s^{fe} = \rho_r^{be} + (\rho_r^{be})^2/2(1 - \rho_r^{be})$. The utilization at the front-end receiving port is $\rho_r^{fe} = \lambda_r^{fe}/\mu_{block}$. The corresponding utilization at the back-end sending port $\rho_s^{be} = \rho_r^{fe} + (\rho_r^{fe})^2/2(1 - \rho_r^{fe})$.

2.3.4.2 Flat Architecture

In the flat architecture each node functions as a logical front-end or back-end node. The links in the network are common for both forward and backward paths. Since there are two types of messages at every port, the arrival rate at any port (sending or receiving) is $\frac{2\lambda S}{N}(1-1/N)$, given that $1/N^{th}$ of the requests per stream are serviced locally and do not traverse the interconnect ports. The receiving port model is modified to be an M/G/1 queue with two service times, both equally likely. The mean service time $1/\mu_r$, equals the average time to send a message (short request and large block) across the interconnect. The sending port is modeled as an M/G/1 queue with the service time distribution equal to the delay distribution of the receiving port, as described in the clustered server model. In a flat architecture the utilization at the receiving port of all nodes $\rho_r = \lambda_r/\mu_r$. Using the interconnect model for a flat architecture and assuming that the arrival rates at all ports are identical, we can estimate the utilization at the sending ports $\rho_s = \rho_r + (\rho_r^2 + \lambda_r\sigma_r^2)/2(1 - \rho_r)$, where σ_r^2 is the variance in the service time at the receiving port.

2.3.5 Mapping Data-type Specific QoS Requirements

To use the model for evaluating various design choices or algorithms, the quality-of-service (QoS) requirements of each supported data type have to be defined. We classify the different types of data into two categories: continuous media data (audio/video, animations), and non-continuous media data (text, images, binaries). We first discuss the definition of quality-of-service, and then define it for the two classes.

2.3.5.1 Defining QoS

Different applications and data types have different semantics for correctness, failure, and performance. For example, if a binary file is transmitted the loss of any bit would be considered a failure, however, for an image transmission the loss of a few bits could be acceptable. An interactive application measures performance in terms of latency, while a batch application measures it in terms of throughput. The quality of service (QoS) specification does not have one fixed value but varies across systems, applications, and data types. Typically, applications define a range of QoS values that are bounded by the preferred value and the acceptable value. QoS can be define in terms of: (i) the type of service, and (ii) the metric used to measure the operational requirements. To further illustrate, the service type could be best-effort or guaranteed service; the metric used could be the measure of performance, correctness, or cost. The performance metrics fall into three categories: (i) timeliness (response time, latency, jitter, deadline misses), (ii) errors (failure rates, accuracy, staleness), and (iii) efficiency (throughput, overheads). In this dissertation, we assume that the QoS requirements of a data type are defined by two of the performance metrics—mean response time and deadline violations.

2.3.5.2 QoS for Continuous and Non-continuous Media Data

For non-continuous media data, we use the mean delay (or response time) as the measure of performance. This can be determined from the delay distribution at each queue of the multi-queue model. The total delay is the combined average of the delays at each queue. Using the disk queue model described in Section 2.3.1, the mean delay at the disk with a utilization of ρ_d, and mean service time $1/\mu_d$, is given by

$$\delta_{disk} = 1/\mu_d + \frac{\rho_d}{2\mu_d(1-\rho_d)}$$

Similarly, using the model described in Section 2.3.2, the mean delay in the forward path is given by

$$\begin{aligned}\delta_{net} &= 1/\mu_{request} + \frac{\rho_r^{be}}{2(1-\rho_r^{be})\mu_{request}} + \frac{\lambda_s^{fe}}{2(1-\rho_s^{fe})}[1/\mu_{request}^2 + \\ &\quad \frac{\rho_r^{be}}{3(1-\rho_r^{be})\mu_{request}^2} + \frac{(\rho_r^{be})^2}{4(1-\rho_r^{be})^2\mu_{request}^2}]\end{aligned}$$

The mean delay in the backward path can be derived from the above equation by replacing the mean service time to be the block service time $1/\mu_{block}$ and switching the back-end sending port with the front-end receiving ports.

For continuous media objects, the jitter in response time is of more consequence than the mean response times. The jitter is measured by the tail of the delay distribution. The tail of the delay distribution is the probability that the delay is greater than the acceptable delay (or deadline) for the delivery of the requested data.

The overall delay distribution is computed from the convolution of delay distributions at the forward path queues, disk queue, and the backward path queues. Assume that the deadline of a request is D_{ddl}, which is the maximum delay that the data retrieval can encounter and still meet its deadline. Let $W_d(t)$ be the delay distribution at the disk based on the model in Section 2.3.1. Let $W_n(t)$ be the network delay distribution, which is the aggregate delay of the forward and backward path queues (Sections 2.3.2 and 2.3.3). The total delay distribution W_t can be determined by the convolution of $W_d(t)$ and $W_n(t)$, i.e.,

$$\mathcal{W}(z) = \mathcal{W}_d(z)\mathcal{W}_n(z)$$

where $\mathcal{W}(z)$ is the Z transform of the distribution $W(t)$. The tail of the delay distribution $W(t)$, i.e., the probability that the delay is greater than D_{ddl}, is given by

$$P[t > D_{ddl}] = 1 - \int_{t=0}^{t<D_{ddl}} W(t)$$

2.3.6 Validation of the Model

To validate the analytical model and enable experimenting with configurations that cannot be modeled analytically, we built a simulation model of the system. In the simulation model all resources (nodes, ports, disks) are modeled as single server queues. The disk service time includes the seek time (square root function of the disk arm distance), the rotational latency, and the block transfer time. The service time for sending a request across the interconnect includes a fixed connection setup time and a transfer time proportional to the size of the block. Table 2.2 lists the default values of the parameters used by the models.

Description	Default
Front-End Processing Delay	8.192 msec.
Back-End Processing Delay	8.96 msec.
Block Size	256 KB
Front-End, Back-End Utilization	70-80%
Number of Processors (70% util.)	35
Processor Speed	100 MHz
Switch Bandwidth	20MB/sec
Switch setup overhead	2 msec.
Front-end s/w overhd. refer [17]	6.2 cyc./byte
Back-end s/w overhd.	3.5 cyc./byte
Mean Seek Time	10 msec.
Disk RPM	7200
Disk data transfer rate	6–12 MB/sec
Number of disks (80% util.)	52
Effective Disk Bandwidth	6.04 MB/sec
Scheduling Policy	FCFS

Table 2.2: Default parameters used. The parameters listed are for an RS/6000 workstation running AIX 3.2 and a commercially available fiber channel switch.

We compare the results of the analytical model with that of the simulation model to predict the overall probability of deadline violations of a continuous media request. Figure 2.5 shows the probability computed by the analytical method described in Section 2.3.5.2, with the one generated by the simulation. The results show a maximum difference of 2% in the predictions.

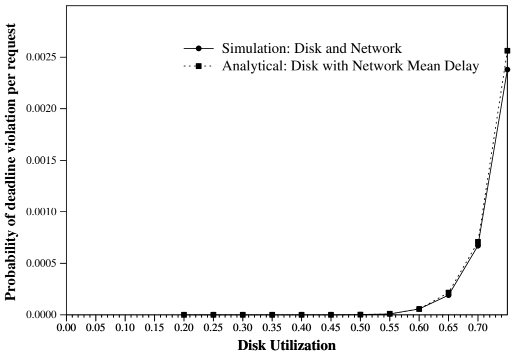


Figure 2.5: Comparison of the analytical and simulation models. The comparison is for a 500 client system consisting of 20 nodes. All clients are accessing continuous media objects, each with an access rate of 0.5MB/sec.

2.4 Summary and Conclusions

In this chapter, we motivated the design requirements of a scalable wide-area information server that supports heterogeneous data types and guarantees data availability. We outlined the inherent properties of a clustered server that make it suitable to meet the three design goals. However, a clustered server design poses a new set of challenges. The approaches we take to address those challenges are described in the following two chapters. To help evaluate various design choices, and gain insights into the behavior of the system, we built an analytical model of the cluster and described how the model could be used to determine the QoS parameters for different data types. In the following chapter, we use the model to determine the parameters and design choices for a clustered system.

Chapter 3

Load Balancing in Clustered Servers

Shared joy is double the joy, shared sorrow is half a sorrow.

—Swedish Proverb

One of the challenges in the design of a clustered server was to balance the load across nodes and disks. In this chapter we focus on load balancing in order to improve scalability while guaranteeing the performance requirements of heterogeneous data types. Load balancing consists of two problems: (i) balancing the load of data store/access by building data layout techniques that exploit the characteristics of the data and the access patterns, and (ii) balancing the load of data delivery by distributing the client connections across the nodes. First, we describe the data placement and architectural configuration for load balancing and better performance. We describe how to determine the optimal placement parameters for different data types based on the analytical model described in the previous chapter. Second, we describe and evaluate techniques for transparently balancing the load of client connections across the nodes.

3.1 Storage Architecture for Load Balancing

In this section, we focus on the architecture of the cluster storage (back-end nodes and disks), the placement techniques for load balancing, and determining the optimal parameters for data layout. Although the techniques and methodologies we describe for placement and load balancing hold for heterogeneous data types, the analysis and evaluation is restricted to continuous media data. This is done for two reasons. First, continuous media data is more demanding compared to other data types in terms of delivery requirements. Second, non continuous media has been studied extensively in related research efforts, hence we will use the well-established techniques [42]. Since non-continuous media data is much smaller in size, it can be replicated across disks with hot objects being cached in memory; in such cases load balancing across disks is not a critical problem.

3.1.1 Terminology

A *stream* is a sequence of continuous media data consisting of audio samples or video frames. The sequences of audio samples or video frame are grouped into blocks that are interleaved across disks; disk interleaving is also called *striping*. The unit of interleaving is called the *stripe unit*. The number of disks across which the data is striped is the degree of interleaving, or the *striping width*. The degree of interleaving is further classified as *narrow-striping*, if it encompasses only a fraction of the disks in the system, or *wide-striping*, if it is across all the disks. The stored data is retrieved in units of the stripe unit size. For continuous media data, the retrieval proceeds at a particular rate determined by the access rate given in frames/sec or samples/sec. There are two approaches used for managing retrieval. One is the *client-pull* mode, where the user requests for a unit of data (equal to the access unit of the data type) and associates a deadline for its delivery. The other approach is the *server-push* mode where the user requests the entire object and the server schedules the retrieval and delivery of the individual data blocks to the user.

3.1.2 Storage Architecture

A simple approach to building a multi-node cluster is to have a single node act as a front-end; this node accepts requests from external clients, maintains the connection, the client state, and creates a global schedule. All other nodes act as back-end nodes, i.e., the data is stored across their local disk arrays. In such an architecture, if the data is routed through the front-end the cluster is no longer scalable as the front-end becomes the bottleneck and a single point of failure. A solution to the front-end bottleneck problem is to allow the back-end nodes to directly communicate with the clients (e.g., Microsoft Tiger, SPIFFI, MARS file servers). This avoids the data routing through the front-end node. However, a single front-end node is a single point-of-failure. In our cluster design, we have multiple front-end nodes that connect to the multiple back-end nodes. Each front-end node accepts client connections, requests the data from the back-end nodes, and sends the data to the clients. The retrieval methodology for continuous media data is a push-pull model; push between the cluster and the clients, and a pull between the front-end and back-end nodes of the server. Since there is no communication among front-end nodes, there is no global state maintenance or global scheduling.

3.1.2.1 Comparison of Cluster Configurations

As described in the previous chapter, the multiple front-end and back-end nodes can be organized in two configurations: tow-tier and flat (see Figures 2.3(a) and 2.3(b), respectively). In a two-tier architecture, the logical front-end and back-end nodes are mapped to different physical nodes of the cluster and are distinct. In a flat architecture, on the other hand, each node provides the dual functions of storage and delivery.

The analytical and simulation models of the network can be used to provide an insight into the performance of the flat and two-tier architectures. It seems intuitive that the port utilization in a two-tier architecture in the backward path (large block) will be much greater than the forward path (small request). On the other hand, in a flat architecture, ports are uniformly utilized, as both types of messages are handled at each port. However, the flat architecture will have more variance in the switching delay due to two types of messages. We can compute the port utilization for the two architectures using the results in Section 2.3.4.

Table 3.1 shows the analytically determined port utilization for the different architectures, assuming a 256KB size block and a switch bandwidth of 20MB/sec. It can be seen that the port utilization of the two-tier architecture are highly skewed, with the back-end sending port saturating for a 40 node system, while the corresponding flat architecture ports are less than 50% utilized. Thus, the two-tier architecture has poor performance when the switch bandwidth

Number of Nodes	Two-Tier				Flat	
	Front-End		Back-End		Sending	Receiving
	Sending	Receiving	Sending	Receiving		
56	7.2%	42%	57.2%	7%	29.1%	24.5%
50	8.3	48	70	8	34.3	28
44	9.4	54	85	9	40	31.5
40	10.5	60	100.5	10	46	35
36	11.6	66	130	11	52.6	38.5
32	12.8	72	164	12	59.8	42

Table 3.1: Port Utilization. The table shows results from the analytical model for a cluster serving 500 streams, each with an access rate of 0.5 MB/sec.)

is low and the node utilization is high.

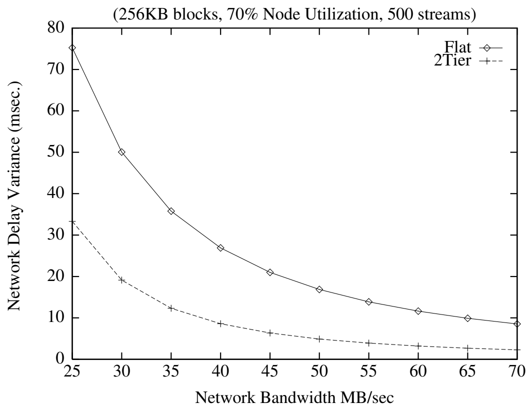
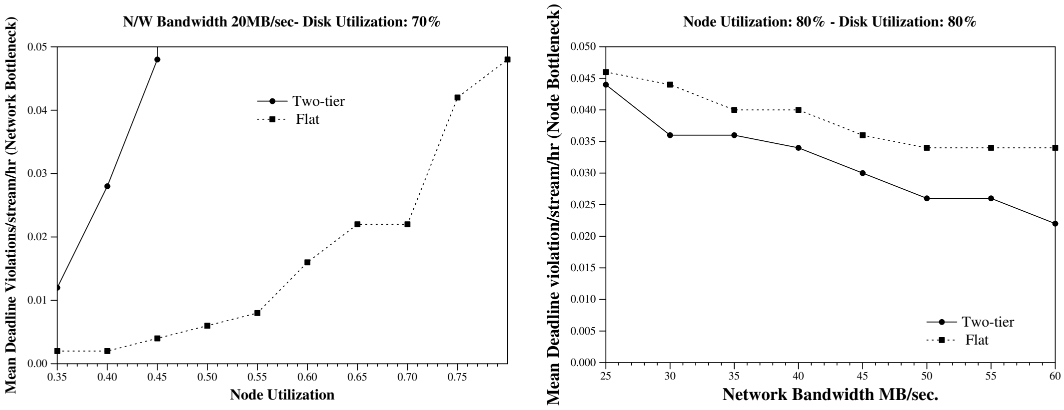


Figure 3.1: Comparison of interconnect delay variance. The analytical model is used to determine the variance in the network delay for different values of the interconnect bandwidth for the flat and two-tier architectures.

The analytical model is used to compute the variance in the round-trip switching delay for the two-tier and flat architectures. The variance and other higher moments required for the computation can be derived from the Laplace transform of the delay distribution of the sending port queue [56] [109]. The flat architecture has a much larger delay variance in the forward path when compared with that of the two-tier architecture as there are only small messages in the two-tier forward path. In the backward path, however, the flat architecture could have slightly smaller variance than the two-tier because of the much lower port utilization. The total variance is higher for the flat architecture. Figure 3.1 shows the total switch delay variance for the two architectures with varying switch bandwidth. The variance for the two-tier architecture is less than half of that for the flat architecture.



(a) Switching Network Bottleneck (b) Node Bottleneck

Figure 3.2: Flat Vs. Two-Tier Architecture. The figures show results obtained from a simulation model for the two architectures. The y-axis is the probability of deadline violation for each stream per hour for a cluster serving 500 streams. In figure (a) the node node utilization are varied while fixing the interconnect bandwidth to 20MB/sec and the average utilization of each disk to be 80%. In figure (b) the interconnect bandwidth is varied with the average utilization of the nodes and disks being 80%.

The overall performance of the two architectures depends on the network and node parameters. We use the simulation model to verify the analytical predictions by observing the probability of deadline violations of a stream request. The flat architecture[1] has larger delay variance but lower port utilization, compared to the two-tier architecture. The choice of architecture depends on which among the nodes and network, is the bottleneck. If the switching network is the bottleneck, then the flat architecture has better performance, because the sending ports in the two-tier saturate, even when the number of nodes in the system is large (low node utilization). Figure 3.2(a) shows the loss rates increasing rapidly due to saturating ports, when the node utilization is just 50%. When the switching network is not a bottleneck (high bandwidth), the port utilization is small and the delay variance dominates. The two-tier has smaller variance, leading to lower probability of deadline violation (as shown in Figure 3.2(b)). When the network is a bottleneck, the flat architecture is preferred due to the uniform port utiliza-

[1] In a flat architecture the data residing on the local disks of a node can be accessed without traversing the switch. Thus 1/Nth of the data per stream can be accessed locally.

tion. When the network is not a bottleneck, the two-tier architecture is desirable for practical considerations as it is simple to configure and manage and has better performance. If the interconnect is a broadcast network (ethernet or token-ring), the two-tier and flat architecture will have similar networking performance.

3.1.3 Placement Parameters

The data layout policy stripes the data across disks attached to different back-end nodes. The striping of data across nodes is called *network striping*. In conventional servers, on the other hand, data is striped across all the disks attached to a single node. There are three advantages to striping in general, and network striping in particular. It inherently balances the load since the request to an object increases the load on all disks and nodes equitably. This is, however, under the assumption that each object is accessed in its entirety and striped across all nodes. Secondly, striping exploits parallelism to effectively to utilize the bandwidth of the disks and node. Finally, as we discuss in the following chapter, striping enables redundant storage and improves availability in case of disk or node failure.

The data layout across the storage nodes and disks, consists of the block placement scheme, and determining the stripe unit size and striping width.

3.1.3.1 Placement Algorithms

Typically, successive blocks of an object are placed on disks using a *round-robin* or sequential placement algorithm. As per the round-robin placement, successive blocks of a video stream are placed on consecutive disks (see Figure 3.3)). One drawback of the round-robin placement is that the addition of a disk or node requires complete re-striping. Another disadvantage is that the sequential access of the disks and nodes by the streams causes a *convoy effect*. If there is a congestion on a disk, it travels to the next disk for the next request and so on. Although this limitation can be addressed by controlling the startup of each stream, such policies incur higher initiation latencies and require global scheduling. We propose the *balanced random* placement algorithm for a scalable clustered server. With balanced random placement, successive blocks are placed on disks using a random permutation, with no two consecutive blocks on the same disk. The first block of a stream could be allocated on any disk with equal probability. An advantage of the balanced random placement policy is that it adapts easily to incremental growth, which is one of the design principles for scalability. If disks are added frequently to the system, then it is much easier to redistribute the data with the addition of a single disk; only $1/D^{th}$ of the data per disk needs copying, where D is number of disks. Random placement also eliminates convoy effects. However, the basic problem with random placement is that: (i) even with all the requests playing out at a constant rate, it is statistically possible that a large number of requests may simultaneously access the same disk or node, and (ii) it requires more meta-data. To overcome the performance problems, the stripe unit size is chosen to improve performance, and replication, as described in the next chapter, is used to effectively mask transient overload.

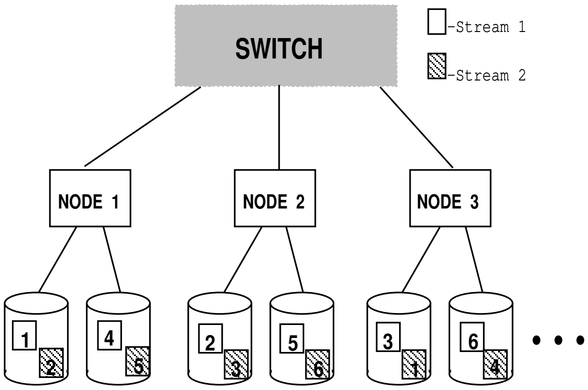


Figure 3.3: Sequential Block Placement

3.1.3.2 Optimal Stripe-Unit Size

The analytical model is used to determine the optimal stripe-unit size (or block size) that minimizes the probability of a request missing its deadline. The optimal block size results from the opposing effects of the disk throughput and the disk delay variance. As the block size increases, the throughput of the disk increases due to amortized seek overhead of disk reads. However, with an increase in block size, the variance in the delay increases. To understand these effects, we compute the disk throughput and the variance for different blocks sizes. In the following derivation we assume that the blocks are network striped and are placed using the balanced random placement.

The utilization ρ_d of the disk represents the load on the disk and is inversely proportional to its effective throughput. The value of ρ_d is the ratio of the arrival rate λ_d and the disk service rate μ_d. The service rate, which is the inverse of the disk service time, includes the seek, rotational latency, and transfer time. The value of μ_d for a block of size b, is given by,

$$1/(s + bT + r_{lat})$$

where s and r_{lat} are the seek and rotational latency overhead and T is transfer time for a unit of data. To understand the effect of increasing block size on the disk utilization, let us assume that the block size b, is increased k-fold. The corresponding arrival rate $\hat{\lambda}_d = \lambda_d/k$, and the corresponding service rate $\hat{\mu}_d$, is given by,

$$1/(s + bkT + r_{lat})$$

The corresponding disk utilization $\hat{\rho}_d$ as a function of ρ is

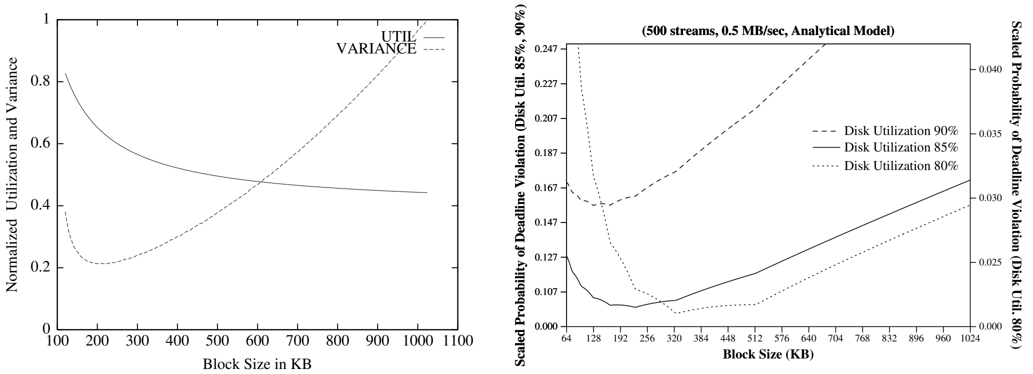
$$\hat{\rho}_d = \rho_d - (1 - 1/k)\lambda(s + r_{lat})$$

Since $\hat{\rho}_d < \rho_d$, it shows that a larger block size decreases the load on disk. On the other hand, when the block size is large, the variance in the response times at the disk and network increases. Even a small queue length results in long delays. The response time variance at the disk is given by,

$$\sigma^2 = \frac{1}{3\mu_d^2(1/\rho_d - 1)} + \frac{1}{4\mu_d^2(1/\rho_d - 1)^2}$$

From the above equation, it can be seen that as the block size decreases, the service rate increases; since the variance is proportional to $1/\mu_d^2$, the variance decreases(Figure 3.4(a)). To reduce the variance a smaller block size is preferred.

The tradeoff between disk throughput and delay variance results in an optimal block size that minimizes the tail of the response time distribution at the disk. The computation of delay distribution and the service time distribution at the disk are modified to be a function of



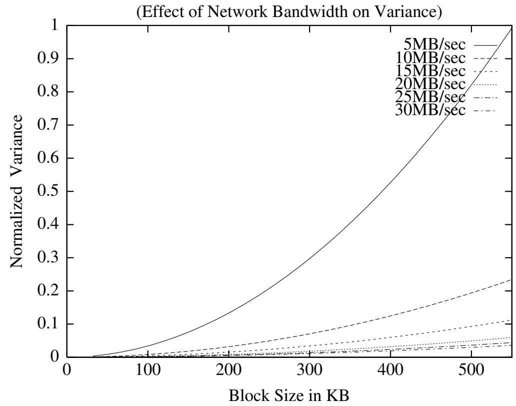
(a) Disk (b) Optimal Block Size Determination

Figure 3.4: Effect of Throughput and Variance on Stripe Unit Size. The analytical model is used to determine the disk delay variance for different disk block sizes. In figure (a) the disk utilization and delay variance is normalized to show the tradeoff on increasing block sizes. In figure (b) the combined effect is presented in terms of the probability of deadline violation of continuous media streams for cluster serving 500 streams, each with an access rate of 0.5 MB/sec.

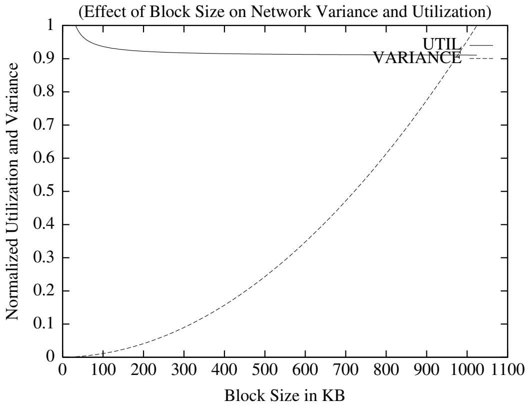
the block size. The block size that minimizes the tail of the delay distribution is the optimal block size.

We can compute the optimal block size from the delay distribution at the disk. Figure 3.4(b) shows the probability of deadline violation using the analytical model for disk utilization of 80, 85 and 90%. For a 90% disk utilization, a block size in the range of 64-290KB is within 5% of the optimal. When the disk utilization is decreased to 85% the optimal block size range shifts to 130-330KB. A further decrease in disk utilization to 80%, results in the block size range of 250-590KB to be within 5% of the optimal. As the disk utilization is decreased, the effect of disk delay variance on the probability of deadline violation, decreases, shifting the optimal range to larger block sizes. For high disk utilization, there is a larger variance, thus the optimal stripe unit size shifts to smaller values. We observe that the minima shifts to smaller block sizes as the disk utilization is increased. The optimal block size selected is the one that lies in the intersection of the optimal size range for the different operational disk utilization.

Effect of the Interconnect As with the disk, a similar analysis can be done to compute the role of the interconnect in determining the optimal stripe unit size. Note that the network will



(a) Effect of Bandwidth on Network Variance



Effect of throughput and network variance

Figure 3.5: Effect of Stripe-unit Size on the Interconnect Variance and Throughput. Figure (a) shows the normalized variance for different interconnect bandwidths as the stripe-unit size is varied. Figure (b) shows the normalized utilization and variance of a switched interconnect for varying stripe-unit sizes and a bandwidth of 20 MB/sec.

have limits on the maximum transmission unit (e.g., IP MTU is 64KB), thus the block retrieved from disk may be transmitted as smaller size packets through the network. The utilization ρ_n, of the network sending port in a two-tier architecture for an arrival rate of λ_n, and the network transmission rate μ_n, can be derived from the queuing model and is given by,

$$\rho_n = \frac{\lambda_n}{\mu_n} + \frac{\lambda_n/\mu_n)^2}{2(1-\lambda_n/\mu_n)} \tag{3.1}$$

Let μ_n, for a packet size p, with the connection set-up overhead c, and the network bandwidth B, be given by,

$$1/(c + p/B)$$

This is the inverse of the time to transmit the packet through the network. The connection setup overhead can be minimized by worm-hole routing in some networks. To compute the effect of increasing the packet size on the port utilization, let us assume that it is increased k-fold. As the size increases, the number of messages in the network decrease, hence the header overhead decreases. The corresponding arrival rate $\hat{\lambda}_n$ decreases by more than k-fold and is $\frac{1}{k/\lambda_n+(k-1)O_h}$, where O_h is the packet header overhead. The new service rate $\hat{\mu}_n$, for the larger packet size, decreases less than k-fold, and is given by $1/(c + pk/B)$. The corresponding utilization of the port $\hat{\rho_n}$, hence decreases. From this we conclude that a larger packet size is favorable for lowering the port utilization. However, the variance in the delay distribution at the network is also dependent on the packet size and is given by,

$$\sigma^2 = E[delay^2] - E[delay]^2 + \lambda_n E[delay^3]/3(1-\rho_n) + (\lambda_n)^2 (E[delay^2])^2/4(1-\rho_n)^2. \quad (3.2)$$

$$E[delay^k] = E[service^k] + \frac{\lambda_n}{(1-\lambda_n/\mu_n)} \sum_{j=1}^{k} \binom{k}{j} E[delay^{k-j}] E[service^{j+1}]/(j+1) \quad (3.3)$$

where $E[service^k] = 1/\mu_n^k$, and $E[delay^k]$ is the k^{th} moment of the delay distribution at the port. As the packet size increases, the variance increases and a smaller packet size if preferred (Figure 3.5(b)). The tradeoff between throughput and delay variance results in an optimal packet size that minimizes the tail of the response time distribution at the network. The computation of delay distribution at the network is modified to be a function of the packet size. This is used to determine the packet size that minimizes the tail of the delay distribution. If the network connection overhead is small, then the gain in port throughput due to increasing the packet size is not that substantial. Thus a smaller packet size is preferred due to smaller variance. However, if the network bandwidth is increased, the effect of variance decreases and a larger packet size is favored. The effect of network bandwidth on the variance is shown in Figure 3.5(a). When the effect of the network is combined with that of the disk to compute a common access unit size (i.e. disk block and network packet), if the network bandwidth is high, the access unit selected is governed by the disk effect. If the network bandwidth is small, the access unit size is smaller than one determined by disk effect as the network effect favors a smaller packet size.

Effect of Buffer Size Due to the sequential nature of continuous media requests, the blocks required later for delivery can be pre-fetched and stored in the read-ahead buffers at the front-end node. Read-ahead buffering is a simple approach to increase the deadline bound of a request, leading to a decrease in the loss probability, while adding some extra memory cost. The delay bound of a block request is the time interval between when the request is sent and the time at which the block is transmitted. Specifically, if there are *k* read-ahead buffers associated with a request, the delay bound of a request increases by a factor of k, from the original bound without buffering. The probability of deadline violations, with *k* read-ahead buffers, can be derived using the analytical model.

If the data transmission unit is equal to the block on disk, the optimal block size on disk will depend on the amount of read-ahead buffering. If a fixed size buffer space[2] is available per stream, there is a tradeoff between the disk performance favoring a large block size, and the lower probability of deadline violations with multiple read-ahead buffers, favoring smaller sized blocks. A smaller block with a single read-ahead will have the same response time deadline as the block of double the size and no read-ahead. The smaller block will have smaller variance and smaller service time and hence the probability of exceeding the deadline is smaller than the larger block.

For a given amount of memory, a single read-ahead buffer with larger block sizes, performs worse than multiple buffers with smaller blocks [110]. However, when the available memory is large, the gain by proportionately increasing the number of read-ahead buffers is small (the gain by increasing the read-ahead from 1 to 2 is much more than that from 3 to 4, as the tail of the delay distribution becomes flat after some large delay), consequently the block size gets governed more by disk performance than read-ahead memory size.

Size of Memory per req (KB)	4 Buffers		3 Buffers		2 Buffers	
	Block size (KB)	P[t > ddl]	Block size (KB)	P[t > ddl]	Block size (KB)	P[t > ddl]
256	64	0.000014	85	0.000237	128	0.012678
512	128	0.0	170	0.000027	256	0.007161
768	192	0.0	256	0.000011	384	0.006105
1024	256	0.0	341	0.0	512	0.005899
2048	512	0.0	682	0.0	1024	0.006873

Table 3.2: Block size and Read-ahead Buffering. In this table we compare the partitioning of a given amount of memory among different number of read-ahead buffer. The size of a read-ahead buffer corresponds to the size of the stripe-unit on disk. For each of the memory sizes we compare the analytically determined probability of deadline violations for different values of read-ahead.

Table 3.2 derived from the analytical model shows that for a given amount of memory, a single read-ahead buffer with larger block sizes, performs worse than a smaller block with multiple buffers. However, when the available memory is large (e.g., $\geq$ 1MB in Table 3.2),

[2] playout + read-ahead buffers

the gain in increasing the number of read-ahead buffers decreases, consequently the block size should be determined only by the disk performance.

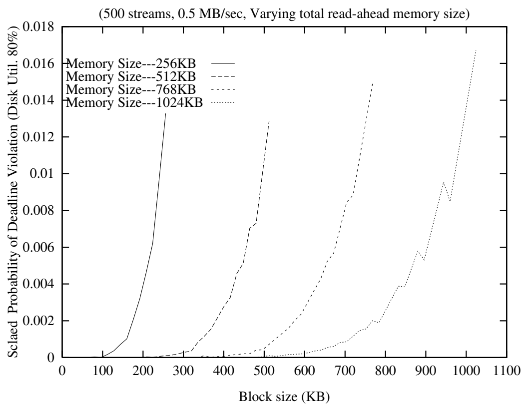


Figure 3.6: Effect of Read-ahead Memory on Optimal Block Size. In this we vary the stripe-unit size and compare the probability of deadline violations for different amount of read-ahead buffer memory available. The ratio of the memory available and the stripe-unit size gives the number of read-aheads. The comparison is for a cluster serving 500 streams, each with an access rate of 0.5 MB/sec and an average disk utilization of 80%

Figure 3.6 shows the effect of read-ahead memory size on the optimal block size. In Figure 3.4(b) in the previous section, the optimal block size was determined based on the disk performance and variance and at 80% disk utilization it was in the range of 300-350 KB. If the additional factor of read-ahead buffering is included, the optimal block size range shifts to less than 90KB for a 0.25MB read-ahead memory size increasing to less than 500KB for 1MB of memory.

3.2 Load Balancing Client Connections

For load-balancing the client connections across the nodes in a cluster, the architecture of the server should be transparent to the clients. The client view should consist of a single front-end node that represents the server. This is particularly important for connection oriented services (e.g., TCP/IP). A simple solution, used widely for Web servers, is to have a single logical name assigned to the multiple front-end nodes [75]. The mapping from a logical name to multiple physical nodes is done at the name resolution stage. A round-robin domain name server (RR-

DNS) maps the logical name to different IP addresses. This results in load balancing across the front-end nodes. However, the load balancing achieved is coarse due to the address caching in the client, the bursty nature of client accesses, and the time-to-live (TTL) values imposed by nameservers on the name-to-address mappings.

Our approach for achieving load balancing is based on transparent routing and is illustrated in Figure 3.7. One or more nodes of the cluster serve as *TCP router(s)*, forwarding client requests to the different front-end nodes in the cluster in a round-robin (or any defined method) order. The name and IP address of the router is public, while the addresses of the other nodes in the cluster is private. (If there is more than one router node, a single name is used and RR-DNS is used to map the name to the multiple routers.) This has been referred to as the *encapsulated cluster* in [98]. The client sends requests to the router node which in turn forwards all packets belonging to a particular TCP connection to one of the front-end nodes. The router can use different algorithms, based on load, to select which node to route to, or use a simple round-robin scheme. In the prototype (see below), the router resides just above the IP layer as a kernel extension. The server nodes directly send the request back to the client without using the router. However, the source address on the packets sent back to the client are that of the router node. This makes the entire routing action transparent to the clients. Note that, the response packets are large compared to the request packets, especially for images and video clips, and these bypass the router. Thus, the overhead added by the router is small.

One advantage of the router scheme over the DNS based solutions is that fine grained load balancing can be achieved by using application specific schemes, and there is no problem of client or nameserver caching. Figure 3.8(a) shows the effect of varying the number of requests per client session on the dynamic load imbalance at the server, for different TTL values at the nameservers. Each request to the server is assumed to be of the same size so that the load imbalance observed is not due to varying request size. The dynamic load imbalance is measured as a ratio of the maximum number (99% percentile) of requests serviced by a node, in any sampling interval, to the mean computed over the total simulation time period. (Note that, both the RR-DNS and router schemes lead to good average load balance across the nodes over an extended period of time. We focus on the dynamic load balance which can lead to

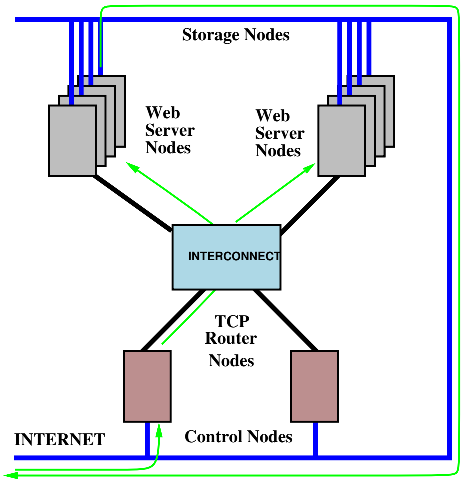
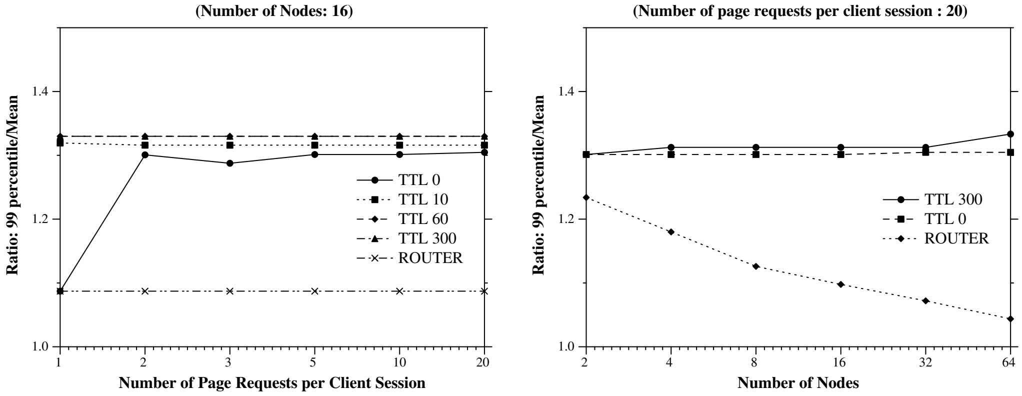


Figure 3.7: Load balancing using a transparent TCP Router

very different peak loads on nodes in the cluster using the two schemes.) The inter-arrival times between page requests per client session is derived from an exponential distribution with a mean of 15 seconds. For a TTL value of 0, as the mean number of page requests per client session increases from 1 to 20, the effect of client address caching becomes more pronounced, resulting in a load imbalance increase from 8% to about 30%. Note, each page requests consist of a burst of requests (mean of 10) to the server. With an increase in the TTL value from 0 to 300 seconds, without client address caching (number of page request per client is one), the load imbalance increases from 8% to about 33%, sharply at first and then gradually. Increasing TTL at the nameservers, or increasing the duration of a client session, adversely effects the load balance at the server nodes. The router approach results in a much lower load imbalance of about 8% caused by the bursty nature of client requests and random arrivals.

The load imbalance is also a function of the number of nodes in the Web server. As the number of nodes increases the load imbalance using the router decreases. Figure 3.8(b) shows the effect of varying the number of nodes on the load imbalance using the RR-DNS and router approaches. When the number of nodes is small (e.g., 2) the load imbalance due to the bursty nature of the requests and client random arrivals is high for both the router (22%) and the RR-DNS (30%). As the number of nodes increases, the effect diminishes for the router as it forwards the requests to different nodes. For the RR-DNS approach, however, the load imbalance is



(a) Effect of Client Sessions Effect of System Size variance

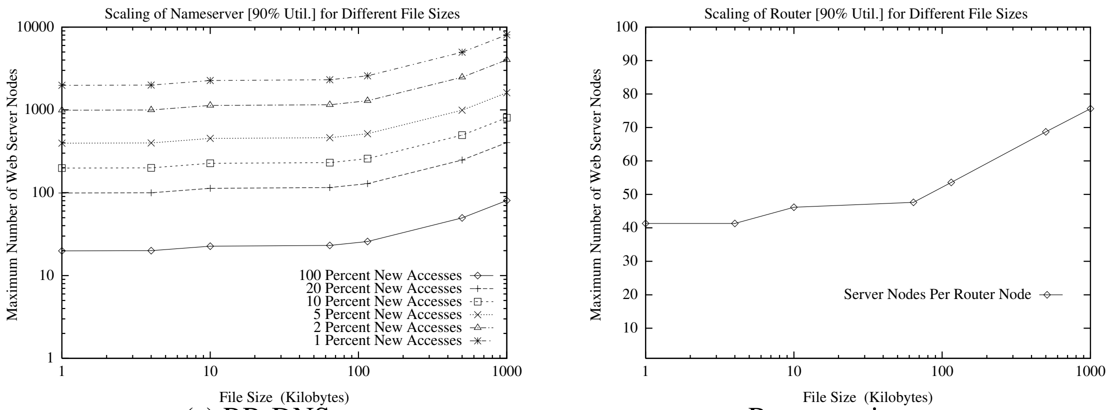
Figure 3.8: Load imbalance of RR-DNS and Router. Figure (a) compares the load imbalance as the ratio of the 99th percentile to the mean for a 16 node cluster using the Router and the round-robin DNS schemes with different values of the DNS time-to-live. Figure (b) compares the load imbalance as the number of nodes vary when each client makes bursty requests of 20 objects per session.

due to client and nameserver address caching, and it gradually increases (up to 31%) as the nodes are increased. Thus, for a large scale system a router based approach is recommended as the load imbalance is significantly smaller for the router than the DNS approach. For smaller systems both approaches are viable. Also in the simulation study each client randomly selects a nameserver, whereas in practice some nameservers may be much more utilized than others, causing larger imbalances using the RR-DNS approach.

3.2.1 Scalability of RR-DNS and Router

Figure 3.9(a) shows the scalability achievable via the RR-DNS solutions, for different workloads (file sizes), and for different ratios of new accesses to total accesses. The RR-DNS is accessed only upon requests from new clients, since the address of the Web server node is cached at the client after the first access. We varied the percentage of new accesses from 100% (every access is from a new client) to 1% (a client makes a hundred requests to the Web site before terminating). Since every document typically consists of several constituent objects, the latter number reflects a few pages accessed per client.

Since each node can handle fewer requests per second when the documents are large, we see that a single RR-DNS can handle the request to a larger number of server nodes when the average file size is larger. In absolute terms, the RR-DNS scales to a large number of nodes (100-200) even at smaller file sizes, assuming that 10-20% of the RR-DNS accesses are from new clients.



(a) RR-DNS Router variance

Figure 3.9: Scalability of RR-DNS and Router. Figure (a) shows the number of nodes required to handle the Webstone benchmark workload for different file sizes when using the round-robin DNS. Each curve is for a given percentage of new requests, since the DNS is used only for routing new requests and the old requests go directly to resolved address that was cached. Figure (b) compares the same for one router node running at 90% utilization.

For the router, we measured the CPU utilization during the execution of the Webstone benchmark [112] with four server nodes configured. For a given server node at 90-95% utilization, the CPU utilization of the router node is about 2.2% when the file-size is small (1 KByte) and 1.2% for large files (1 MByte). Thus, based on the average file size of the Web server workload, a single router node (operating at 90% utilization) can support between 40 and 75 nodes. Figure 3.9(b), shows the scaling of a single router node for documents with different file sizes. If larger configurations are desired, multiple router nodes can be used, with load balancing across the routers being achieved via RR-DNS. As previously discussed, the router solution has better load balancing properties compared to the RR-DNS solution.

3.3 Related Work

In the recent past, several research projects have investigated techniques for designing small-scale servers [6, 18, 20, 65, 103, 118]. Almost all of these efforts, however, have focussed on relatively single node servers, in which all the disks are assumed to be connected one node. In single-node servers with disk arrays, the processing node becomes the bottleneck, as it cannot sustain the bandwidth requirements of a large number of clients. Such servers cannot scale to support the increasingly large number of requests or request rates required by new and emerging applications. The research issues involved in utilizing clustered architectures for supporting heterogeneous data types, including continuous media data, have not received much attention.

In a concurrent work, Freedman and DeWitt [43] have studied the SPIFFI multi-node server. In their work, all clients have their clocks nearly synchronized and are assumed to possess information about the server architecture, data placement, and scheduling. Whereas their model of a cluster is appropriate for a set of terminals connected to a group of processors, it is unsuited for geographically separated clients that do not possess information about server internals.

A large amount of research has focused on special-purpose servers for continuous media data [18, 48, 49, 99, 118, 119, 128]. The Fellini system is a single server machine with multi disks with buffer management techniques for continuous media along with text data types [90]. Berson et. al., also evaluate multi-disk architectures using staggered striping schemes to effectively utilize disk bandwidth and also handle disk failure through parity based schemes [6]. There are also a number of ongoing commercial and research projects that focus on building large-scale servers. The Berkeley initiative for large scale servers is based on using commercial continuous media file systems (e.g. Microsoft Tiger) and adding database type search and retrieval functionality [102]. The MITRA project uses multi-disk workstations with an ATM switch interconnect and a centralized scheduler with staggered striping (they do not handle failure) [47]. The MARS project uses an ATM port interconnect to connect storage devices and hardware RAID for fault tolerance and centralized scheduling [12]. The work by Liu et. al., uses the Onyx symmetric multiprocessor with application-level striping and non-centralized scheduling [64, 65]. The SPIFFI project used a network of workstations with client pull architecture in a LAN environment [43]. Some of the commercial projects include Microsoft Tiger that uses off the shelf processors connected by an ATM switch with centralized scheduler and block level replication [11]. The Oracle large-scale server uses DEC alpha servers and DEC Memory channel interconnects; but the system details are not available [77]. These projects have taken the tightly-coupled, centralized control approach to building large-scale servers. This approach violates the scalability principles of eliminating central control along with independent nodes.

3.4 Summary and Conclusions

In this chapter, we investigated the data layout and load balancing schemes for our clustered server design, that consists of multiple front-end and back-end nodes. We examined the performance of different cluster configurations, and concluded that flat architectures are practical for low bandwidth switches as the port utilization are better balanced. When the switch bandwidths are high and the processing nodes become the bottleneck, the two-tier architecture which has lower delay variance, becomes viable. We discussed our balanced random placement algorithm and network striping technique that allows both incremental growth and inherent load balancing. Further, we determined the optimal stripe unit size for continuous media data. The optimal striping unit size balances the tradeoff between disk throughput, favoring a large size, and delay variance, favoring a smaller stripe-unit size. We showed the effect of the interconnect bandwidth on the stripe unit size, which favors a smaller size when the bandwidth is low. The available buffer memory also plays a role in the stripe-unit size determination. With a small amount of memory available a smaller size block that allows multiple read-aheads is preferred. To balance the number of client connections across the front-ends nodes, we designed and evaluated a router scheme for transparent load balance and compared it to the earlier DNS scheme. The router scheme achieves 4 times better performance in load balancing compared to the RR-DNS approach.

In the following chapter, we focus on the second challenge of clustered server design that of guaranteeing data availability and masking disk and node failures from the clients.

Chapter 4

Guaranteeing Data Availability

For every action there is an equal and opposite malfunction.

— Murphy's Third Law of Motion

One of the challenges of a clustered server architecture is its susceptibility to partial failures. The larger the number of nodes and disks in the cluster, the higher the probability of a partial failure. Typical mean time to failure (MTTF) values for a disk is 300,000 hours, while for a node (including software failures) it is around 18,000 hours [22]. A 10 node, 100 disk system, will experience a disk failure every 4 months and a node failure every 2.5 months. Partial failures can be masked by storing redundant data, and migrating the load of the failed resource among the surviving disks and nodes.

In this chapter, we describe and evaluate placement schemes for redundant data that can mask both node and/or disk failures in a cluster. The requirements of the placement schemes for redundant data are to: (i) balance the load during normal and failure mode operations, (ii) minimize the computation and storage overheads, (iii) support optimizations and performance guarantees of heterogeneous data types, and (iv) scale to a large number of nodes and disks. Earlier work on redundant placement techniques addresses only the first two of these requirements. They do not support heterogeneous data types with real-time performance requirements, and they do not scale to a large number of disks and nodes. We design redundant data placement schemes: (i) the sequential parity placement with multiple strides, and (ii) the balanced random parity placement, that balance the load during normal operations and after failure; and scale with increasing number of disks and nodes. We discuss optimizations for continuous media objects with real-time requirements, that exploit the sequential nature of the accesses. We design an analytical cost performance model to determine the optimal degree of redundancy to reduce the storage overhead (i.e., parity group size), while maintaining the necessary performance requirements after failure. The model is also used to determine the region of applicability of different schemes. For masking the loss of a client connection on a node failure, we rely on known distributed recovery schemes and briefly mention the techniques used [42].

In our discussions, we assume that a cluster consists of front-end nodes that establish client connections and deliver the data, and back-end nodes that store and retrieve the data. Disks are attached to the back-end nodes. Data is striped across multiple back-end nodes. For supporting heterogeneous data types, we focus on continuous media data that has more stringent performance requirements.

The remainder of the chapter is organized as follows. In Section 4.1, we discuss different approaches to provide high availability. Section 4.2 discusses our redundant data placement schemes. Section 4.3 compares the real-time performance of the various availability schemes. Section 4.4 describes an analytical model to determine the cost performance tradeoffs between different placement schemes and to determine the optimal parity group size. Section 4.5 briefly outlines handling failed client connections and Section 4.6 describes the related work. In Section 4.7 we describe our prototype that creates a transparent single system view of the cluster storage nodes. Finally, Section 4.8 summarizes our contributions.

4.1 Overview of Techniques for Redundancy

To guarantee high availability, the failure of a disk or node must be masked from the client. For continuous media data, failure masking techniques should also ensure that the real-time requirements of the requests are not violated. For example, in a 50 disk system, with 256KB blocks, a stream access rate of 0.5 MB/sec will result in a *glitch* every 25 seconds. A node failure makes the disks attached to the failed node become inaccessible thereby causing more disruption in service compared to a disk failure. To ensure continuous service, various hardware and software techniques can be used to mask single disk or node failure. Most of these techniques store redundant data or provide redundant paths, which can be used after failure to retrieve the lost or inaccessible information on the failed resource. The redundant data and the original are placed in mutually exclusive locations; i.e., across different disks for masking disk failure, and across different nodes to mask node failure.

The hardware and software techniques for high data availability include: (i) mirroring, (ii) twin-tailing, and (iii) RAID. Before evaluating these techniques, we briefly outline the approaches in the next section. In a later section, we compare these schemes based on the following factors: (i) space overhead, (ii) bandwidth overhead, (iii) real-time performance, (iv) tolerance of multiple failures, (v) scalability, and (vi) overall cost/performance.

4.1.1 Mirroring

With software mirroring, the data blocks on each disk are replicated on a different disk (attached to a different node). In the event of a failure, the mirror-disk is read for the corresponding data block. Mirroring of data can be done at the *disk-level* or at the *block-level*. At the disk level, each disk has a corresponding mirror-disk on a different node. When a failure (disk or node) occurs, the front-end nodes re-route the request to the back-end node with the mirror-disk. The entire load of the failed disk is handled by the mirror disk. The 100% load overhead on a mirror disk adversely effects on the performance after failure. With block-level mirroring, each block of data on any disk has a replica stored on another disk attached to a different node. Block-level mirroring achieves load balancing; all disks equally share the load of a failed disk. During normal operation, all disks operate at the desired load. Both approaches, however, double the space requirements. Mirroring has a 100% space overhead.

4.1.2 Twin-tailing / Multi-tailing

Twin-tailing is a hardware approach to handle node failure without storing redundant data. Node failure makes the local disks inaccessible, but not inoperable. Instead of equating node failure to multiple concurrent disk failures, the disks can be made accessible from a different node. All disks attached to a node can be twin-tailed by a disk adapter to a *buddy* node. In the event of a node failure, the front-end nodes re-route all the requests for the failed node to the corresponding buddy node. The buddy node can access the disks on the failed node to service the requests.

If each node has only one buddy node, a failure will cause a 100% load increase at the buddy. The overload problem can be alleviated by having a different buddy node for each disk; this approach leads to non-trivial problems in configuration and systems management, and does not scale. It may be possible to multi-tail disks beyond two-way, but it is typically infeasible for more than 4 or 8-way. For a large-scale server, 4-way or 8-way tailing will still have a load balancing problem; hence, we do not consider multi-tailing as an approach for redundancy in this dissertation.

4.1.3 RAID

Parity based schemes are used to provide redundancy without the complexity of twin-tailing or the space overhead of mirroring. The parity schemes are based on the Redundant Array of Independent Disks (RAID) architecture. The parity, in RAID architectures, is computed by an exclusive-OR operation over a group of data. Since a parity block is maintained for a group of data blocks the disk space overhead of RAID is less than that of mirroring. The RAID architecture is designed for hardware controlled recovery and fault-tolerance. The RAID disk controller manages the placement and retrieval, masking it from the operating system. Different hardware RAID schemes are classified into levels, based on the granularity of the interleaving and the striping pattern across the disks. In RAID level 1 each disk has a corresponding mirror disk; this is the same as disk-level mirroring. RAID level 2, uses a memory style error correcting code and is not commonly used. RAID level 3 and 4 have a single parity disk that stores the parity information for all the remaining disks. While RAID level 3 has a bit wise granularity, RAID level 4 has block interleaved parity. Both these schemes do not achieve load balancing. The parity disks remains idle during normal read operations and the remaining disks have a 100% load increase after failure. RAID level 5, with a distributed block-interleaved parity, addresses this drawback by distributing the parity blocks across all the disks (e.g., a left symmetric parity placement). By distributing parity across disks, RAID level 5, balances the load during normal operations. However, during failure there is a 100% load increase on all the disks. This is due to the fact that a parity group size in RAID level 5 is equal to the number of disks in the system (a *parity group* consists of a set of data blocks and the corresponding parity block). Since all disks are involved in each parity group, after failure, all the disks are accessed for recomputing the block on the failed disk. Several improvements to RAID level 5 address this limitation. The *declustered parity* scheme, reduces the parity group size from being the number of disks in the system (D), to be much smaller than D. By storing more redundant information, the declustered approach trades disk space for performance. The parity blocks are all distributed across different disks. For a D disk system and a parity group size G (G $<$ D), the load on the surviving disks increases by G-1/D-1. The space overhead, on the other hand is higher; it is 1/(D-1) for RAID5, while it is 1/(G-1) for RAID5 with declustered parity. The challenge in declustered parity is to distribute the parity groups in such a manner that the load remains balanced after failure. To achieve this distribution, the total number of parity groups that include a given pair of disks should be constant across all pairs of disks (i.e., for any i, j, the disk numbered i, j, will appear in exactly n parity groups, for a given value of n). A layout strategy derived from a balanced incomplete block design, satisfies this property. Note that generating block design is a non trivial problem and there may not be not known design for all values of of D and G. A complete block design is exponential in computational complexity, requiring the analysis of $\binom{D}{G}$ distinct mappings (e.g., for D = 100 and G =5, a full block design table will have 75 million tuples) [62]. Since the block design table are know for small value of D, this approach cannot scale to a large number of disks.

The different RAID schemes discussed above, implement RAID in hardware. Hardware RAID schemes [93] cannot mask node failure. Moreover, with network striping across nodes, hardware RAID is not feasible, as the parity groups cannot be across nodes. Hardware RAID schemes could use twin-tailing to use disks across nodes (this, however, does not scale to more than 2 to 4 nodes). The RAID techniques can be extended to enable software management of placement, parity computations and reconstruction. Software based RAID schemes can support network striping and parity groups across nodes. Software RAID can be used to mask both node and disk failures. Since the parity placement schemes used in the different levels of hardware RAID are not scalable to a large number of disks and nodes, they cannot be directly applied to software RAID schemes for a clustered server.

4.2 Parity Placement Schemes for Clustered Servers

In our cluster design, we assume that the objects are striped across disks and nodes using network striping. For masking disk and node failure, data blocks and parity blocks are striped across disks and nodes. A *parity group* consists of a set of data blocks and a parity block, all belonging to the same object. We describe three parity placement techniques and compare them based on (i) space overhead, (ii) scalability, (iii) load balancing during normal operations and failure, (iv) real-time performance for continuous media objects. In the remaining discussion in this chapter, when discussing performance or load balancing, the focus is primarily on reads since in a wide area information system the reads dominate the writes. By default, in the following discussion node failure refers to back-end node failure which only affects the availability of data and not the client connections.

4.2.1 Sequential Parity Placement

A straightforward approach for parity placement in software RAID is the *sequential parity placement*. In the sequential parity placement scheme, the parity groups of an object are placed in a sequential fashion, (based on the disk numbers) across the disks. Let us assume that a disk numbered, i, is attached to a node numbered, $i \ mod \ N$, where N is the number of nodes. For example, a parity group of size 5, results in a *DDDDPDDDDPDD..* placement, where D is a data block and P is a parity block. In this way no two blocks of an object belonging to the same parity group will be placed on the same *node* or *disk*, given that the number of nodes is greater than the parity group size. Different objects have their first data block on randomly selected disks, leading to a uniform distribution of parity information across all disks. Figure 4.1 shows the placement of blocks of two objects with a parity group size of 5. All the parity blocks for object 1 are on disks 4 and 9, and for object 2 are on disks 3 and 8. The block numbering notation $a.b$, for an object refers to the b^{th} block within the a^{th} parity group of the object.

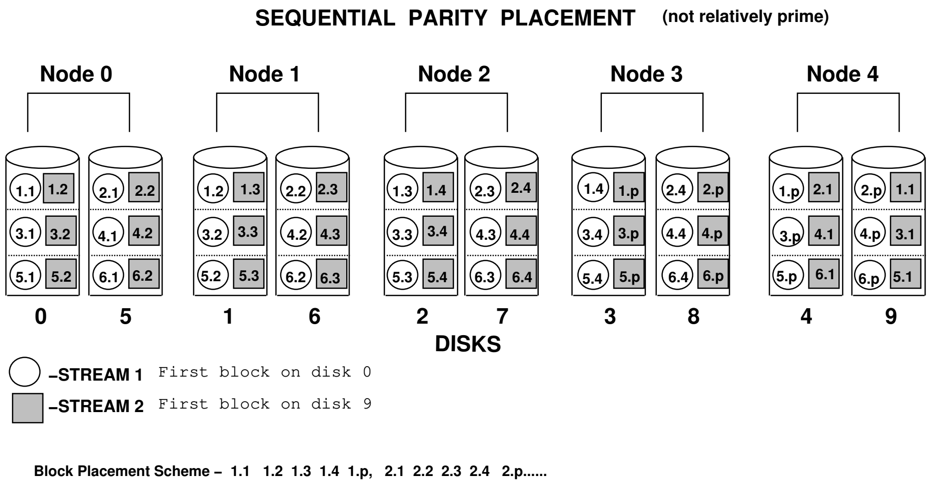


Figure 4.1: Sequential Parity Placement. The parity group size is 5 for a cluster of 10 disks and 5 nodes.

In sequential placement, when the parity group size is a factor of the number of disks (or nodes) in the cluster, the parity blocks belonging to an object will be placed on $\frac{D}{GCD(P,D)}$ disks (where GCD is the greatest common divisor). Since the parity blocks are not evenly distributed among all disks, there is a load imbalance during normal operations. To distribute the parity blocks among all the disks, the parity group size to be *relatively prime* to the number of disks (and/or nodes), i.e., $GCD(D, P) = 1$. In a sequential parity placement scheme that satisfies the primality constraint, the parity blocks are spread across all disks (and nodes) and do not wrap around to the same disk (or node), thereby achieving load balancing during normal operations. Figure 4.2 shows an example of the organization of a sequential parity scheme, satisfying the primality constraint, that can tolerate single node and disk failures. The example shows a 5 node cluster with a parity group size of 3, and 2 disks per node. The parity blocks for both the objects are spread across all the disks and nodes in the system.

In the sequential parity placement scheme, all the disks in the system are uniformly utilized during normal operation. However, during failure the load of the failed disk is not shared equally among all the disks in the system. The blocks of an object belonging to the same parity group are placed on contiguous disks with the parity block being at the end of the parity

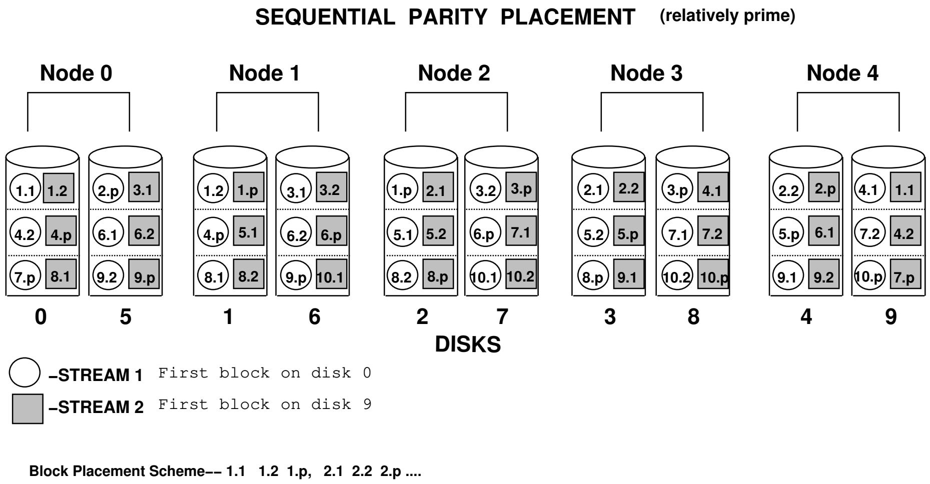


Figure 4.2: Sequential Parity Placement with primality constraint. The parity group size is 3 for a cluster of 10 disks and 5 nodes.

group. Thus, only (P-1) disks following the failed disk contain the corresponding parity blocks for the data blocks on the failed disk. Thus only $P-1$ disks share the load of a failed disk. After failure, $D-P$ disks will operate at the normal load while $P-1$ of the disks have their load increase by a factor of $\frac{P}{P-1}$. (For example, in Figure 4.2 if disk 3 fails, then the parity blocks, 2.p and 5.p for data blocks, 2.1 and 5.2 of object 1, have to be obtained from disks 4 and 5.) If the disks are driven to their highest utilization during normal operation, the overload during failure could lead to congestion and overload.

The sequential parity placement with parity group size P, has a space overhead of $P/P-1$. For a given parity group size all parity placement schemes have the same space overhead. Since only P disks are involved in the reconstruction of a failed disk block, in the best case, the sequential parity placement scheme can tolerate D/P concurrent disk failures. Sequential parity placement can recover from multiple disk failure as long as the failures are in different parity groups. However, the probability of two independent disks failing concurrently is low. For example, if the MTTF for a single disk is 300,000 hours, the MTTF for any disk in a 1000 disk system is 300 hours, while for 2 disks failing together, it is about 20 years.

For continuous media objects, the real-time performance of sequential parity placement is a function of the parity group size and overhead for XOR computation. Under normal operation, the data blocks are read at regular intervals based on the playout rate of the stream, and the parity blocks are not accessed. In the event of a failure, for each stream that has the next requested block on the failed disk, the associated delivery node issues simultaneous requests for the remaining blocks in that parity group, to different disks/nodes, and performs an XOR operation to recompute the lost data. All responses to the parallel reads for the blocks in the parity group and the subsequent XOR computation must be completed by the real-time deadline of the initial request. Although this occurs only when blocks on a failed disk/node are accessed, it increases the load on the nodes and disks and can cause the number of missed deadlines to increase significantly.

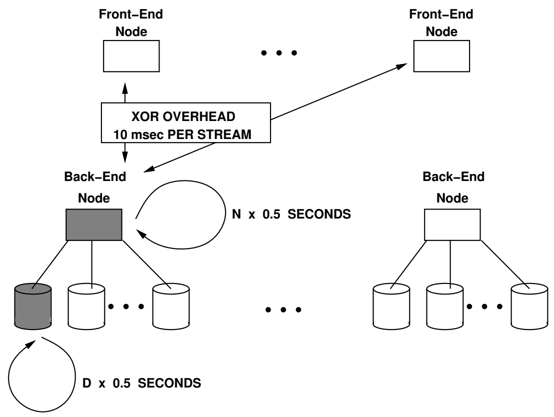


Figure 4.3: Frequency of Accessing Failed Node/Disk. For the blocks of a continuous media object read at rate of 2 blocks/sec, each disk is accessed every D/2 seconds.

A small amount of *parity-group buffering* (equal to P-2 buffers for each stream accessing the failed disk/node concurrently) can alleviate the problem of parallel reads. Since continuous media objects are read sequentially, some of the parity-group blocks required later for an XOR computation can be stored in the parity buffer after they are fetched. On an average, P/2 blocks belonging to the same parity group would have been read before encountering a request for a block on the failed disk/node. Only the remaining blocks in the parity group need to be fetched in parallel. If the size of each block is B bytes, and there are N nodes and D disks serving S streams, an average of S/D streams are accessing a failed disk at any time. In the event of node failure, S/N streams are accessing disk blocks on a failed node. Therefore, the maximum size of a parity-group buffer would be (P-2) $\times$ B $\times$ S / D, for supporting single disk failure, and (P-2) $\times$ B $\times$ S / N, to support single node failures. On an average only half of the maximum buffers required will be used if a running XOR is maintained.

The additional processor load due to the XOR computation is not prohibitively high. Typical measurements show that an XOR of 256 Kbyte blocks, for a parity group size of 5 (i.e, a 3 way XOR), takes about 10-15 msecs. of CPU time. For a cluster supporting 500 stream, with 10 back-end nodes, 10 front-end nodes (50 streams per front-end processor) and 50 disks, and a playout rate of 0.5 MB/sec (two 256 Kbyte requests every second, which is a typical MPEG-2 rate), each stream requests a block from a given disk every 25 seconds. The XOR processing overhead at a front-end node for 50 streams, (10 msecs. per stream) is about 0.5 second (50 $\times$ 10 msec.) every 25 seconds, or about 2% of the CPU utilization. For back-end node failure, the XOR overhead is incurred more often, since each back-end node is visited by a stream every 5 seconds (0.5 $\times$ 10). The front-end processor overhead is about 10% in this case. This is depicted in Figure 4.3. In case of a system with 1000 streams (100 disks and 20 back-end nodes), the overheads are reduced by a factor of 2 to about 1% for disk failure and about 5% for node failure.

4.2.2 Balanced Random Parity Placement

To alleviate the problem of load imbalance during failure in the sequential placement scheme, the parity blocks associated with the data blocks on a disk have to be distributed uniformly across all the other disks.

Our simple scheme called *balanced random parity placement* (Figure 4.4), positions the parity blocks of an object on a random permutation of selected disks. The constraint on placement is that blocks belonging to the same parity group are not placed on the same disk or on any disk on the same node. This handles both disk and node failures. The random parity placement scales to a large number of disks and nodes since the scheme achieves statistical load balancing without exponentially complex incomplete block design for placement.

In the balanced random parity placement scheme, all the disks in the system are uniformly utilized during normal operation. After a disk failure, the load of the failed disk is

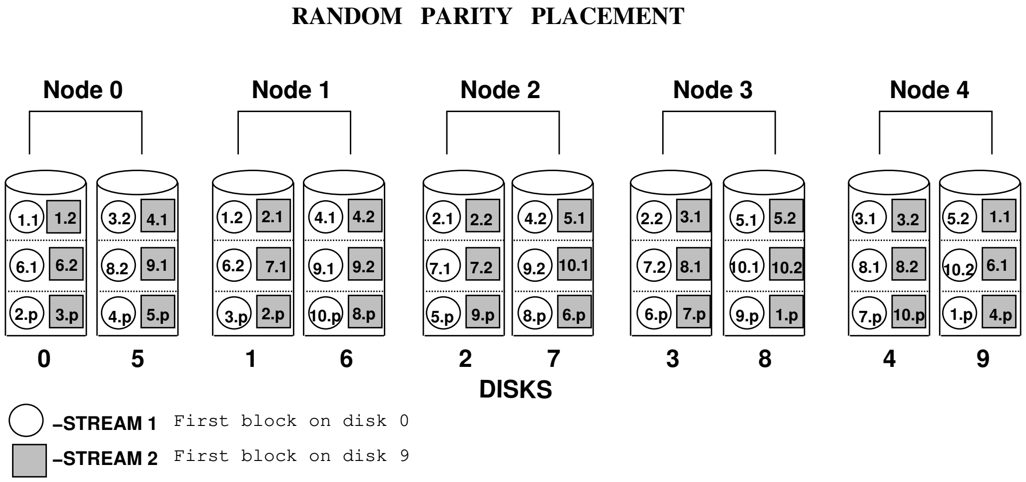


Figure 4.4: Balanced Random Parity Placement. The parity group size is 3, for a cluster of 10 disks and 5 nodes.

shared equally among all the disks/nodes in the system. After failure, $D-1$ disks will operate have the load increase by a factor of $\frac{P-1}{D-1}$. The space overhead of $\frac{P}{P-1}$ is equal to that of sequential parity placement. Since random parity placement distributes parity blocks across all disks and nodes, it cannot tolerate multiple failures. The random parity placement scheme requires more meta data compared to a sequential parity placement scheme. To reduce the metadata we design a scheme that randomize the placement using a simple placement function.

4.2.3 Multi-stride Parity Placement

To decrease the amount of meta-data required for random parity placement, we design sequential parity placement with multiple strides. The stride is the number of disks between two consecutive blocks of an object. If block i is placed on disk D_j and block $i+1$ is placed on disk D_k, the stride $\sigma = |j-k|$. In the sequential parity placement described earlier, the stride used between blocks is 1. In the stride placement scheme the blocks and parity are placed sequentially on disks $D_j, D_{j+\sigma}, D_{j+2\sigma}, D_{j+3\sigma}, ...$ and so on. If the stride is made relatively prime to the number of disks (and/or nodes), i.e., $GCD(\sigma, D) = 1$, the parity blocks associated with the data blocks on a disk will be distributed across multiple disks (Figure 4.5). While in sequential placement the distribution of parity blocks was across $P-1$ disks, with stride placement they

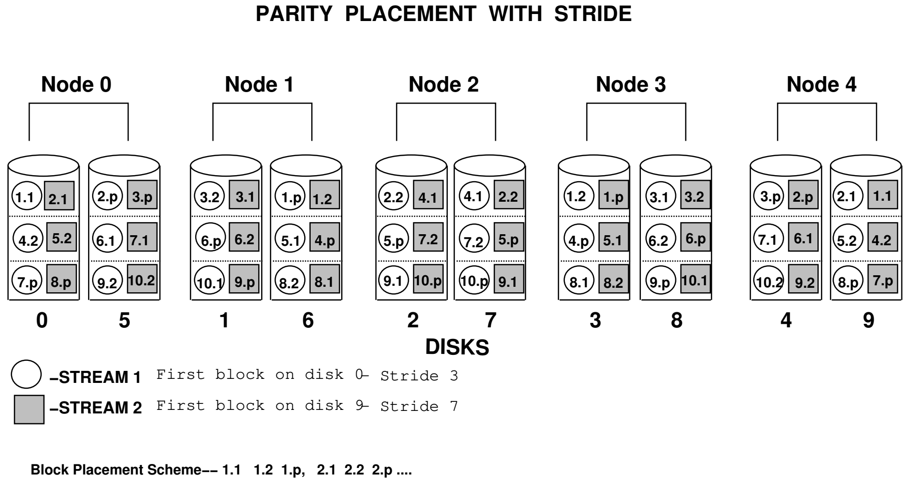


Figure 4.5: Multi-stride Parity Placement. The parity group size is 3, for a cluster of 10 disks and 5 nodes. The stride varies across objects and for a particular object.

are distributed across $(\frac{\sigma(P-1)}{GCD(\sigma,D)})\ mod\ D$ disks. If different objects have different strides for placement, the distribution becomes more widespread. If the stride is changed in every round (a new round begins when the first block of a parity group of an object is placed on the same disk as the first block of the object), the placement approaches random parity placement and is equivalent to it in performance during failure.

4.2.4 Buffering and Pre-fetching for Continuous Media Data

With the parity placement techniques discussed above, on failure the load on the surviving disks increases by a factor $\frac{P-1}{D-1}$. For continuous media data that consists of sequential reads, the load on the surviving disks reduces to $\frac{1}{D-1}$. Since all blocks of an object are read in sequential order and the entire parity group consists of blocks of same object, there is no extra load of reading the entire parity group. Only the access of the parity block corresponding to the data block on the failed disk contributes to the extra load. This requires extra buffering to store the blocks of a parity group to recompute the failed disk block when required. A single buffer will suffice if a running XOR of the blocks is computed and stored.

Multiple parallel read of the parity group could cause deadline violations of continuous media requests. An optimization for relaxing the real-time constraints is to pre-fetch the blocks in the parity group [7, 120]. When a request is made for a block on the failed disk, the parity group for the next block on the failed disk could be pre-fetched and stored in the *parity-group* buffer. In this way no parallel request need be made to read the parity group of the next unavailable block, since they have already been pre-fetched. The pre-fetch requests could be given a large deadline (based on the number of disks and the playout rate) so that they do not interfere with the requests of other streams that have more stringent real-time requirements. However, with pre-fetching, the buffer requirements increase to the number of blocks in the parity group for every stream in the system, i.e., the parity-group buffer space required for S streams is (P-1) $\times$ B $\times$ S.

4.2.5 Comparison of Redundancy Schemes

Factor	Mirroring	Software RAID	
	Block Level	Sequential Parity	Balanced Random Parity
Space	2 D_{space}	$D_{space}P/(P-1)$	$D_{space}P/(P-1)$
Disk BW (n)	$D_{bw}\frac{(D-1)}{D}$	$D_{bw}(P-1)/P$	$D_{bw}(D-1)/D$
Disk BW (f)	D_{bw}	$(P-1)$ disks at D_{bw}	D_{bw}
Multiple failure	1	D/P	1
Node BW (n)	$N_{bw}(1-\frac{1}{N_dN})$	$(1-P_o)N_{bw}(1-\frac{1}{N_d(P-1)})$	$(1-P_o)N_{bw}(1-\frac{1}{N_dN}))$
Node BW (f)	N_{bw}	$(P-1)$ nodes at N_{bw}	N_{bw}
Extra buffering	-	$(P-1)$ S/D	$(P-1)$ S/D
Extra metadata	Per block	-	Per block
KEY: D_{space}: Normal disk space requirement; D_{bw}: Max. disk bandwidth; N: Number of nodes; P: Parity group size; D: Number of disks; S: Number of streams; N_{bw} : Operational node bandwidth N_d: Maximum number of disks per node; P_o : Overhead of parity computation; (n): Normal Operation; (f): With Failure *The operational bandwidths and utilization of nodes and disks varies with application requirements.*			

Table 4.1: Comparisons of Different Redundancy Schemes

Table 4.1 summarizes the space and bandwidth overhead of the disks and nodes for mirroring and software RAID with sequential and random parity placement. The mirroring, sequential parity placement, and balanced random parity placement schemes are compared by the amount of space overhead, node and disk bandwidth overhead for failure and normal mode operations, the ability to handle multiple failures, and the book-keeping required.

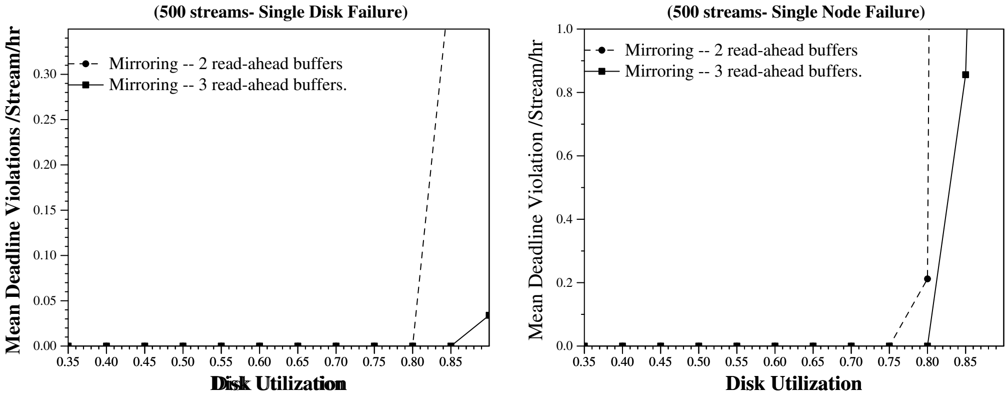
4.3 Evaluation of Real-time Performance

For continuous media objects we compare and evaluate the real-time performance of the schemes for availability. We use the real-time performance of block-level mirroring as a best case and compare it with the performance of the sequential and balanced random parity placement techniques for software RAID. The real-time performance is measured by the average number of deadline misses per unit time for the continuous media streams. In order to study the deadline miss rate under disk and node failure, we simulated a system with 500 concurrent streams, for a default parity group size of 5. We simulated the failure of a disk after a short duration of normal operation (about 120 seconds) and compare the performance of the different parity placement schemes.

4.3.1 Mirroring

Since mirroring has the lowest load increase on failure and no XOR overhead, the real-time performance of block-level mirroring during failure is better than the various software RAID schemes, and can be used as a basis for comparison. Disk-level mirroring translates to 50% of the disks remaining idle during normal operation and is not considered. Using block-level mirroring, all the disks operate at the desired bandwidth utilization during normal operation, and equally share the load of a failed disk during failure.

Figure 4.6(a) shows the number of deadline violations per hour with mirroring during single disk failure, for a 500 stream system and multiple read-ahead buffer sizes. With an increase in read-ahead buffer size from 2 to 3, the maximum operational disk utilization (without deadline violations) increases from about 80% to about 85%, and the real-time performance degrades gradually. Under failure free operation, using a similar configuration, the maximum disk utilization is around 90%. The rate of deadline violations estimated for a single node failure is shown in Figure 4.6(b). The feasible operating range of disk bandwidth utilization is significantly lower for single node failure than that for a single disk failure. This is because in an N node, D disk system, node failure causes the load on the operational disks to increase by about 1/N, while with disk failure the load increases by 1/D. With node failure the maximum



(a) Single Disk Failure (b) Single Node Failure

Figure 4.6: Mean Loss Rate with Block Level Mirroring. The y-axis shows the number of deadline violations per hour for streams with an access rate of 2 blocks/sec with a 256KB block size. The x-axis show the utilization of each disk in the cluster. The two graphs represent the performance for different read-ahead buffer sizes per stream.

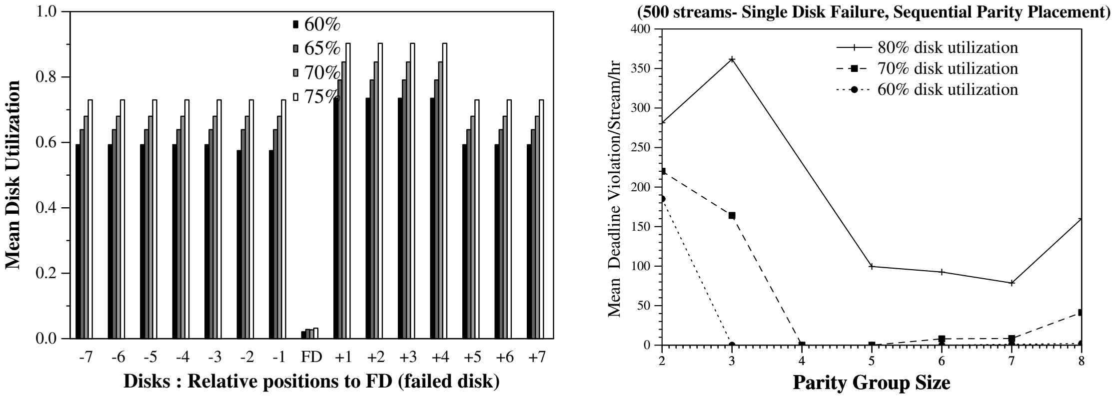
disk utilization is 75% and can be increased to 80% with 3 read-ahead buffers.

4.3.2 Sequential Parity Placement

As discussed earlier, sequential parity placement does not balance the load across disks after failure. Only $P - 1$ disks share the load of the failed disk. This load imbalance affects the real-time performance of sequential parity placement.

Figure 4.7(a) depicts the disk utilization using sequential placement of parity blocks while operating with a single failed disk and a parity group size is 5. The figure shows the disk utilization of individual disks for four different average disk utilization. We observe that although the average disk utilization is as expected, the utilization of the neighboring 4 disks (for P=5) holding the parity blocks is 18% to 20% higher. For example, with a parity group size of 5, when the normal disk operating utilization is 75% the overload during failure is about 19% (75/4), since one-fourth of the parity blocks are on each of the neighboring disks. This drives the disk utilization of the neighboring disks up to about 94%, causing overload and significantly more deadline violations.

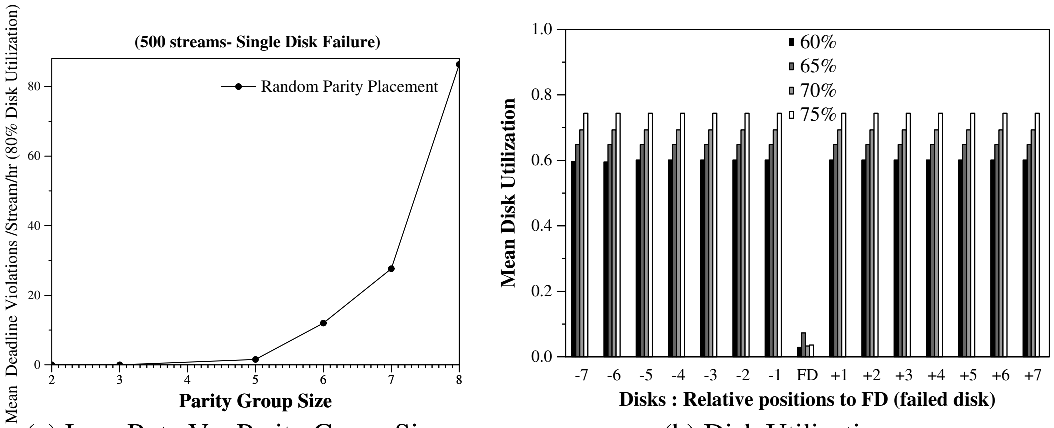
Figure 4.7(b) shows the deadline miss rates for different parity sizes, for a sequential



(a) Disk Utilization (b) Loss Rate Vs. Parity Group Size

Figure 4.7: Sequential Parity Placement. In Figure (a) the disk are labeled relative to their position w.r.t the failed disk in the sequential order. The disk utilization after a single disk failure is plotted for different operational disk utilization. In Figure (b) the y-axis is the average rate of deadline violations per stream per hour after a single disk failure. The x-axis is the parity group size which is selected to be relatively prime to the number of disks.

parity placement scheme with different average disk utilization. With a parity group size of 2, sequential parity placement is equivalent to disk-level mirroring, without spare disks (i.e., each disk stores 50% data and 50% parity blocks). Since the mirrored disk handles the entire load of the failed disk, the overload results in congestion and loss. Two factors affect the rate of deadline violation during failure: (a) the overload at the disks, and (b) the number of parallel requests and XOR overhead. As the parity group size increases, the load on the failed disk is shared by multiple disks, decreasing overload. However, as the parity group size increases, the number of parallel requests increase along with the XOR overhead. The tradeoff between disk overload and parallel request results in a parity group size with the lowest loss rate. The parity group size of 3 (Figure 4.7(b), 80% disk utilization) has the highest deadline miss rate, because the failed disk load is shared by just 2 disks which are already close to peak utilization and there is an additional XOR overhead. For lower disk utilization, the overload shared by the neighboring disks is lower and the deadline miss rates decrease. The disk utilization has to be less than 50% for mirroring (parity group size 2) to have negligible deadline miss rate.



(a) Loss Rate Vs. Parity Group Size (b) Disk Utilization

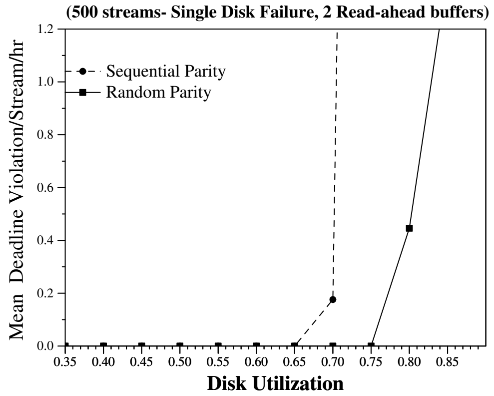
Figure 4.8: Random Parity Placement. In Figure (a) the y-axis is the average rate of deadline violations per stream per hour after a single disk failure. The x-axis is the parity group size. The operational disk utilization is 80%. In Figure (b) the disk are labeled relative to their position w.r.t the failed disk in the sequential order. The disk utilization after a single disk failure is plotted for different operational disk utilization.

4.3.3 Random Parity Placement

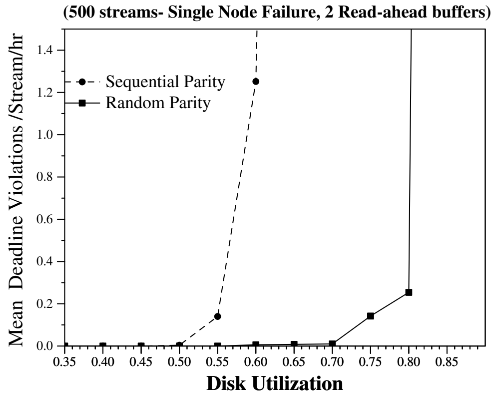
A uniform load across disks during failure results in significantly lower deadline miss rates as observed from Figure 4.8(a). Since there is no load imbalance after failure using the random parity placement, the deadline miss rate is affected only by: (i) the parallel reads of the blocks of a parity group, and (ii) XOR overhead. The affect of these factors increases with an increase in the parity group size. As the parity group size increases the deadline miss rate increases, as shown in Figure 4.8(a).

Figure 4.8(b) shows that the utilization of all disks are nearly equal after failure, with random parity block placement. The figure shows the disk utilization of individual disks for four different average disk utilization and parity group size of 5. We observe that average disk utilization of all the disks increases by around (75/50) 1.5%, for a 75% operation utilization and 50 disk cluster. This is a moderate increase in load and has a significantly lesser impact on the deadline miss rate.

Figure 4.9(a) depicts the mean deadline miss rates per stream over the duration of the simulation with single disk failure and varying operational disk utilization. From Figure 4.9(a)



(a) Disk Failure



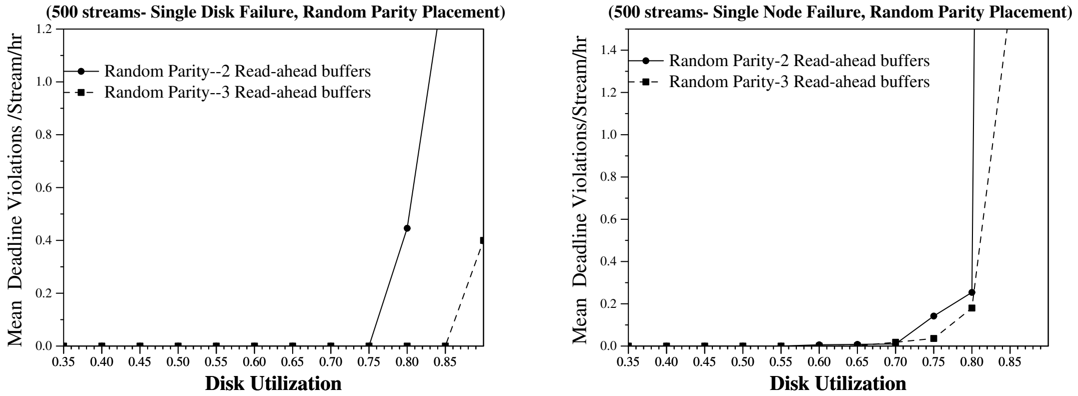
(b) Node Failure

Figure 4.9: Comparison of deadline miss rates of parity placement schemes for single disk and node failure. The parity group size is 5.

we observe that with sequential parity placement, the maximum operational disk utilization is about 65%, while with random parity placement, it is about 75%. With node failure, the deadline miss rates increases as multiple disks become inaccessible. Figure 4.9(b) shows that with node failure, sequential placement can limits the disk utilization to 50% for zero deadline misses, while random placement increases the limit to about 65%.

4.3.4 Read-ahead Buffering

By increasing the read-ahead buffer size, with random placement of parity blocks, the deadline miss rate can be reduced substantially. With increased buffer size, larger variations in response time can be tolerated. The read-ahead buffers and the parity buffers may overlap as they can be mapped to the same physical memory pool. Figure 4.10(a) shows that with disk failure, the maximum operational disk utilization can be increased from about 75% to 85% by increasing the read-ahead buffering from 2 to 3. With node failure (Figure 4.10(b)), the operational disk utilization can be increased from about 65% to 70% by increasing the read-ahead buffering.



(a) Disk Failure (b) Node Failure

Figure 4.10: Read-ahead Buffering. The figures compare the performance of balanced random parity placement for different values of the read-ahead buffer size for disk and node failures.

4.4 Cost Performance Analysis

Thus far, we have focused primarily on the real-time performance (deadline miss rate of stream blocks) as a measure for comparisons of the various schemes. However, there are other factors such as the CPU cost of XOR computation, buffer costs, and disk space costs that merit comparison. In order to combine all these factors into a single metric for comparison, we estimate the overall system cost per stream for a desired QoS (e.g., deadline misses).

For the cost analysis, we want to solve the inverse problem, i.e., given a desired value of QoS, determine the system configuration that achieves it at the minimum cost. First we extend the analytical model for measuring the cluster performance to include failure. Then we build a cost performance model to determine the region of applicability of different schemes, and determine the optimal parity group size for software RAID with balanced random parity placement.

4.4.1 Analytical Model

The analytical model of the cluster discussed in Chapter 2, computed the delay distribution for a data request. For continuous media objects, the QoS was measured by the probability of deadline misses, which is the tail of the delay distribution. In this section we extend the model to handle failures, and use the probability of deadline miss after failure as the QoS measure for continuous media objects.

Let $W(t)$ be the total delay distribution for the cluster. The tail of the delay distribution, i.e., the probability that the delay is greater than D_{ddl}, is given by

$$P[t > D_{ddl}] = 1 - \int_{t=0}^{t<D_{ddl}} W(t) \tag{4.1}$$

In the following discussion we compute the probability of deadline miss on a single disk failure. A similar analysis can be done for a node failure. If a software RAID scheme is used to mask a single disk failure, for every read to the failed disk, multiple requests for blocks in the parity group need to serviced and then used for an XOR computation. Each of these requests should not observe a delay greater than $\hat{D}_{ddl} = D_{ddl} - d_{xor}$. The term d_{xor}, is the time required to do a $(\rho - 2)$ way XOR on blocks in a parity group of size ρ. The overall deadline miss probability under failure, given by Equation 4.2, consists of two parts: probability of deadline miss for computing a read for the failed disk and the probability of a deadline miss for a read to an operational disk. Since one of the disks has failed, the mean arrival rate at the disks, λ_d, is modified to $\lambda S/(D-1)$, in the computation of the delay distribution in the original model (from Equation 2.3).

$$\begin{aligned}
Pr[t > D_{ddl}\ on\ failure] &= (1 - (\int_{t=0}^{t<\hat{D}_{ddl}} W(t))^{\rho-1})(Pr[req.\ to\ failed\ disk]) + \\
&\quad (1 - \int_{t=0}^{t<D_{ddl}} W(t))(Pr[req.\ to\ surviving\ disk]) \qquad (4.2)\\
Pr[request\ to\ failed\ disk] &= 1/D \\
Pr[request\ to\ surviving\ disk] &= 1 - 1/D
\end{aligned}$$

With mirroring the parity group size ρ, is equal to two, and the above equation becomes identical to Equation 4.1. The maximum operational disk utilization U_d, for a given deadline miss probability $\mathcal{L}$, is the value of ρ which satisfies the condition

$$U_d = \rho \ \mid \ Pr[deadline\ miss\ under\ failure|\rho] \leq \mathcal{L}$$

4.4.2 Cost-Performance Comparisons

The system cost includes the cost of the nodes, cost of disks and the memory cost. Table 4.2 gives the cost parameters used in the comparison. Equation 4.3 gives the system cost for mirroring C_{mir}, and the corresponding cost for software-RAID, C_{RAID}.

	Parameter	Default Value
O	Number of media objects	100
L	Duration of a media object	2 hr
λ	Playout rate of a stream	0.5 MB/sec
U_d^{RAID}	Operational disk utilization (given QoS and parity group size)	
U_d^{mir}	Operational disk utilization (given QoS and mirroring)	
U_n^{RAID}	Operational node utilization (given QoS and parity group size)	70%
U_n^{mir}	Operational node utilization (given QoS and mirroring)	70%
B	Read-ahead buffer size for a given QoS	2
P	Parity group size	5
C_d	Cost of a disk	\$1800
C_b	Cost of a buffer (256KB block, \$30/MB)	\$7.50
C_n	Cost a node	\$10000
D_s	Capacity of a disk	4 GB
D_{bw}	Effective Bandwidth of the disk for a given block size	
N_{bw}	Effective Bandwith of the node for a given block size	
S	Number of streams	500

Table 4.2: Cost Parameters. The cost and capacity parameters were selected from the average cost and capacity of disks and memory in 1994.

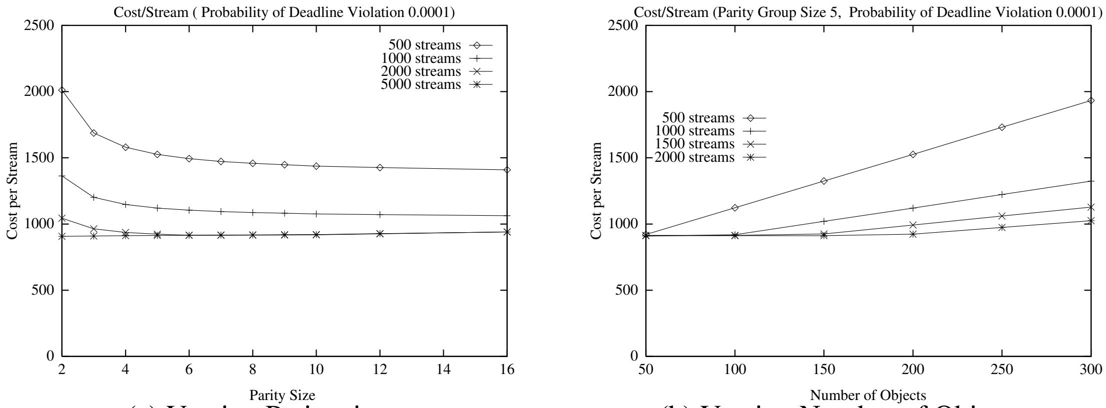
$$C_{mir} = \overbrace{max\{\frac{2OL\lambda}{D_s}, \frac{\lambda S}{D_{bw}}\}C_d/U_d^{mir}}^{DiskCost} + \overbrace{BC_bS}^{BufferCost} + \overbrace{(\frac{\lambda S}{N_{bw}})C_n/U_n^{mir}}^{NodeCost} \tag{4.3}$$

$$C_{RAID} = \underbrace{max\{\frac{OL\lambda P}{D_s(P-1)}, \frac{\lambda S}{D_{bw}}\}C_d/U_d^{RAID}}_{DiskCost} + \underbrace{BC_bS}_{BufferCost} + \underbrace{(P-2)C_bS/N}_{Parity-bufferCost} + \underbrace{(\frac{\lambda S}{N_{bw}})C_n/U_n^{RAID}}_{NodeCost} \tag{4.4}$$

The number of disks used in the comparison is determined based on the maximum of: (i) the aggregate disk bandwidth at the allowable disk utilization for a given deadline miss rate and, (ii) the disk space requirements for a given availability scheme. The disk space requirement depends on the number of objects stored in the server. The disk bandwidth requirement depends on the total number of streams that will be playing out at any given time. The number of nodes is determined based on the node bandwidth requirement. The buffer cost is the memory overhead of the read-ahead buffer. The parity-group buffer cost is based on the node failure case, where an average of S/N streams access the failed node at a given time. For handling only disk failure, the parity-group buffer cost is reduced to $(P-2)C_bS/D$.

In the cost comparisons we only consider block level mirroring and software RAID with balanced random parity placement. Mirroring is cost-effective only when missing real time constraints during failure is more expensive than the extra disk capacity. When the disks are bandwidth constrained, (i.e., the number of disks is determined by the disk bandwidth and not by the storage requirements) then mirroring is not expensive as disk space comes for free. If a large number of long continuous media objects are stored on the server, the disks are constrained by the storage capacity, (i.e., the number of disks is determined by the space requirements of the video objects). When the disks are space constrained, the space requirement for mirroring is $2(P-1)/P$ times that for software RAID with parity group size of P. With software RAID, when the parity group size is large (> 3), the deadline miss rate under failure for an identical system configuration is worse than that with mirroring. The deadline miss rate can be lowered by increasing the read-ahead buffer sizes or operating the disks at a lower utilization. This adds an additional memory and disk cost in order to maintain similar deadline miss rates. Software RAID becomes cost effective only when mirroring causes the disks to be space constrained.

Based on the operating disk utilization determined by the analytical model, Figure 4.11(a) shows the cost per stream for different numbers of streams with varying parity group sizes for a fixed number of objects (the term movies and continuous media objects is used interchangeably). With 500 streams, the disks remain space constrained and the cost per stream decreases with larger parity group sizes. For 5000 streams the disks are bandwidth constrained and the cost per stream gradually increases with larger parity sizes. Figure 4.11(b) shows the disks changing from bandwidth constrained to space constrained as the number of objects are increased. Figure 4.12(a) shows the cost trends with varying number of streams for different



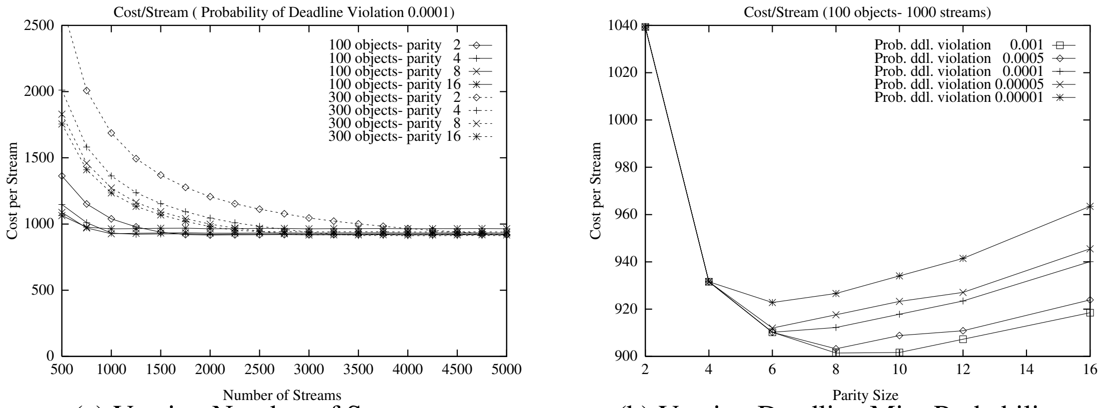
(a) Varying Parity size (b) Varying Number of Objects

Figure 4.11: System Cost for Disk Failure. Figure (a) compares the overall system cost for varying parity group sizes, for different number of streams served by the system at a given probability of deadline violation. Figure (b) compares the cost for a parity group size of 5 with varying number of total objects stored in the system.

parity group sizes. Mirroring (parity group size 2) remains cost-effective only beyond 1750 streams with 100 objects and beyond 5000 streams with 300 objects.

The effect of varying the values of acceptable deadline miss rates is shown in Figure 4.12(b). From the analytical model, the maximum disk utilization to maintain the given loss rate can be determined for different parity sizes. For a library of 100 objects, the optimal parity size varies with the deadline miss probability parameter. As the deadline miss tolerance decreases, the cost increases due to the lowering of operational disk utilization. The optimal parity group size also decreases with decreasing deadline miss tolerance. For example, for a deadline miss probability of 0.00001, the cost per stream is minimum at a parity group size of 6, while for a deadline miss probability of 0.001 it is 10.

For a given system capacity (number of streams) and QoS criterion (deadline miss rate), the optimal parity size that minimizes the cost can be determined for different number of objects in the server. At lower parity group sizes, the extra space overhead could make the disks space constrained. In such cases the disks will not operate at the maximum bandwidth utilization governed by the deadline miss rate, increasing the system cost. Figure 4.13 shows the optimal parity size (least cost per stream) for a 2000 stream system, with the number of video objects



(a) Varying Number of Streams (b) Varying Deadline Miss Probability

Figure 4.12: System Cost for Disk Failure. Figure (a) compares the cost per stream for varying number of streams. The curves are for different parity groups sizes and the number of objects stored in the system. The probability of deadline violation is 0.0001. In figure (b) the cost per streams is determined for varying parity groups sizes. Each curve represents a different tolerance of deadline violations. The system consists of 100 objects and services 1000 streams.

ranging from 50 to 300. After 100 objects, the disks become space constrained at lower parity group sizes and the optimal parity size increases from 2 (mirroring) to 4 and 6.

4.5 Client Connections and Front-end Node Failures

In our discussion so far, a node failure was assumed to be that of a back-end node. In case of a front-end node failure, the client connection is broken. We rely on known distributed recovery schemes for masking front-end node failures from the client [42].

Node failures are detected by using a heart-beat and membership protocol [69] that essentially eliminates nodes from the membership group in a consistent manner, when (several) heartbeat messages from the node are not received. For front-end node failure, if the DNS or router solution (as discussed in Section 3.2) is used for load balancing, the failed node is removed from either the name server tables or the router configuration table. Once these tables are updated to reflect the failure, all remote clients that try to make a connection after the failure will not be directed to the failed node. However, in the DNS approach some clients may have cached the address of the failed node and will continue to try connecting to it. Similarly, with

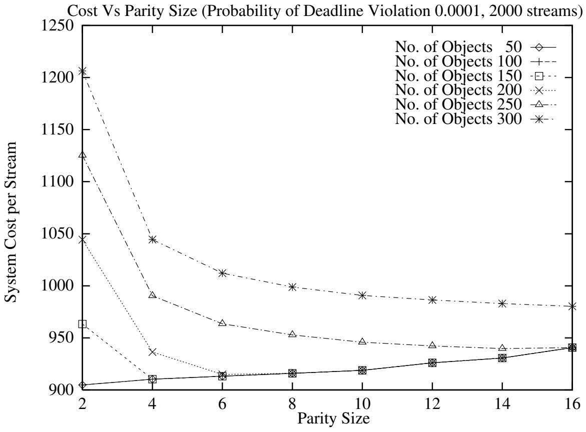


Figure 4.13: Optimal Parity Group Size (2000 streams)

the router based approach a router node may fail causing clients to see a failure. To handle these scenarios each front-end node and router node has a "buddy" which takes over its IP address and connection after failure. Remote clients that have cached the address of the failed node will connect to the buddy transparently. For IP takeover the nodes have a redundant adapter. One of the adapters is used for servicing all the external IP requests and one for booting. The IP address of the main adapter is taken over by a spare adapter in the buddy node. The switching is done in software and is orchestrated by a recovery driver. Details of the techniques used are beyond the scope of this dissertation.

4.6 Related Work

Most previous work on high data availability, examine single node servers and specifically focus on RAID schemes across disks within a node [6, 7, 111, 120].

Several research efforts have focused on guaranteeing availability for continuous media servers. The streaming RAID proposed by Tobagi et al. [111], divides the disks into *parity groups*[1]. In their scheme, entire parity groups are read out in one cycle and sent on the network during the next cycle. Berson et. al., generalize the streaming RAID approach [7]. Their

[1]While they refer to parity groups as clusters of disks, we avoid this terminology as we focus on a cluster of nodes

staggered stripe scheme, reads a parity group in a single cycle, but plays it out over *n* cycles, roughly halving the memory requirements. Their non-clustered scheme has the entire stripe group read in a cycle, only under failure conditions. Similarly, there are numerous other research efforts on RAID, mirrored data layout, and declustering schemes [16, 21, 93, 60, 61], but these do not focus on clusters with heterogeneous data types.

Previous work that considers high availability in a clustered environment that we know of is by Frey [44] and the work of Haskin [59]. Frey et. al., describe a round-robin layout across nodes and disks, but they do not consider real-time applications. Haskin et. al. consider only mirroring across nodes in clustered servers.

4.7 Prototype Implementation

We developed a prototype implementation of a clustered server and used it to instantiate our design choices. In a clustered architecture, one of the design principles for scalability is that every node in the cluster has an identical view of all the I/O resources. This enables incremental growth, since applications and users need not adapt to the addition or removal of storage nodes and disks.

To provide a "single system image" to the cluster applications we use a layer of software called *virtual shared disk* (VSD) [4]. VSD is a software disk proxy that allows a remote I/O device to appear as if it was a local device. The VSD software is packaged as a device driver and appears to the system as a logical volume or hard disk, providing the same interface to the application as a local device. Furthermore, all processing for a request at the VSD server node, where the physical disk is connected, is done at the interrupt level; there is no associated kernel process or daemon. This is in contrast to network filesystems where daemons are used at server nodes. Consequently, VSD is able to sustain bandwidths that are several times more than that of networked file systems. The VSD software makes the cluster architecture and remote I/O handling transparent to the nodes. A standard UNIX based I/O interface is provided by VSD with the requests being ordered by their arrival time. The enhanced version, called real-time VSD, handles requests with deadlines. This is done by specifying an *ioctl* interface, which passes a deadline along with the block addresses. The deadline based ordering is done at the VSD server queue. There are also ioctls defined for updating a deadline and killing outstanding requests whose deadlines have expired. A layer of application software runs on the front-end nodes that interacts with VSD to fetch the blocks of a video object at a given rate and a given priority level.

The software layer at the front-end nodes, maintains the meta-data associated with each object in the system. The logical block represents the size of a request to the VSD, which could be mapped to multiple packets within the switch communication layer and to multiple physical blocks on disk. Typically a block is split into multiple 64 Kbyte packets at the switch communications layer. The packet size is a tunable parameter. We use network striping across nodes in our prototype. The balanced random placement scheme is used for blocks within a stripe group. Disks can be added or removed from a stripe group causing the data to be reorganized. The size of a logical block on disk is flexible, although the typical size used is 256KB which is in the optimal size range discussed in Section 3.1.3.2. For failure handling the balanced random parity placement scheme is used. Since continuous media objects could be large ($>$ 2 GB) VSD supports large video objects even on 32-bit machines. The prototype allows for either flat or two-tier architectures. Since the number of clients is small in the prototype, the streams can be played out to a dummy device to measure the performance of the system. Each node can push around 20 MBytes/sec of stream data at about 80% CPU utilization. The number of disks per nodes is small in the current prototype, leading to the disks becoming the bottleneck and hence the nodes are driven to only half their potential bandwidth.

4.8 Summary and Conclusions

To address the goal of high availability for clustered servers, we propose and study the performance of a software RAID scheme with declustering to support single disk and node failures. We observe that sequential placement (with primality constraint) of parity blocks of objects, balances the space and bandwidth utilization of all disks only during normal operation. During failure only a few ($P-1$ for a parity group size P) of the disks share the load of the failed disk, leading to load imbalance. With balanced random placement of parity blocks, the space and bandwidth of all disks is equally utilized during both failure and normal operation. Balanced random placement of parity blocks, spreads the load due to parity accesses across all disks and makes software RAID a cost-effective technique that performs well under both node and disk failures. We propose the multi-stride parity placement as an efficient implementation of the balanced random parity scheme.

We describe a cost model of the system and use it to determine the region of applicability of different parity sizes for a given QoS requirement. For a given system configuration an optimal parity size can be determined that minimizes the cost. Although mirroring has the best real-time performance, it is cost-effective only when the disks are bandwidth constrained and the space overhead is not critical. For stringent real-time requirements, the optimal parity group size decreases, to reduce overheads of parallel reads and XOR computation. On the other hand, larger parity group sizes have smaller disk space overheads but larger memory costs, resulting in a relatively small optimal parity group size.

To provide a single storage view of all the back-end nodes and disks in the system, our prototype used the virtual shared disk abstraction. We briefly described the prototype and discussed some implementation details.

Chapter 5

Scalable Distributed Cache Design

With twenty five years of Internet experience we've learned exactly one way to deal with exponential growth—caching.

—Van Jacobson

5.1 Introduction

The goal of a wide-area information system is to enable increasing numbers of users to access vast amounts of information stored at geographically distributed sites. Due to the non-uniformity of information access, however, popular objects create "hot spots" of network and server load, thereby, limiting the scalability and significantly increasing the latency for information access. Large-scale distributed caches appear to provide an opportunity to combat this latency because they allow users to benefit from data fetched by other users [33, 54], and their distributed architectures allow clients to access nearby copies of data in the common case. Our vision of a distributed caching architecture consists of a very large number of caches that enable efficient data sharing among an increasing client population. The individual caches are close to the client population they serve. This view is in contrast to the current hierarchy of a few super caches distributed across a wide geographical region. Current web cache systems define a hierarchy of data caches in which data access proceeds as follows: a client sends a request to a cache, and if the cache contains the data requested by a client, it replies with the data.

Otherwise, the cache may ask its neighbors for the data, but if none of the neighbors possess the data, then the cache sends its request to its parent. This process recursively continues up the hierarchy until the data is located or the root cache fetches the data from the server specified in the request. The caches then send the data down the hierarchy to the client, and each cache along the path caches the data. Unfortunately, these hierarchies of data caches often achieve modest hit rates [2, 33, 51, 54], can yield poor response times on a cache hit [81, 101], and can slow down cache misses.

The goal of our distributed caching architecture is to: (i) improve performance, (ii) scale to a large number of users and caches, and (iii) support the requirements of heterogeneous data types. In the following three chapters, we outline design principles and strategies to meet the above goals.

In this chapter, we first attempt to understand the factors that limit the performance of current web caching hierarchies through measurements of several caches on the Internet and analysis of several traces of web traffic. Based on these measurements, we derive four basic design principles for improving the performance of large-scale caches: (1) minimize the number of hops to locate and access data, (2) do not slow down misses, (3) share data among many caches, and (4) cache data close to clients. Although these principles may seem obvious in retrospect, we find that current cache architectures routinely violate them at a significant performance cost. Based on these principles, we have designed, simulated, and implemented two strategies for organizing large-scale, distributed caches.

Our first strategy addresses the first three principles for improving performance. This strategy has three salient features. First, it separates data paths from metadata paths and maintains a hierarchy of metadata that tracks where copies of data are stored. Second, it maintains location hints so that caches can locate nearby copies of data without suffering network latencies. Third, it uses direct cache-to-cache data transfers to avoid store-and-forward delays. Thus, whereas a traditional cache hierarchy often requires both requests and responses to traverse several caches in the hierarchy, our architecture minimizes the number of hops through the hierarchy by using location hints to go directly to the nearest cached copy of data and by using cache-to-cache data transfers to send the data directly back. Whereas a traditional hierarchy may forward cache misses through several caches before going to the server, caches in our system use hints to detect misses locally and then proceed directly to the server. A novel aspect of this strategy is that although it uses a hierarchy to distribute metadata, it only stores data at the leaves of the hierarchy, and it always transfers data directly between caches rather than through the metadata hierarchy.

Our second strategy addresses the fourth design principle by redistributing data across the caches in the system. It does this by "push caching" [57] data near clients that have not referenced the data but that are likely to reference it in the future. Thus, it reduces or eliminates the compulsory miss penalty normally experienced the first time a particular client accesses an object. We bound the performance achievable by such algorithms and then examine two simple and efficient algorithms for choosing what data to push. One algorithm pushes data that have been recently updated, and the other pushes data that are widely shared.

Furthermore, our metadata hierarchy design facilitates widespread sharing of data among large numbers of caches. Several aspects of the design address scalability by: (i) reducing space overhead by compressing the size of hints so that hint caches can track the contents of many data caches, (ii) using a metadata distribution hierarchy to distribute hints with low bandwidth overhead and propagation delay, (iii) dynamically configuring the hierarchy to automate the management of systems with many caches and support incremental growth, and (iv) creating multiple hierarchies and load balancing among them.

We evaluate our design strategies using simulation studies of three large workloads that contain accesses by users of Digital Corporation's proxy server [32], UC Berkeley's Home-IP service [54], and Prodigy ISP's dial-up web browsing service. Our algorithms focus on *improving hit and miss response times* rather than trying to improve global hit and miss rates. Thus, we parameterize our results using estimates of Internet access times based on our own measurements and those in the literature [101]. For these traces and network configurations, we find that our hint architecture provides speedups of 1.27 to 2.30 compared to a standard, three-level cache hierarchy and that push caching improves performance by up to an additional factor of 1.25. Overall, our techniques provide speedups of 1.27 to 2.43 compared to a traditional hierarchy.

We have implemented a prototype by augmenting the widely-deployed Squid proxy cache [123]. It implements a hint distribution hierarchy, local hint caches, and cache-to-cache data transfers. A key insight gained in this implementation was an understanding of the desirability of storing hints as small, fixed-sized records. The prototype has been in day-to-day use at our site for several months.

In this study we focus on a collective cache environment where caches are willing to cooperate to some degree to provide improved performance for all of their clients. We believe our model will be interesting to system designers building large-scale, distributed cache infrastructures in a range of environments including network service providers, independent service providers, cache service providers [24, 85, 108, 123], collections of caches linked by formal service agreements [107], and large intranets. Also, once the trade-offs in these cooperative environments are understood, we plan to examine ways to apply some of these techniques in more competitive environments.

The rest of the chapter is organized as follows. Section 5.2 evaluates the performance of traditional cache hierarchies and examines the characteristics of several large workloads and then derives a set of basic design principles for large-scale, distributed caches. Section 5.3 examines how to separate data and metadata paths in these caches to improve response times of requests. Section 5.5 discusses techniques for distributing data across caches to increase the frequency of hits to fast, nearby caches. Section 5.6 surveys related work, and Section 5.7 summarizes our conclusions and outlines areas for future research.

5.2 Evaluating Traditional Cache Hierarchies

In this section, we evaluate the performance of traditional cache hierarchies using measurements of several caches on the Internet and trace-driven simulations, with the goal of understanding the factors that limit cache performance. Based on these measurements, we derive the four design principles for distributed caching on the Internet that are summarized in Table 5.1.

Design Principle	Strategy
Minimize the number of hops to locate and access data	Separate data and metadata, local hints, and cache-to-cache transfer
Do not slow down misses	Location-hints
Share data among many caches	Separate data paths and metadata paths location-hints
Cache data close to clients	Push caching

Table 5.1: Design principles and strategies for distributed caching in the Internet

5.2.1 Access Costs in a Traditional Cache Hierarchy

Traditional hierarchical cache architectures such as Harvest [19] or Squid [123] define parent-child relationships among caches. Each cache in the hierarchy is shared by a group of clients or a group of children caches. In such cache hierarchies, data access proceeds as follows: if the lowest-level cache contains the data requested by a client, it sends the data to the client. Otherwise, the cache asks each of its neighbors for the data. If none of the neighbors possess the data, then the cache sends a request to its parent. This process recursively continues up the hierarchy until the data is located or the root cache fetches the data from the server. The proxies then send the data down the hierarchy and each cache along the path caches the data.

Although hierarchies such as Harvest and Squid were designed under the assumption that caches could be layered without adding much delay [19], we hypothesize that two aspects of this architecture as applied to Internet caches can significantly limit performance. First, the cost of accessing a series of caches in the hierarchy adds significant "store-and-forward" delays to higher-level cache hits and to cache misses [84]. Second, when high-level caches service a large number of clients distributed over a large geographic region, the network delays between a client and a high-level cache may be large, which reduces the benefit of hits to far-away caches.

To understand these effects better, we use two sources of information. First, to understand the details of performance in a controlled environment, we construct a test hierarchy and examine it under a synthetic workload. Second, to understand how such systems perform in real hierarchies and under real workloads, we examine Rousskov's measurements of several Squid caches deployed at different levels of a hierarchy [101]. Although Squid supports the Internet

Cache	Location	Machine
Client	CS department, UC Berkeley	166MHz Sun UltraSparcs
L1	CS department, UC Berkeley	166MHz Sun UltraSparcs
L2	CS department, UC San Diego	150MHz DEC 2000 Model 500
L3	CS department, UT Austin	166MHz Sun Ultra2
Server	CS department, Cornell University	DEC Alpha

Table 5.2: Testbed hierarchy setup

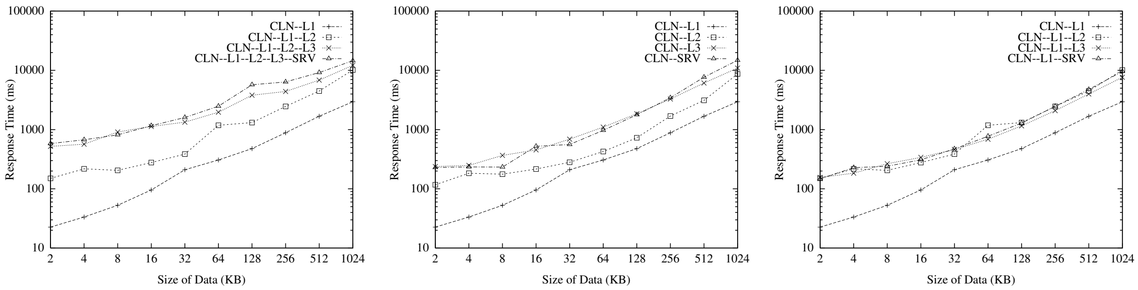
Cache Protocol (ICP) to allow a cache to query its neighbors before sending a miss to a parent [124], we are interested in the best costs for traversing a hierarchy, so neither configuration we examine uses ICP.

Both analysis yield similar results. Both suggest that there is a significant per-hop cost for traversing multiple levels of cache. This yields the first design principle in Table 5.1: *minimize the number of hops to locate and access data*. Both also support the hypothesis that distant caches may be expensive to access even without this per-hop cost due to network limits. This yields the third design principle in Table 5.1: *cache data close to clients.* In what follows, we describe both the experiments.

5.2.1.1 Analysis of a Testbed Hierarchy

We constructed a testbed to examine the relationship among caches arranged in a large, three-level hierarchy in which the level-1 (L1) cache services a department, a level-2 (L2) cache services a state, and a level-3 (L3) cache services a large region. Table 5.2 details our testbed hierarchy. Although these caches are distributed across a large geographic region, they are relatively well connected, so some less ambitious hierarchies may have similar characteristics. Each cache in this setup runs version 1.1.17 of the Squid proxy cache. The client and L1 cache are connected by a switched 10Mbit/s Ethernet. All of the sites are connected to the Internet via 45Mbit/s T3 connections, and the backbone connections between all sites are at least T3.

For our experiments, we arranged for a specific level of the cache to contain an object of a specified size and for the other levels of the cache not to contain that object. We then used an instrumented client program to time how long it took to get that object from the hierarchy.



(a) (b) (c)

Figure 5.1: Measured access times in the testbed hierarchy for objects of various sizes: (a) objects accessed through the three-level cache hierarchy; (b) objects fetched directly from each cache and server; and (c) all requests go through the L1 proxy and then directly to the specified proxy or server.

We repeated this experiment 10 times for each configuration of the caches over the course of 3 hours during the late afternoon on several weekdays and discarded the high and low values observed. Each data point in the graphs in Figure 5.1 represents the mean of the remaining eight measurements.

Note that, in our experiments, the caches were idle other than our requests, which were made one at a time. If the caches were heavily loaded, queuing delays and implementation inefficiencies of the caches [81, 5] might significantly increase the per-hop costs we observe. Busy nodes would probably increase the importance of reducing the number of hops in a cache system.

Figure 5.1 summarizes the performance of the testbed hierarchy. Figure 5.1(a) shows the performance the system uses the standard three level data hierarchy. In contrast, Figure 5.1(b) shows the access time when the Berkeley client accesses the Berkeley, San Diego, and Austin caches *directly* by circumventing the hierarchy. Figure 5.1(c) shows the case when direct accesses must always go through the L1 cache such as when the L1 cache acts as a firewall for the clients. These measurements support and quantify the intuition that accessing a series of caches in a hierarchy incurs a significant cost. For instance, the difference between fetching an 8KB object from the Austin cache as part of a hierarchy compared to accessing it directly is 545 ms. Put another way, if the system could "magically" send requests directly to the correct level of the hierarchy and that level of the hierarchy send the data directly back to the client that needs

	Client Connect		Disk		Proxy Reply	
	min	max	min	max	min	max
Leaf	16ms	62ms	72ms	135ms	75ms	155ms
Intermediate	50ms	550ms	60ms	950ms	70ms	1050ms
Root	100ms	1200ms	100ms	650ms	120ms	1000ms
Miss	550ms 3200ms					

	Total Hierarchical		Total Client Direct		Total via L1	
	min	max	min	max	min	max
Leaf	163ms	352ms	163ms	352ms	163ms	352 ms
Intermediate	271ms	2767ms	180ms	2550ms	271ms	2767ms
Root	531ms	4667ms	320ms	2850ms	411ms	3067ms
Miss	981ms	7217ms	550ms	3200ms	641ms	3417ms

Table 5.3: Summary of Squid cache hierarchy performance based on Rousskov's measurements.

it, a level-3 cache hit time could speed up by a factor of 2.5 for an 8KB object. In Section 5.3, we propose an architecture that approximates this ideal.

Figure 5.1(b) also indicates that even if a cache architecture were able to avoid the cost of multiple-hop store-and-forward, accessing distant caches is still more expensive than accessing nearby ones. This conclusion is not surprising; accessing a distant node on the Internet will increase latency due to increased routing distance and increased risk of encountering routing delays, and it will decrease bandwidth for analogous reasons. This experiment suggests that in addition to reducing the number of hops needed to access distant data, cache hierarchies should take action to access nearby data as often as possible. In Section 5.5, we propose algorithms that address this design principle.

5.2.1.2 Analysis of Squid Hierarchies

Rousskov [101] has published detailed measurements and performance analysis of several Squid caches that are deployed on the Internet in the United States and Europe. We use that information to derive the estimates for the minimum and maximum access times to different cache levels summarized in Table 5.3.

We derive these estimates by computing the time that a leaf, intermediate, and root cache took to service a hit for a request sent from the next level down in the hierarchy. Rousskov breaks down hit response time into three components: "Client connect," which is the time from when the cache's accept() system call returns until the cache has a parsable HTTP request, "Disk," which is the time it takes to swap in data from the disk, and "Proxy reply," which is the time it takes to send the data back on the network. Since Rousskov gathered the measurements at the proxy caches, we must estimate the miss time by looking at his measurements of how long the top-level proxy spent waiting to connect to and receive data from miss servers. Rousskov measured these values over a 24-hour period for each cache and publishes the median value during each 20-minute period. Table 5.3 shows the minimum and maximum values seen for these 20-minute medians between the peak load hours of 8AM to 5PM.

We estimate bounds on the time to access an object in each level of a hierarchy by adding the minimum or maximum client connection and proxy reply times for all levels that must be traversed and then adding the disk response time for the highest level cache accessed. We estimate bounds on the total response time to directly access some level of the cache by adding the minimum or maximum client connection time, disk response time, and proxy reply time. Note that this calculation does not account for possible pipelining between the disk response time and the proxy reply time.

These results support the same general conclusions as the measurements of our testbed: hops are expensive, and accessing far away caches is expensive. These data suggest cache systems may pay a particularly high penalty for accessing distant caches during periods of high load. For example, directly accessing a leaf cache during periods of low load costs 163 ms which is twice as fast as the 320 ms cost of directly accessing a top level cache. However, the top level cache experienced a 20-minute period during which the median access time was 2850 ms, whereas the worst 20-minute median access time for the leaf cache was 352 ms. Our interpretation is that although accessing distant caches can be tolerable in the best case, caching data near clients may be an important technique for insulating clients from periods of poor performance.

5.2.2 Workload Characteristics

In this subsection we examine how characteristics of web workloads stress different aspects of large cache systems in order to understand what techniques will be important for improving performance in such systems. This analysis yields the second and third principles listed in Table 5.1:

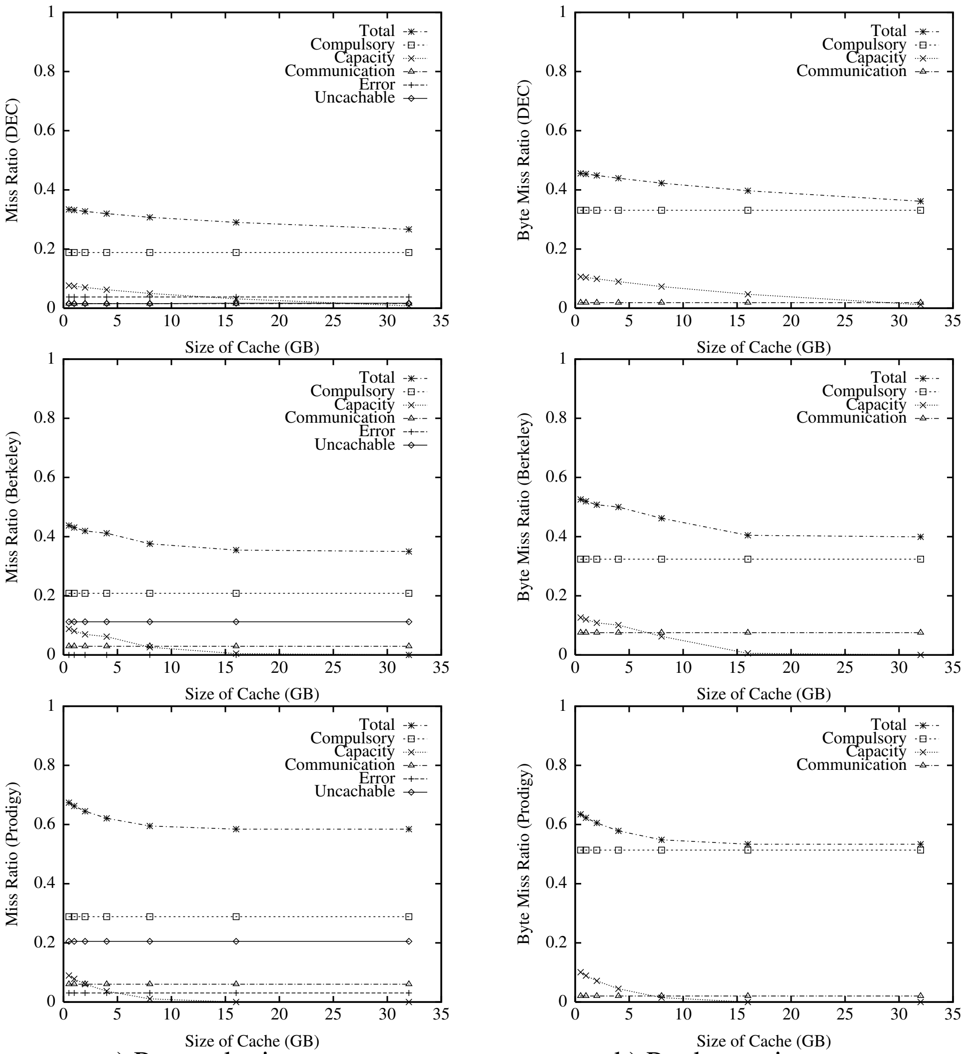
- Even an ideal cache will have a significant number of compulsory misses (misses to objects never referenced before in that cache) and communication misses (misses to objects that have changed since they were last referenced.) Thus, *cache systems should not slow down misses*.

- Cache systems should facilitate strategies that *share data among many clients* to reduce compulsory misses in low-level caches.

Analysis of these workloads leads us to make two other observations. First, we expect capacity misses (misses to objects that have been replaced due to limited cache capacity) to be a secondary consideration for large-scale cache architectures because it is economical to build shared caches with small numbers of capacity misses. If more aggressive techniques for using cache space are used (for example, push caching), capacity may again be a significant consideration. Second, cache architectures should be scalable to allow large numbers of clients to share the cache to achieve good global hit rates. Scalability means that caches should avoid centralized bottlenecks and that the distance between clients and caches should not increase as system size increases.

5.2.2.1 Sources of Cache Misses

In this study we use three large-scale traces taken at proxies serving thousands of clients. These include users of Digital Corporation's proxy server [32], UC Berkeley's Home-IP service [54], and Prodigy ISP's dial-up web browsing service. Figure 5.2 shows the breakdown of cache miss rates and byte miss rates for a global cache shared by all clients in the system as cache size is varied; the cache uses LRU replacement, and the figure caption explains the categories of misses



a) Per-read miss rate b) Per-byte miss rate

Figure 5.2: Request miss rates and byte miss rates for the traces examined in this study. For each graph, we categorize misses as *capacity* for accesses to data that have been discarded from the cache to make space for other data, *communication* for accesses to data that have been invalidated from the cache due to an update, *error* for requests that generate an error reply, *uncachable* for requests that specify that the cache must contact the server to retrieve the result, for non-GET requests, or for requests that are designated as CGI requests in the traces, and *compulsory* for the first access to an object.

we examine. For all of these traces, even an ideal cache will suffer a significant number of misses. Most of these misses are compulsory misses that occur the first time a client in the trace references an object. For the Berkeley and Prodigy traces, there are also a significant number of uncachable requests and communications misses. For multi-gigabyte caches, capacity misses are a relatively minor problem compared to the other sources of misses.

Given the relatively high rate of misses even with large caches, cache systems cannot afford to slow down misses. For example, in the DEC traces 19% of all requests result in compulsory misses (e.g., they are the first request to the data by any client in the trace). For these requests, any time spent in the cache system is wasted and will hurt overall response time.

Reducing the quantity or cost of global compulsory, uncachable, and error misses may become important if techniques such as those suggested by this study prove to be effective at harvesting the remaining potential hits and improving hit time. However, we do not address that problem here. Possible avenues include increasing the number of clients sharing a cache system [33, 54], allowing servers to send new data to cache systems before any nodes in the system request the data, providing better cache consistency [58, 80, 127] to reduce the number of requests marked as uncachable, caching dynamically generated results [113], dynamically replicating servers [115], and negative result caching [83, 19]. Since we are interested in studying the effectiveness of caching strategies, for the remainder of this study, we do not include "Uncachable" or "Error" requests in our results.

5.2.2.2 Sharing

Figure 5.3(a) illustrates the importance of enabling widespread sharing in large-scale cache systems. In this experiment, we configure the system as a three-level hierarchy with 256 clients sharing a L1 proxy, eight L1 proxies (2048 clients) sharing a L2 proxy, and all L2 proxies sharing an L3 proxy. As more clients share a cache, the compulsory hit rate for that cache falls because it becomes less likely that any given access to an object is the first access to that object. For example, in the DEC traces the hit rates improve from 50% for L1 to 62% for L2 and 78% for L3. Gribble and Brewer [54] and Duska et. al [33] reach similar conclusions. This characteristic of the workload suggests that cache systems should accommodate large numbers

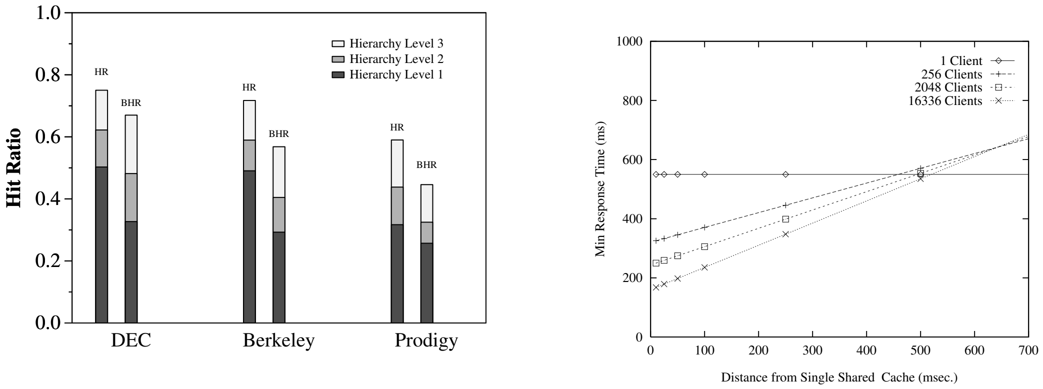


Figure 5.3: Figure (a) shows the overall per-read and per-byte hit rate within infinite L1 caches shared by 256 clients, infinite L2 caches shared by 2048, and infinite L3 caches shared by all clients in the trace. As sharing increases, so does the achievable hit rate. Figure (b) shows the response time for caches with different degrees of sharing as the distance between clients and caches increase.

of clients and thereby reduce compulsory miss rates by avoiding bottlenecks that limit system scalability. Additionally, such systems will provide more scalable performance if the distance between clients and caches increases slowly as system size increases.

For caches with different degrees of sharing, Figure 5.3(b) depicts the variation in the request response times with increase in the distance between clients and the shared cache. It illustrates a dilemma faced by the designers of shared proxy caches: although it is important to share a cache among many clients, it is also important that the shared cache be close to clients. For example, a 256-client cache with an average hit time of 50 ms can outperform a 16,336-client cache that averages 300 ms per access.

We will use this 3-level configuration of proxies as our default hierarchy for the remainder of this chapter.

5.2.3 Summary

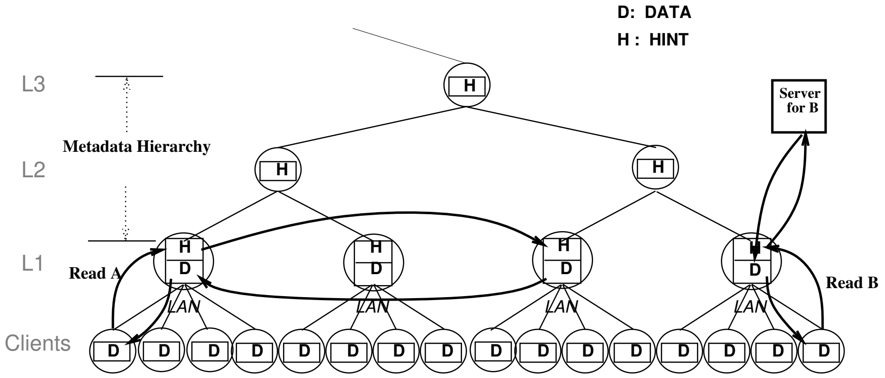
Although the design principles suggested by these measurements seem simple, traditional hierarchies of data caches violate them in various ways. First, a request may have to traverse several hops in a cache hierarchy to get to the data, and the data may traverse several hops to get back to the client. Second, cache misses will be significantly delayed by the having to traverse the hierarchy. With respect to the third principle, cache hierarchies appear to be able to do a relatively good job at sharing data among many caches: the root cache in such a hierarchy will see misses from all clients, so only the first client to access data will suffer a compulsory miss, and only the first client to access data after it changes will suffer a communications miss. Unfortunately, the shared, high-level caches in hierarchies also tend to be far away from clients with the most widely shared caches generally being the most distant. Thus, the strategy used to improve sharing in such hierarchies makes it hard to follow the fourth rule, cache data close to clients.

This analysis suggests that the focus of a new cache architecture should be on improving hit and miss times, while providing a scalable way to to maximize sharing to maintain good hit rates. As noted in Section 5.2.2.1, further improvements to global hit rates may require more radical techniques for dealing with communications and compulsory misses and uncachable requests, which are outside the scope of this study. There does, however, appear to be significant room for improving the hit and miss response times of distributed caches on the Internet.

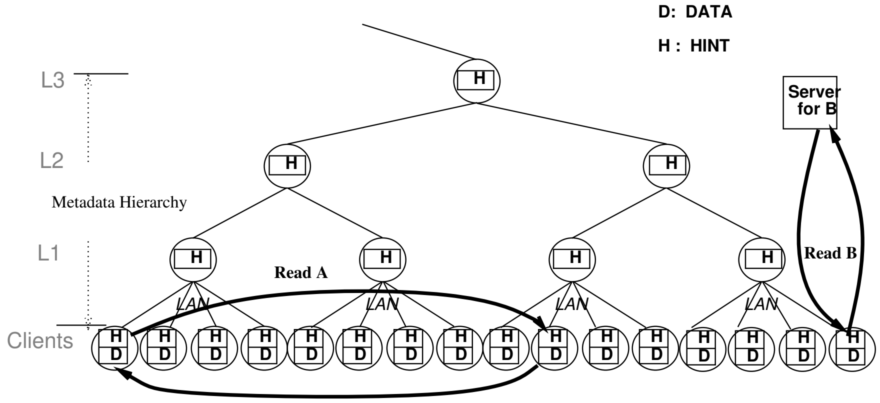
5.3 Overview of Hint Caches

In this section we examine a design that instantiates the principle of separation of data paths from metadata paths in order to implement two specific strategies. First, rather than traversing a hierarchy of data caches to locate cached data, the nodes maintain a directory of location-hints so they can send requests directly to caches with the data. The metadata hierarchy is used to maintain and propagate hints. Second, when a hit occurs at a remote cache, rather than sending data through a series of data caches in a hierarchy, the system uses cache-to-cache transfers to send the data directly from the cache supplying it to the cache that needs it. Thus, the data path consists of at most one cache-to-cache hop.

Figure 5.4-a illustrates our basic design, and Figure 5.4-b shows an alternate configuration. In the basic design, when a client cannot find data in its local cache, it sends the request to its closest proxy cache (an L1 proxy cache). The L1 proxy cache first tries to satisfy the request from data stored in its local cache. On an L1 miss, the proxy consults its local copy



(a) Basic hint hierarchy



(b) Alternate hint hierarchy

Figure 5.4: (a) Basic and (b) alternate hint-hierarchy design.

of the hint directory to find the nearest copy of the data. If the location-hint directory suggests a remote cache location for the item, the proxy sends the request to that cache, and that cache returns the data (the lines marked *Read A* in the figure.) If the hint directory does not know of a remote copy or if the hint turns out to be incorrect, the L1 proxy sends the request directly to the server indicated in the URL (the lines marked *Read B* in the figure.). In the basic design, the data always resides at the L1 proxy caches. The metadata hierarchy is only used to maintain and propagate location hints.

An alternative design based on the same principles is illustrated in Figure 5.4-b. In this design, the metadata hierarchy is extended beyond L1 caches to the clients. Each client uses its local location-hint directory to decide which cache, if any, to access. This design is able to consult the hint directory more quickly than the first design because the hints are local to the client, and direct remote cache access is even faster because the L1 cache is skipped. However, many large sites do not allow clients to access the Internet directly. Also, the client hint directory may be space-constrained and so may be less effective than the larger hint directory that may be economical at a proxy shared by multiple clients. Since the first configuration is most widely applicable, we focus on it for most of this section. For environments that can choose either design, we compare the trade-offs between the two at the end of this section.

Several other cache systems use metadata directories or multicast to locate data and then allow direct cache-to-cache data transfers. The primary differences among these schemes is how they structure their metadata directories. In local area networks, cooperative caching [26, 1] and global shared memory [40] hash a global directory across a set of nodes, and clients access the distributed global directory with network messages; in a LAN, the cost of this extra network hop is acceptable. In wide area networks, the WAN-xFS proposal [25] uses a hierarchy of metadata nodes to improve scalability, and the CRISP cache [46] uses a centralized global directory. Recently, the designers of CRISP have hypothesized that hashing the global directory for scalability or caching portions of this global directory at clients might be a useful addition to their design [45]. Several systems, including the Internet Cache Protocol (ICP) [124] and Zhang et. al's adaptive caching proposal [129], replace directories with multicast queries to nearby caches.

In contrast with these approaches, our system uses a scalable hierarchy of location hints combined with caching of these hints near clients. Four aspects of this design help it meet the design principles developed in the previous section. First, by caching location-hints near the clients, the system can quickly determine where, if anywhere, to find the needed data. This supports the principles of minimizing the number of hops to locate data and the principle of not slowing down misses. Second, by using a metadata hierarchy we localize updates of location-hints; this improves scalability, and it also speeds up the propagation of hints. Third, the system automatically maps the metadata hierarchy across the data nodes using a randomized hash function for scalability and fault tolerance. Fourth, the system uses small, fixed-sized records to store location-hint information. Consequently, this design reduces access time on a hit or miss, facilitates more widespread sharing of cached data, and reduces the overhead of distributing hints.

A potential additional benefit of separation of data and metadata is that it may reduce capacity misses by making better use of cache space than a cache hierarchy that caches the same data at multiple levels of the hierarchy. If peer-to-peer transfers are allowed, hierarchical double caching is not needed to facilitate data sharing, so it will often be a waste of disk space. Because we do not expect capacity misses to be a significant limit on performance, we do not investigate this benefit in detail. However, this aspect of the design could come into play in space-constrained systems.

5.3.1 Caching Location-hints Near Clients

A hint is an $(objectId, nodeId)$ pair that indicates the node that caches the nearest known copy of an object. Although our design specifies a metadata hierarchy for propagating hints, requests for hints are always satisfied locally by the leaf hint cache. By satisfying requests using only local information, hint caches observe the design principle of minimizing the number of cache hops to locate data. The system uses the hint hierarchy to propagate hints as a background task.

An advantage of maintaining hint caches rather than multicasting queries is that the propagation of hints happens independent of hint lookups. Conversely, multicast queries locate objects on demand by polling neighboring caches, potentially increasing latency. An additional advantage of maintaining hint caches is that a node with a hint cache can "query" virtually all of the nodes in a distributed system at once. This facilitates the design principle of maximizing sharing by allowing a cache to benefit from the contents of any other cache in the cooperative system. In contrast, multicast-based approaches generally limit sharing to a modest collection of nearby nodes in order to limit the overhead of responding to queries and to limit the time spent waiting to hear from all neighbors on cache misses. Multicast-based systems generally pursue more widespread sharing by breaking searches into several hops, each of which involves a multicast to a group of nodes nearer to the cache root or to the data server than the hop before. This effort to increase sharing, unfortunately, increases the number of hops on a hit or miss.

The downside of relying solely on hint caches is that they may sometimes contain incorrect information. In our system, hint errors can be suboptimal positives (they direct a request to a far away node when a closer one also has the data), false positives (they direct a request to a node that doesn't have the data), or false negatives (they say no node has the data when, in fact, one does). On a suboptimal positive, a node pays a higher cost to access data than it should have. On a false positive, the remote cache replies with an error code and the first node sends the request to the server specified in the URL. On a false negative, the node sends its request to the server specified in the URL. In both the false positive or false negative case we do not make another attempt to locate the data in the hierarchy. One could imagine a system that queries other directories in a hierarchy or that multicasts when a hint cache query fails to reveal a nearby cache, but we expect that when the hint cache fails, it is unlikely that a hit will result, so further time spent searching the cache is likely to be wasted. This design decision reflects the principle of not slowing down misses.

5.4 Scalable Metadata Hierarchy

The goal of the metadata distribution hierarchy and hint caches is to provide a fast and scalable way for a node to determine where to go for data that is not locally cached. This section examines four aspects of the design: propagating hints through a distribution hierarchy, storing hints as small, fixed-sized records, automatically configuring the hierarchy, and load balancing

Procedure PropagateHints(C, P)
On receiving new hint from child $j \in C$ (or self)
    if hint of first copy cached in parents subtree
        send hint to parent P
    if no child $i \in C$ had earlier sent hint for object
        send hint to all remaining children, $i \in C - \{j\}$
    else
        terminate propagation
    endif

On receiving hint from parent P
    if no child had earlier sent hint for object
        $\forall i \in C$ send hint to child, i
    else
        terminate propagation
    endif

Figure 5.5: Algorithm for Hint Propagation

across multiple hierarchies.

5.4.1 Propagating Location-hints

Our system propagates hint updates through a metadata hierarchy to reduce the amount of information sent globally. When a node in the metadata hierarchy learns about a new copy of data from a child (or the local data cache if it is the leaf), it propagates that information to its parent only if the new copy is the first copy stored in the subtree rooted at the parent. The node determines this by examining its local hint cache; if the parent had already informed the node of a copy in the parent's subtree, then the node terminates the update propagation. Similarly, when a node learns about a new copy of data from a parent, it propagates that knowledge to its children if none of its children had previously informed it of a copy. The details of the algorithm are given in Figure 5.5.

This protocol allows the hierarchy to limit the extent to which updates propagate through the tree. When some cache in the system references data cached nowhere else, that copy is the closest one to everyone in the system, so knowledge of it propagates to all the caches in the system. If a cache in the same subtree then loads the data into its cache, few other caches would regard that copy as significantly closer than the first copy, so it is not cost-effective to distribute information about that copy widely. On the other hand, if a far away cache loads the data, about half of the caches in the system would learn about this new, closer copy. The next copy might be of interest to a quarter of the caches, and so on.

The simple hierarchy reduces load significantly compared to the centralized directory scheme. To further improve scalability, both the centralized directory or the root could be distributed across multiple physical nodes. We describe how our system automatically splits all internal nodes in the tree in the next subsection.

One potential drawback of terminating updates is that when an object is first brought into the system, the accesses from many caches could overload the holder of that first copy. However, in our system, when later caches pull copies into their subtrees, their neighbors learn of the new copy and will soon decide that is a better place to go for that data. In essence, the system builds a distribution tree for popular data. Using a hierarchical scheme rather than a single, centralized directory for distributing hint information helps this happen quickly, since nodes in a subtree near a new copy of data will learn of the new copy quickly, while updates to more distant nodes may take longer. Thus, hint caches trade potentially worse hit rates for better hit times or miss times. Two factors affect the scalability and hit rates: (i) the capacity of the hint cache, (ii) the bandwidth for update propagation, and (iii) the freshness of the hints. To limit the amount of space used for storing hints, they are stored as small fixed size records. Since the hints store only one entry per unique object (consisting of the closest location), the hint cache size does not grow with the number of caches in the system. Since the metadata hierarchy is used to prune the update propagation and the hint update messages are small in size (around 20 bytes), the bandwidth used for hint propagation is negligible. We quantify these results in the experimental evaluation in the following chapter. Another possible limitation of hints is that they may be stale. Updates to the hints will take some time to propagate through the system, during which time caches may make suboptimal decisions about where to send their requests. To facilitate this, our system uses a metadata hierarchy that preserves locality for hint updates, it uses a scalable hierarchy to avoid bottlenecks, and it uses small hint records to reduce the network overheads of propagation.

5.4.2 Self-configuring the Metadata Hierarchy

In order to scale to large systems of caches, the hint distribution hierarchy is self-configuring. In particular, our system constructs a hierarchy using an algorithm developed by Plaxton et. al [97] to embed virtual hint distribution trees onto the nodes in the system. This algorithm has the following properties:

- *Automatic configuration.* Given a list of nodes and the approximate distances between them, the system builds trees automatically without manual specification of the parent/child relationships or manual assignment of objects to subtrees.
- *Fault tolerance and automatic reconfiguration.* As nodes enter or leave the system, the algorithm automatically reassigns children to new parents. This reassignment disturbs very little of the previous configuration.
- *Load distribution.* Different objects use different virtual trees depending on their object IDs. For example, if there are n nodes in the system, each node will be the root for roughly $1/n$ of the objects.
- *Locality.* Near the leaves of the virtual trees, the distance between parents and children tends to be small; near the roots, this distance is generally larger.

The algorithm works as follows. We assign each node in the system a pseudo-random ID (the MD5 signature of the node's IP address), and we assign each object in the system a pseudo-random ID (the MD5 signature of the object's URL.) Conceptually, the algorithm builds a tree for each object in which the IDs of nodes at high levels of the tree match the object's ID in many low-order bits, while the IDs of nodes at low levels of the tree match the object's ID in few low-order bits. The root node for an object is the node whose ID matches the object's ID in the most low-order bits.

Figure 5.6 illustrates the algorithm for a simple binary tree. The algorithm first embeds a set of virtual trees across the nodes as part (a) of the figure illustrates. In this setup phase of the algorithm, to construct level i of the hierarchy, each node finds two level $(i + 1)$ parents.

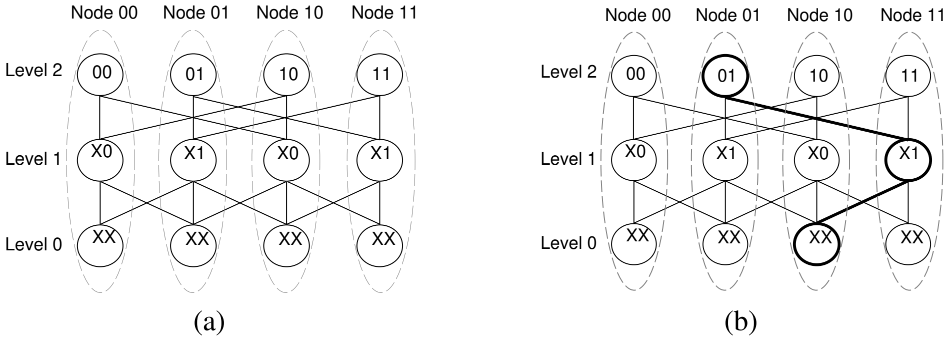


Figure 5.6: Plaxton et. al's randomized scalable tree algorithm for a binary tree. In part (b) of the figure, the hierarchy traversed by an object with ID $xxxxxxx01$ starting from node 10 is highlighted in bold.

Both parents' IDs must match that node's bottom i bits, but one differs in the $(i+1)$th bit and the other matches in that bit (e.g., the second parent is the node, itself.) For locality, nodes choose the nearest eligible level $(i+1)$ parents. For low levels of the tree, only a few bits need to match, so parents tend to be quite nearby. At higher levels, a node often has to settle for a parent that is farther away. Note that there may some maximum value for i above which no valid level $(i+1)$ parents exist. When a node is removed (or added), its children at various levels select new parents. When a node is removed, of course, the new parents will generally be farther away than the old ones were.

An update of the metadata status for an object begins at the node where the status changes. At level i, a metadata node sends the update to the parent whose $(i+1)$th bit matches the object ID's $(i+1)$th bit. Part (b) of the figure shows the path that the system would use to notify a root hint cache that node 10 had just accessed an object with id $xxxxxxx01$. In our system, to accommodate flatter hierarchies with k-ary trees, nodes match $log(k)$ bits at a time to choose from among k parents.

5.5 Push Caching

The hint directory and direct forwarding algorithm described above does a good job at providing fast access to data stored in a distributed cache. However, from Figure 5.3 we note that a significant number of hits in the system are to distant caches and Figure 5.1 and Table 5.3 suggest that accessing these caches can be slow even with the direct-access facility provided by hints. For example, the data in Section 5.2.1 suggest that for the testbed hierarchy L1 cache accesses for 8KB objects are 4.75 times faster than direct accesses to caches that are as far away as L2 caches and 6.17 times faster than direct access to caches that are as far away as L3 caches. The data also suggest that the penalty for accessing distant caches is modest when Internet load is low (e.g., the *Total Direct Min* numbers in Table 5.3), but the penalty is high when the network is congested (e.g., the *Total Direct Max* numbers).

In this section we explore using push based algorithms to pursue the design principle: *cache data close to clients.* Unlike traditional caching schemes that cache data on demand, push caching anticipates future demands and sends objects to caches that have not requested them. We examine two simple algorithms that attempt to make controlled numbers of extra copies of data and distribute those copies through the system in order to replace hits to distant caches with hits to closer ones.

5.5.1 Push Algorithms

Push algorithms attempt to balance the overhead of making and storing extra replicas with the benefit of reducing access times when the system can access data from a closer replica than before. For systems that cannot predict future accesses with certainty, push algorithms face a dilemma — creating more replicas incurs high costs but makes it more likely that a replica will be near where it needs to be.

We consider four design criteria for the push algorithms examined here. First, the algorithms should strive for efficiency. An algorithm is efficient if a large fraction of the replicas it creates actually succeed in improving system response time. Second, the algorithms should provide significant performance gains; an efficient algorithm that only pushes a small number of objects during the course of the trace (but which succeeds in slightly improving performance with each push) would not be worth pursuing. Third, the algorithm should be adaptive; if the algorithm is running in a system that is resource-rich, it should aggressively replicate data to improve response time even at the cost of efficiency. Conversely, if the system is resource-poor, it should be more conservative. Fourth, the algorithm should be simple both to ease implementation and to avoid consuming too many resources figuring out what to push; with potentially millions of objects in a cache, the push system cannot spend too much effort tracing the popularity of or making decisions about any one of those objects.

The class algorithm we consider observes two important restrictions.

1. We assume that the algorithms are not supplied with knowledge of future access patterns. They must therefore predict future access patterns based on the past. In particular, we do not assume any external directives about future accesses such as hoard lists [72, 74] or server hits [91].

2. We limit pushing or prefetching to increasing the number of copies of data that are already stored at least once in the cache system. Thus, our algorithms can only affect the number of L1, L2, and L3 hits, not the number of system-wide misses. Notice that more aggressive prefetching algorithms than the ones we examine could could fetch objects that are not cached anywhere in the cache hierarchy by accessing the original servers. For example, one could imagine having the cache hierarchy "crawl" the Internet in the background, looking for new pages. Clearly such an algorithm could further improve performance by reducing the number of complete misses endured by the system. However, in this study we restrict ourselves to algorithms that do not increase the load on servers outside of the cache system.

We examine two push algorithms and compare them to a base case and best case.

5.5.1.1 Base-case and Best-case

For reference, we compare our results to two extreme cases. In the base case, the system only replicates data on demand and in response to client accesses to the data. Our first base case algorithm is the standard data hierarchy algorithm, which creates replicas of data along the hierarchy path between the client and the level of the cache that supplies the data (e.g., on a L3 hit, the hierarchy creates new L2 and L1 replicas). Our second base case algorithm is the hint hierarchy algorithm described in Section 5.3.

We also compare our results to a best case. In the best case algorithm, we imagine that whenever an object is fetched into the cache system, it appears not only in the cache that referenced the data but also in all caches that will reference it in the future. We further assume that these extra copies do not consume additional disk space at the extra caches, so the caches' local hit rates are not damaged by this aggressive replication. This scenario replaces all L2 and L3 hits with L1 hits and does not change the number of misses. Our best case algorithm represents the best performance achievable by a push algorithm that meets the two restrictions mentioned above. Note that a system with large amounts of spare bandwidth and disk space could approach this level of performance by broadcasting all data fetched on misses to all caches. We seek more efficient algorithms that can work well when disk space or network bandwidth or both are constraints.

5.5.1.2 Push Updates

We first consider a simple, efficient algorithm that approximates update-based cache consistency. This algorithm is based on the observation that when an object is modified, a good list of candidates to reference the new version of the object is the list of caches that previously cached the old version of the object. Thus, when the cache system fetches an object due to a communication miss (see Figure 5.2), it sends copies of that object to caches that were storing the previous version of that object.

For cache systems that use invalidate-based cache consistency, caches track objects that they recently invalidated from their cache, and when they receive hints about locations of objects on that list, they send read requests to the hinted location. For cache systems that use polling-based consistency, the hints should be augmented with the objects' version numbers or last-modified-times. In that case, if a cache hears the location of a new version of an object it is caching, it should fetch that object.

This algorithm can adapt to limit the resources consumed by hint propagation in two ways. First, whenever a cache updates an object, the cache "ages" the object by moving it down the LRU list. Thus, objects that are updated many times without being read will be evicted from the cache. In resource-rich configurations, this aging will be slow, while in resource-poor

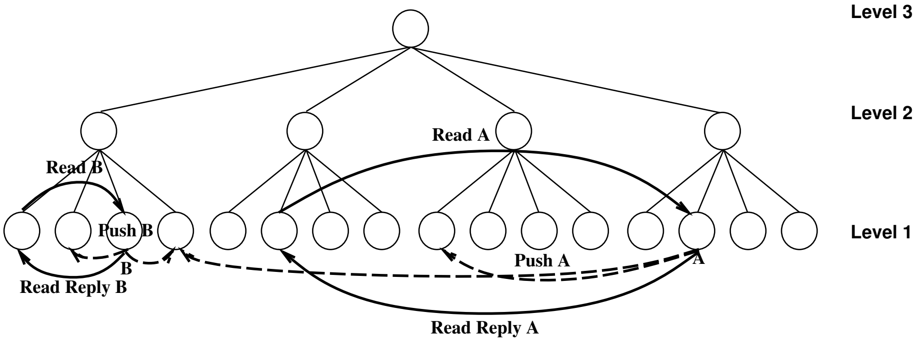


Figure 5.7: Hierarchical push algorithm for 3-level hierarchy. Once two level-2 subtrees fetch object A, the algorithm pushes object A to all level-2 subtrees. Once two level-1 subtrees under a level-2 subtree fetch object B, the algorithm pushes object B to all level-1 nodes under that level-2 parent.

systems it will be fast. Second, caches place an upper limit on the update-fetch bandwidth they will consume and discard update-fetch requests that exceed that rate.

With respect to the four design criteria discussed above, we note that this algorithm restricts itself to looking at a particular situation where it can be relatively confident that the copies of the object it makes will be useful. This allows the algorithm to achieve good efficiency at a cost of limiting the maximum performance it can achieve. The algorithm is somewhat adaptive in that it can be configured to use more or fewer resources by changing the aging rate depending on the environment. Finally, the algorithm is simple — it makes decisions based on information already present in the system, and it uses events already present in the system to notify caches when to consider making copies of particular objects.

5.5.1.3 Hierarchical Push-shared

The hierarchical push-shared algorithm is more general and less selective than the push updates protocol. Because the algorithm considers more objects as candidates for replication it is less efficient than the previous algorithm, but it can achieve more significant performance gains.

As Figure 5.7 illustrates, when a cache fetches an object from a cousin for which a level-l parent is the least common ancestor in the metadata hierarchy, the cache supplying the object also pushes the object to a random node in each of the level-$(l-1)$ subtrees that share the

Algorithm Push-shared
For each node A
  on receiving request for object from node B
    find C the least common ancestor of A, B.
    l = level of C in metadata hierarchy
    for each level $l-1$ subtree
      select a random node, j, in the subtree
      push object to node j
    end for
  end

Figure 5.8: Algorithm for hierarchical push-shared

level-l parent. The algorithm is described in Figure 5.8. For example, in a three-level metadata hierarchy when a cache fetches data from a distant cache that shares a level-3 parent, the system creates one copy of that object in each of the level-2 hierarchies. Thus, the next access to the data will cost at most a level-2 access. Similarly, on a level-2 hit, the algorithm pushes the object to *all* level-1 caches that share the same level-2 parent.

The intuition behind this algorithm is that if two subtrees in a hierarchy access an item, it is likely that many subtrees in that hierarchy will access that item. Notice that although this algorithm is simple and does not explicitly track object popularity, it results in the desired effect that popular items are more widely replicated than unpopular items. Also note that if there is locality within subtrees, items popular in one subtree but not another will be more widely replicated in the subtree where the item is popular.

With respect to the four design criteria discussed above, this algorithm is less specialized than the previous algorithm. This allows it to replicate objects that the previous algorithm cannot consider. By considering a wider range of candidates for replication, this algorithm may be less efficient, but it may also provide better latency improvements. The algorithm can be adapted to resource rich environments (push to k nodes in a subtree rather than just 1 node in a subtree) or resource poor environments (push a copy to $\frac{1}{j}$ of the eligible subtrees). We will examine three configurations: push-1 sends a copy to 1 node in each eligible subtree, push-half sends a copy to half of the nodes in each eligible subtree, and push-all sends a copy to all nodes in an eligible subtree. Finally, the algorithm is simple — it makes decisions based on information already present in the system; also, it uses events already present in the system to notify caches to consider whether to make copies of particular objects.

5.6 Related work

In Section 5.3, we discussed a number of cache systems that separate data from metadata using metadata hierarchies, directories, or multicast. This section considers other areas of work relevant to understanding our results.

Several studies have examined Internet workloads in depth with the goal of understanding how to improve performance. Arlitt and Williamson [2] examined basic workload characteristics for six servers. Duska et. al [33] examined workloads for several 3-week traces, Gribble and Bewer [54] examined a large trace of Berkeley HomeIP users. These studies support the conclusion that cache architectures that scale are important because increasing the number of users sharing a cache system increases the hit rates achievable by that system. Rousskov [101] and Maltzahn et. al [81] measure the performance of cache servers deployed in the field and examine why the performance of these servers has been worse than laboratory tests predicted [19]. These studies suggest that long network round trip latencies may be a significant factor.

Hierarchical caching has been examined in the context of file systems [8, 104, 52]. Muntz and Honeyman [84] concluded that the additional hops in such a system often more than offset improvements in hit rate and characterized the extra level of cache as a "delay server." We reach similar conclusions in the context of Internet caching, leading to our design principle of minimizing the number of hops on a hit or miss.

Hierarchical caching has been more widely accepted on the Internet, and it is common in areas with low-speed connections to the Internet [85, 108]. In addition, many sites add proxy caching to their corporate firewalls. A widely deployed and studied system has been the publicly-available Harvest cache [19] and its successor, the Squid cache [123]. A key motivation of our design is that we disagree with the conclusion that "cache hierarchies do not significantly reduce cache performance on cache misses" [19]. The evidence suggests to us that for large scale systems with long network delays or busy servers or both, reducing the number of hops is vital to performance.

Several researchers have proposed improving the scalability of a data hierarchy by splitting responsibilities according to a hash function [116]. This approach may work well for distributing load across a set of caches that are near one another and near their clients, but in larger systems where clients are closer to some caches than others, the hash function will prevent the system from exploiting locality.

Several studies have examined push caching and prefetching in the context of web workloads [55, 57, 91]. These systems all used more elaborate history information to predict future references than the algorithm we examine. Because large, shared caches do a good job at satisfying references to popular objects, we explore prefetching strategies that will work well for the remaining large number of objects about whose access patterns little is known. Kroeger et. al [73] examined the limits of performance for caching and prefetching. They found that the rate of change of data and the rate of accesses to new data and new servers limits achievable performance.

5.7 Conclusions

Although caching is increasingly used in the Internet to reduce network traffic and the load on web servers, it has been less successful in reducing response time observed by clients. We examine several environments and workloads and conclude that this may be because traditional hierarchical caches violate several basic design principles for distributed caching on the Internet. We examine specific strategies based on these principles. First, we develop the hint cache architecture to minimize the number of hops on a hit or miss, support widespread sharing, and not slow down the misses. Second, we build a scalable metadata distribution hierarchy that avoids central bottlenecks, scales to a large number caches and self-configures dynamically. Third, we design push caching algorithms that dynamically move the data closer to the clients. In the following chapter, we quantify the performance improvements using these strategies and describe a prototype implementation.

Chapter 6

Experimental Evaluation of the Caching Architecture

> *It doesn't matter how beautiful your theory is, it doesn't matter how smart you are — if it doesn't agree with experiment, it's wrong.*
>
> *—Richard P. Feynman*

In this chapter we evaluate the performance of hint caches, the scalability of the metadata hierarchy, and the performance gain by push caching. We evaluate our design strategies using simulation studies of three large workloads that contain accesses by users of Digital Corporation's proxy server [32], UC Berkeley's Home-IP service [54], and Prodigy ISP's dial-up web browsing service. Our algorithms focus on *improving hit and miss response times* rather than trying to improve global hit and miss rates. Thus, we parameterize our results using estimates of Internet access times based on our own measurements and those in the literature [101]. For these traces and network configurations, we find that our hint architecture provides speedups of 1.27 to 2.30 compared to a standard, three-level cache hierarchy and that push caching improves performance by up to an additional factor of 1.25. Overall, our techniques provide speedups of 1.27 to 2.43 compared to a traditional hierarchy. We present our prototype implementation that augments the widely-deployed Squid proxy cache [123]. It implements a self-configuring hint distribution hierarchy, local hint caches, and cache-to-cache data transfers. The push-shared

Trace	# of Clients	# of Accesses (millions)	# of URLs (millions)	Dates	Client ID
DEC [32]	16,660	22.1	4.15	Sep. 1 – Sep. 21, 1996	preserved
Berkeley [53]	8,372	8.8	1.8	Nov. 1 – Nov. 19, 1996	preserved
Prodigy	35,354	4.2	1.2	Jan. 13 – Jan 15, 1998	dynamic IP

Table 6.1: Characteristics of trace workloads.

cache algorithm is implemented by a simple extension to the metadata distribution hierarchy.

The rest of this chapter is structured as follows. Section 6.2 compares the performance of hint caches with a hierarchical Squid cache. In Section 6.3 we evaluate the scalability of hint caches by considering the disk space and network bandwidth overheads. Section 6.4 evaluates the performance of different push caching algorithms. Section 6.5 describes our prototype implementation. Finally, Section 6.6 summarizes the results.

6.1 Methodology

The simulation experiments in this study use three large-scale traces taken at proxies serving thousands of clients. Key parameters of these traces are summarized in Table 6.1. All three traces track the activities of thousands of clients over several days. For the DEC and Berkeley traces, each client has a unique ID throughout the trace; for the Prodigy trace, clients are dynamically bound to IDs when they log onto the system. In analyzing the cache behavior of these traces, we use the first two days of each trace to warm our caches before beginning to gather statistics. Due to time constraints and memory limitations of our simulation machines, the results in Section 5.5 use only the first seven days of the DEC and Berkeley traces.

To determine when objects are modified and should thus no longer be provided from the cache, we use the last-modified-time information provided in the DEC traces. When the last modified times are not provided, we infer modifications from document sizes and return values to if-modified-since requests. Both of these strategies will miss some of the modifications in these traces.

Current web cache implementations generally provide weak cache consistency via ad hoc consistency algorithms. For example, current Squid caches discard any data older than two days. In our simulations, we assume that the system approximates strong cache consistency by invalidating all cached copies whenever data changes. We do this for two reasons. First, techniques for approximating or providing strong cache consistency in this environment are improving [58, 80, 127], so we expect this assumption to be a good reflection of achievable future cache technology. Second, weak cache consistency distorts cache performance either by increasing apparent hit rates by counting "hits" to stale data or by reducing apparent hit rates by discarding perfectly good data from caches. In either case, this would add a potentially significant source of "noise" to our results.

These traces have two primary limitations that will affect our results. First, although we use traces with thousands of clients, the population of clients in the traces is still a small fraction of the web population. Gribble and Brewer's analysis of the Berkeley trace indicates that achievable hit rates will improve as more clients are included in a cache system, but they caution that it is difficult to know at what point beyond the population sizes studied that such a trend will stop [54]. Duska et. al conclude that as the rate of requests a cache system handles increases, its achievable hit rate will also increase [33].

The second limitation of these traces is that they are gathered at proxies rather than at clients. Thus all of these traces will display less locality and lower total hit rates than would be seen by clients using such a system. Thus, our results should be interpreted as the hit rate for requests that miss in clients' local caches or the response time for local misses; the total hit rates and response times observed by clients will generally be better than reported here.

6.2 Performance of Hint Caches

Figure 6.1 shows the simulated performance for the DEC, Berkeley, and Prodigy traces under a series of algorithms both to quantify the overall performance of the system and examine the impact of assuming a proxy-based implementation. The figure provides two baselines for performance. It includes a traditional 3-level hierarchy and a *best case* L1 cache that is shared by all clients in the system, but that is as close to all clients as the smaller L1 caches used in the standard configuration.

Traces	Max	Min	Testbed
Prodigy	1.80	1.38	2.31
Berkeley	1.79	1.32	2.79
DEC	1.62	1.28	1.99

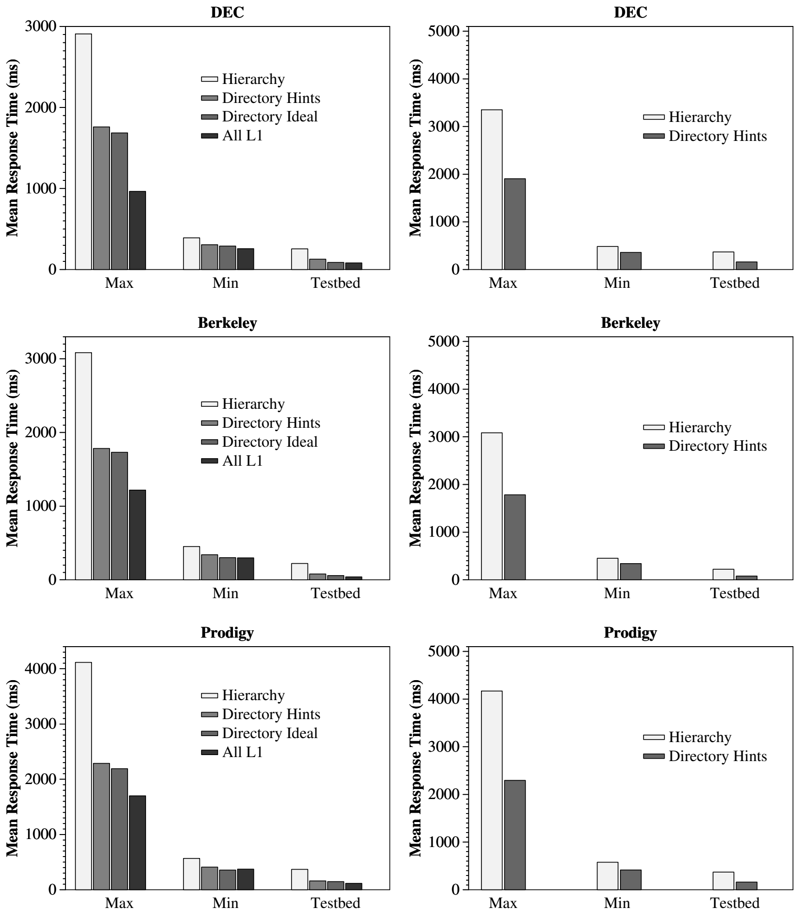
Table 6.2: Performance of DEC, Berkeley, Prodigy traces measured as a ratio of the response time for the traditional data hierarchy to that of the hint directory architecture for the three access times shown in Figure 6.1

First, note that our system configuration is designed to let us explore how different architectures balance sharing, locality, and scaling. As the bars for the *best case* L1 cache show, if the underlying physical network and cache machines allow sharing to be increased without increasing cache access delays, the best configuration is a large, shared cache.

Direct access (and hint caches) significantly outperform the traditional 3-level hierarchies with speedups ranging from 1.3 to 2.9. Notice that these improvements in response time come not from improving the global hit rate compared to a standard hierarchy but from improving hit times and miss times. The additional network hop for the proxy-based hint cache configuration hurts performance modestly compared to the ideal system. The realistic implementation achieves speedups of 1.3 to 2.3. For this set of network topologies and workloads, another reasonable alternative is for each group of 256 clients to share a L1 caches but not to share caches further. Such an approach falls short of the ideal direct access protocol by as much as a factor of 1.53 and falls short of the hint cache protocol by as much as a factor of 1.3.

The speedup of the hint cache system relative to a single shared L1 cache, depends on the degree of sharing at the L1 cache. If a large fraction of the total hits happen at the L1 level, the performance improvement by using hint caches will not be substantial. However, if the hit ratios at the L1 level are not a significant fraction of the total, hint caches achieve much higher performance gains due to the increased sharing and efficient access. Figure 6.2 shows the variation in performance for different implementations. It compares a single L1 cache with hint caches and a 3-level hierarchy.

We also examined the trade-offs between the hint hierarchy configurations shown in



(a) Infinite disk (b) Space constrained

Figure 6.1: Simulated performance for DEC, Berkeley, and Prodigy traces. The three groups of bars for each trace show the performance when the access times are parameterized by the *Testbed* times shown in Figures 5.1 or the *Min* and *Max* access times measured by Rousskov shown in the *Total Hierarchical* and *Total via L1* columns of Table 5.3. In figure (a) each node has infinite disk space for caching and hints. We show performance for a 3-level hierarchy, idealized forwarding, hint caches, and an idealized cache shared by all clients in the system and as close to each as their normal L1 cache. In figure (b) each node in the traditional hierarchy has 5 GB of disk used for caching objects, while each L1 node in the hint hierarchy system has 4.5 GB of disk for caching objects and each L1, L2, or L3 node in the hint hierarchy has 500 MB of disk for storing hints (notice that this arrangement gives more space to the standard hierarchy).

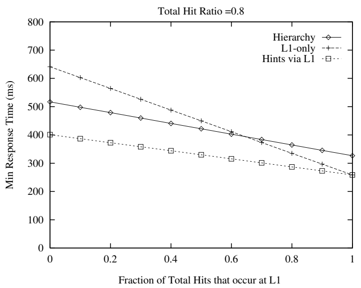


Figure 6.2: The minimum response time for the 3-level hierarchy, a single L1 cache and the hint caches. The graphs show the performance while varying the fraction of the total hits that occur at the L1 level. The figure assumes a total hit ratio of 0.8 and the latency numbers are parameterized from Table 5.3.

Figure 5.4. In summary, for installations that allow clients to access distant caches directly, the choice of which architecture to use comes down to a trade-off between two design principles. On one hand, the (potentially) larger L1-proxy hint cache can index a larger number of remote cache entries, and it thus may facilitate the principle of widespread sharing better than the smaller client hint cache. On the other hand, accessing the client hint cache is faster than accessing the proxy hint cache, so the second configuration better fulfills the design principles of minimizing the number of hops on a hit or miss and of not slowing down misses; this advantage may be reduced if a system requires clients to send all requests through a proxy for security reasons. For the testbed parameters and the DEC trace, as long as client caches are large enough so that the false-negative rate for the client hint caches is below 50%, the alternate configuration is superior. At best, if client hint caches are large enough so that they match the hit rate of proxy hint caches, they improve response time by about a 20% compared to proxy hint caches for the testbed hierarchy and the DEC trace.

6.3 Scalability of Hint Caches

To evaluate the scalability of hint caches we consider: (i) the disk space overhead for hint cache, (ii) the freshness of the hints, and (iii) the network bandwidth overhead for propagation.

First, a node's hint cache will only be effective if it can index significantly more data objects than the node can store locally, and it will observe the design principle of maximizing

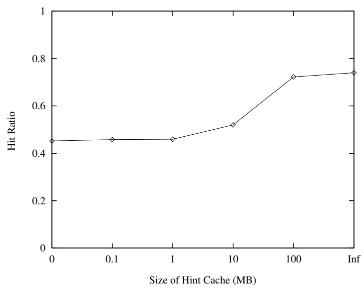


Figure 6.3: Hit rate assuming that groups of 256 clients from the DEC trace each access an infinite proxy cache and that each proxy cache can access other proxy caches via a hint cache of the specified size. Each entry in the hint cache takes 16 bytes and they are stored in a 4-way set associative array with total size specified in MB on the x-axis.

sharing if it can index most or all of the data stored by the caches in the cache system. As we describe in detail in the prototype implementation (Section 6.5.1), our system compresses hints to 16-byte, fixed-sized records. At this size, each hint is almost three orders of magnitude smaller than an average 10 KB data object stored in a cache [2]. Thus, if a cache dedicates 10% of its capacity for hint storage, its hint cache will index about two orders of magnitude more data than it can store locally. Even if there were no overlap of what different caches store, such a directory would allow a node to directly access the content of about 63 nearby caches. Because the hint cache need only store one entry when an object is stored in multiple remote caches, coverage should be much broader in practice. Another way of viewing capacity is to consider the reach of a 500 MB index (10% of a modest, 5 GB proxy cache). Such an index could track the location of over 30 million unique objects stored in a cache system.

Figure 6.3 shows how the size of the hint cache affects overall performance in the DEC trace. Very small hint caches provide little improvement because they index little more than what is stored locally. For this workload, hint caches smaller than 10 MB provide little additional "reach" beyond what is already cached locally, but a 100 MB hint cache can track almost all data in the system.

Not only can the hints be stored in a reasonable amount of space, maintaining them consumes little network bandwidth. For example, for the DEC trace, the most heavily-loaded hint cache sees an average of 1.9 hint updates per second. As Section 6.5 describes, each

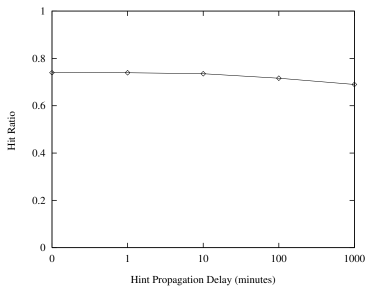


Figure 6.4: Hit rate assuming that groups of 256 clients from the DEC trace each access an infinite proxy cache and that each proxy cache can access other proxy caches via a hint cache that is updated the specified number of minutes after objects appear or disappear from caches.

hint update is 20 bytes, so maintaining this hint cache consumes only 38 bytes per second of bandwidth. Certainly this degree of bandwidth consumption poses no difficulty for T1 or better-connected sites; in fact, it is only about 1% of the bandwidth of a 33.6Kbit/s modem. Of course, peak bandwidths may be much higher than average bandwidths, but it appears that even a modestly-well connected host will be able to handle hint updates with little effort.

A second possible limitation of hints is that they may be stale. Updates to the hints will take some time to propagate through the system, during which time caches may make suboptimal decisions about where to send their requests. Figure 6.4 quantifies the dependence of global hit rate on the amount of time it takes to update hints in the system. In this experiment, we assume that whenever an object is dropped from a cache to make space for another object or an object is added to a cache, none of the hint caches learn of the change for an amount of time specified on the x-axis in the figure. This experiment suggests that the performance of hint caches will be good as long as updates can be propagated through the system within a few minutes. To facilitate this, our system uses a metadata hierarchy that preserves locality for hint updates, it uses a scalable hierarchy to avoid bottlenecks, and it uses small hint records to reduce the network overheads of propagation.

6.3.1 Hint Propagation Overhead

Our system propagates hint updates through a metadata hierarchy to reduce the amount of information sent globally. The protocol allows the hierarchy to limit the extent to which updates

Organization	Average	Peak	99%	90%
Centralized directory	4.8	14.4	10.2	7.4
Hierarchy (root)	2.1	9.8	6.2	4.0
Plaxton hierarchy (max-node)	0.15	2.4	1.1	0.6

Table 6.3: Number of location-hint updates sent to the root during the first seven days of the DEC trace. The columns show the average, peak, and 99th and 90th percentiles of requests per second over 10-second periods. All cache updates are sent to the centralized directory; the hierarchy line shows load when the hint hierarchy prunes requests but the system still uses a single root; the Plaxton hierarchy line shows load on the most heavily loaded 64 nodes, which acts as a root for about 1/64th of the objects and also acts at different levels of the hierarchy for other objects.

propagate through the tree. When some cache in the system references data cached nowhere else, that copy is the closest one to everyone in the system, so knowledge of it propagates to all the caches in the system. If a cache in the same subtree then loads the data into its cache, few other caches would regard that copy as significantly closer than the first copy, so it is not cost-effective to distribute information about that copy widely. On the other hand, if a far away cache loads the data, about half of the caches in the system would learn about this new, closer copy. The next copy might be of interest to a quarter of the caches, and so on.

Table 6.3 examines how effective the metadata hierarchy is at filtering traffic by comparing the number of updates seen by the root under the DEC trace for three configurations: a centralized directory which receives all updates, a single hierarchy that filters updates to reduce load, multiple hierarchies with load balancing using the Plaxton algorithm. This experiment suggests that the filtering function of the simple hierarchy reduces load significantly compared to the centralized directory scheme. To further improve scalability, both the centralized directory or the root could be distributed across multiple physical nodes.

One potential drawback of terminating updates is that when an object is first brought into the system, the accesses from many caches could overload the holder of that first copy. However, in our system, when later caches pull copies into their subtrees, their neighbors learn of the new copy and will soon decide that is a better place to go for that data. In essence, the system builds a distribution tree for popular data. Using a hierarchical scheme rather than a single, centralized directory for distributing hint information helps this happen quickly, since nodes in a subtree near a new copy of data will learn of the new copy quickly, while updates to more distant nodes may take longer.

6.4 Evaluation of Push Algorithms

We use our simulator to evaluate the push-update and push-shared algorithms. The base cases correspond to the standard hierarchy and our hint algorithm from the previous section. The best case corresponds to a system that uses our hint algorithm, but for which all L3 and L2 hits are replaced by L1 hits. We then examine both the update push and the hierarchical miss push algorithms.

Although we do not expect capacity misses to be a major factor in understanding systems that replicate data on-demand, in push based systems, speculative replication may displace more valuable data from caches. To monitor this effect in our simulations, we report results for the space-constrained configuration in which each of the 64 L1 caches has 5 GB of capacity. For the best case simulations, we also limit the local cache to 5 GB, but we do not charge the caches any space for objects that have been pushed to them but not referenced by them.

In these simulations, we look at two figures of merit. First, we examine how much the push algorithm reduces system response time by replacing accesses to distant caches with accesses to nearby ones. Second, we examine efficiency by examining what fraction of data that is pushed is later used.

Figure 6.5 shows the simulated response time for the DEC trace under a range of push options. This experiment suggests that an ideal push algorithm could achieve speedups of 1.54 to 2.63 compared to the no-push data hierarchy, and speedups of 1.21 to 1.62 compared to the no-push hint hierarchy; the largest speedups come when the cost of accessing remote data is high such as the Max value in Rousskov's measurements (see Table 5.3). The hierarchical push algorithms described here achieves speedups of 1.42 to 2.03 compared to the no-push data hierarchy, and speedups of 1.12 to 1.25 compared to the no-push hint hierarchy. Update push does not achieve any appreciable performance improvement compared to the no-push,

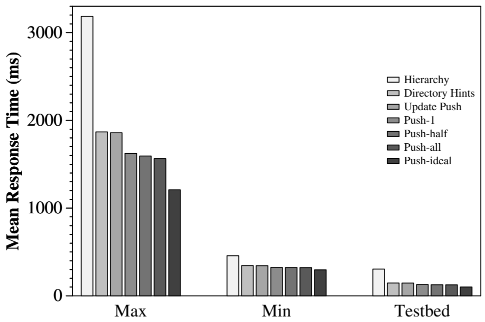


Figure 6.5: Simulated response time for DEC trace workload for six algorithms: no push (data hierarchy), no push (hint hierarchy), update push, push-1 (send a copy to 1 node in each eligible subtree), push-half (send a copy to half of the nodes in each eligible subtree), and push-all (send a copy to all nodes in an eligible subtree).

hint-hierarchy case.

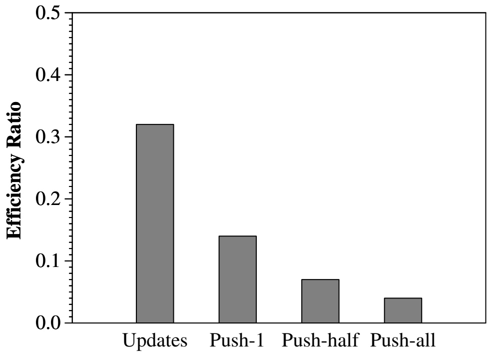
Figure 6.6 shows the efficiency of the algorithms. This experiment suggests that the update push algorithm is efficient: one-third of the data it pushes are used. The hierarchical algorithms have efficiencies between 13% and 4% and they increase the bandwidth consumed by up to a factor of four compared to the demand-only case. This may be an acceptable trade-off of bandwidth for latency in many environments.

6.5 Prototype Implementation

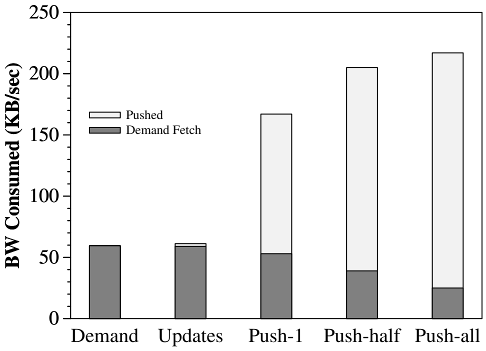
Our prototype implements hint caching, push caching [71], and automatic configuration. It is based on Squid 1.1.20[123] and named *Cuttlefish*.

There are three primary interface commands between Cuttlefish and the hint cache. One is an *inform* command, which tells the hint cache that a copy of a given object is now stored locally, and the hint cache should advertise it. The second is an *invalidate*, which tells the hint cache that that copy is no longer present; this causes the hint cache to advertise the non-presence, and to use the next best location (if one is known) of that object. The third is *find nearest*, which asks the hint cache to report the location of the nearest copy of an object, if one is known.

In addition, the hint module uses Squid's internal communications interfaces to prop-



(a)



(b)

Figure 6.6: (a) Efficiency and (b) bandwidth of push algorithms for the DEC trace. For the efficiency graph, each bar shows the fraction of all bytes pushed that are later accessed. The bandwidth graph shows the bandwidth consumed by pushed data and data fetched on demand for the algorithms.

agate hint information. The system sends hint updates to neighboring caches as HTTP POST requests to the "route://updates" URL. Support for sending and receiving POSTs has been added to Squid in order to support both this facility and a best-effort object consistency protocol. Periodically, each cache POSTs to its neighbor a message containing the HTTP headers and the batch of all updates that the cache has seen in the most recent period; each update consumes 20 bytes: a 4-byte action, an 8-byte object identifier (part of the MD5 signature of the object's URL), and a 8-byte machine identifier (an IP address and port number.) Nodes randomly choose the period between updates using a uniform distribution between 0 and 60 seconds to avoid the routing protocol capture effects observed by Floyd and Jacobson [41].

It would be desirable if these updates were multicast efficiently to exactly the interested parties. However, the amount of data involved seems overly small to warrant the complexities of reliable multicast, and the interval until the next transmission concerning the object is too large to use raw UDP multicast. Thus, it seems wise to start by simply using TCP for data exchange. If an efficient inter-cache reliable multicast facility becomes available, it would be a useful optimization to our protocol.

For self-configuration, our implementation consists of three layers. The *membership layer* tracks which nodes are participating in the cache system via an algorithm that floods

"Join" and "Leave" messages through the system. The membership layer also implements a security policy to restrict who may join a cooperative cache system. The *connectivity layer* at each node tracks: (1) its distance—measured in hop count and round-trip latency—to other nodes and (2) the nodes that are alive at any given time. The *hierarchy layer* uses the information provided by the lower layers to produce the system's metadata hierarchy. We adapt Plaxton's algorithm [97] to produce a set of hierarchies that (1) balance load by making different nodes act as roots (or high-level internal nodes) for different objects, and (2) maintain locality so that parents and children near the leaves of the tree are likely to be near one another, with the distance between parents and children increasing at higher levels of the tree. Although self-configuration represents one of the more complex aspects of our implementation, our experience has been that it is a convenient facility even for the small systems we have deployed for production use.

6.5.1 Data Structures and Hint Records

An important implementation detail in our prototype is our decision to use small, fixed-sized records and to store hints in a simple array managed as a k-way associative cache. In particular, our design stores a node's hint cache in a memory mapped file consisting of an array of small, fixed-sized entries. Each entry consumes 16 bytes: an 8-byte hash of a URL and an 8-byte machine identifier.[1]

We store a URL hash rather than a complete URL to save space in the table and to ensure that entries are of fixed size. Small records improve performance by allowing a node to store hint entries for a large number of objects; this facilitates the principle of maximizing sharing. Small records also reduce the network cost of propagating hints, and they allow a larger fraction of the hint table to be cached in a given amount of physical memory to avoid disk accesses.

Our design decision to work with URL hashes rather than complete URLs means that if two distinct URLs hash to the same value, the hint for one of them will be incorrect and a node may waste time asking another cache for data it may not have. In that case, because the read request contains the full URL, the remote node will return an error and the first node will treat the request as a miss. With 64-bit hash keys based on MD5 signature of the URLs, we do not

[1] A special value for the hash is used to signify an invalid entry.

anticipate that hash collisions will hurt performance.

Fixed-sized records simplify and speed up manipulation of the on-disk data structures when the hint cache does not fit in physical memory. The system currently stores hints in an array that it manages as a 4-way associative cache indexed by the URL hash, and it maps this array to a file. Thus, if a needed hint is not already cached in memory, the system can locate and read it with a single disk access.[2]

We store entries in a 4-way associative cache rather than, for instance, maintaining a fully-associative LRU list of entries to reduce the cost of maintaining the hint cache when it does not fit in memory. We include a modest amount of associativity to guard against the case when several hot URLs land in the same hash bucket. One concern with using a hash-based data structure is that this approach does not cluster "hot" objects onto memory pages, so it may exhibit poor memory page locality. We are therefore considering adding a front-end cache of hint entries. However, it is not clear that any arrangement of a hint cache will yield good memory locality because the stream of references to the hint cache exhibits poor locality. In particular, reads of the hint cache have been filtered by the proxy's cache of objects stored locally, and once an entry for an object is read from the hint cache, that object will be stored in the proxy's cache; it is then unlikely that that hint will soon be referenced again. It does seem possible that there will be some temporal locality among updates to hint entries or between updates and reads (e.g., if an object is read into a cache by one node, that hint may soon be read by another). Our system does provide a mechanism for preferentially caching recently updated entries.

We measured the time to do a hint lookup to be 4.3 μs, in the case when the hint cache fits in memory and 10.8 ms, in the case when the hint must be faulted in from disk. These measurements were on a 200 MHz Sun Ultra-2, which is a 7.72 SPECInt95 machine.

[2] We currently store the array in a file, so accesses may cause disk accesses of file system index structures. It would be trivial to store this array on the raw disk device if we find these metadata accesses to be expensive.

6.5.2 Implementing Push-shared

The push-shared algorithm requires that each proxy determine the least common ancestor between itself and any other proxy in the metadata hierarchy. To obtain this information, we extended the implementation of the metadata propagation to include a hop-count. When a proxy propagates a hint message through the metadata hierarchy, it increments a hop-count for each level in the upward path. The hop-count represents the number of levels up the metadata hierarchy the hint was propagated. Each proxy stores the hop-count for every hint. This information is used by the receiving proxy to determine how many levels apart the sending proxy was, in the hierarchy. The hop-count adds an additional 4 bits (for 16 levels) to the hint record stored, and an additional 4 bits to the hint message that is propagated. When a proxy receives an access request, based on an earlier hint propagation, the GET request has an attached hop-count. The proxy with the object propagates a hint message, with the PUSH flag set along with the hop count it received. A hint message for push is propagated up the metadata hierarchy for the number of levels specified by the hop count. When the specified number of levels is reached, the proxy pushes the hint to any randomly selected proxy in its subtree. A cache on receiving a push hint requests the object from the sending proxy after some random delay. The push hint only propagates a directive to the caches to prefetch the object and does not propagate the actual data. The caches use a random delay before issuing a GET request to avoid a sudden burst of GET requests for the object.

6.5.3 Experimental Results

In this section, we compare the performance of Cuttlefish with that of Squid. Cuttlefish implements a hint cache to track the nearest proxy that contains the each object. Since this look up is in the critical path of each remote request, it is an important parameter for evaluating the performance of Cuttlefish.

To measure this overhead, we configured a simple hierarchy consisting of a client and two caches all connected by 100MBit/s Ethernet. In Squid, the clients act as an L1 and L2 cache; in Cuttlefish, they are peers. We arrange for a client to send a series of requests to one cache and for the data to satisfy those requests to come from the other cache. The difference in measured time represents the extra cost of the hint directory lookup compared to Squid's simple decision to access its single parent.

We vary the size of the objects fetched and, for each size, we fetch a list of 10 different URLs, each of that size. We repeat this experiment five times. Thus, fetch times for for each size of document was measured fifty times for statistical confidence. In our strawman implementation the access time for Cuttlefish are 25% worse than that of Squid.

We wish to evaluate system performance under a realistic workload and under realistic network conditions. Unfortunately, since performance depends on sharing among nodes and the network performance between nodes, evaluating large cache systems is difficult. To make this study practical, we evaluate Cuttlefish and Squid under a trace-based workload in a novel configuration that models a large number of virtual nodes distributed across a WAN using a small number of physical nodes on a LAN. In this study we use careful mapping of virtual to physical nodes so that our measurements provide a realistic but pessimistic estimate of the performance an actual system would achieve, and we use live measurements of WAN performance to approximate WAN latencies.

Our mapping of virtual nodes to physical ones allows us to carefully examine the performance of a few nodes while simulating the rest of the system at a much coarser granularity. In particular, in the traditional hierarchy, the performance seen by a client depends directly only on three cache nodes: its direct L1, L2, and L3 ancestors. We therefore execute Squid caches on 3 nodes representing the 3 squid caches and *simulate* the rest of the system so that those nodes see the same stream of requests they would see in the real system. Thus we model a system consisting of several thousand nodes using three cache nodes, a few trace replay nodes, and a server node running Apache to handle cache misses.

Devising a reasonable mapping for Cuttlefish is more challenging because any cache may talk to any other cache. Again, we simplify the problem by considering performance as observed from one cache in the system. From this point of view, collections of distant caches can be modeled by a single node whose cache contents represent the union of the contents of the caches it models. We thus model a large Cuttlefish installation (as seen from one cache) as the L1 proxy, a Cuttlefish node that represents all nodes that are about as far away as an L2 proxy, and a Cuttlefish node that represents nodes as far away as an L3 proxy. Again, we use simulation to generate a stream of side-requests to the L2-distant and L3-distant proxies so that their cache contents are faithful to the collections of caches they model.

Ideally, we would have liked to run this test with the three caches spread over the United States. Squid and Cuttlefish, however, both require more disk space than is easy to arrange at remote sites. Instead, the caches and web server were all placed at UT, and the effect of remoteness was simulated through a 'delay service.'

Before each connection to a server or to another cache, to send a request, our Squids and Cuttlefish connected to 'delay servers' to model WAN delay. These servers implemented a very simple delay protocol. Clients initiate transactions by opening a connection to the server and sending a string stating a number of bytes; the client then waits for the server to send that many bytes, and then the client closes the connection. This models HTTP protocol delay between wherever the client and server are placed. In this case, the clients were all at UT Austin, and the servers were UCSD, UC Berkeley, and Cornell. Requests for the L2 cache generated delay transactions to UCSD, L3 requests generated delay transactions to Berkeley, and server requests generated delay transactions to Cornell. L1 delay was considered to be adequately modeled by delay on the UT LAN.

The workload for each cache was generated from requests in the Prodigy trace. The Prodigy trace was partitioned by source client, and each client's requests were assigned to a particular cache's request stream. The L1 cache was assigned the URLs of 200 clients. The L2 cache was thought of as a parent of 10 L1 caches, and assigned the URLs of 20,000 clients. The L3 cache was thought of as a parent to 10 L2 caches. In this comparison, Cuttlefish latency is about 30% lower than Squid latency.

6.6 Conclusions

We find that by minimizing the number of hops on a hit or miss, supporting widespread sharing, not slowing down misses, and avoiding centralized bottlenecks, a strategy of direct cache-to-cache data transfers supported by hint directories can improve performance by a factor of 1.27 to 2.30 for workloads and configurations we examine. Second, we find that simple push algorithms can support the principle of accessing data from nearby caches and achieve a further improvement of up to a factor of 1.25. Overall, our techniques provide speedups of 1.27 to 2.43 compared to a traditional cache hierarchy. Our prototype implementation over Squid is deployed among a set of caches at a few universities across the U.S.

Chapter 7

Cache Resource Management for Heterogeneous Data

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

—Karl Marx

7.1 Introduction

For effective cache performance, each proxy cache needs to share the resources among a heterogeneous set of objects with varying requirements. Since web objects are relatively large compared to file blocks, they are cached on disk instead of memory. Disk caches differ from memory caches in that they are much larger in size but have much lower bandwidth. For the current web traffic, consisting of mostly text and image data, a disk cache is equivalent to a large but slow memory cache. Thus, the algorithms currently used for web caching are extensions of traditional memory caching algorithms.

It is conjectured, that by the year 2005, more than 50% of the information available over the internet will contain continuous media (CM) data [50]. Currently continuous media data is accessed similar to text and images using the download-and-play mode. With a download-and-play mode of access, data is transferred completely to the client site before display. Due to the large sizes of CM objects, this results in large space usage and wait time at the client. Streaming mode of access addresses this issue by enabling the client to initiate display of data with only a small start-up latency (e.g., Vosaic, RealAudio, Macromedia, Progressive Networks, VDO, Vivo, VXtreme, Xing) [67]. To support the real-time requirements for streaming, web caches need to reserve cache bandwidth for each continuous media object. Thus, continuous media objects stored in the cache occupy both space and bandwidth. Since disks have limited bandwidth (disk bandwidths are around 2-7 MB/sec compared to memory bandwidths of 80-100 MB/sec), only a limited number of continuous media objects can be guaranteed real-time delivery. Current caching algorithms do not consider the space and bandwidth occupancy of the different types of objects. The main focus of this work is to develop cache replacement algorithms that handle different data types (continuous and non-continuous) for disk caches in the world wide web.

In this chapter, we present a resource based caching (RBC) algorithm for web proxies and servers. RBC makes replacement decisions based on the resource requirement of different objects and their contribution to the overall cache performance. The key aspect of RBC is selecting the unit of object to cache and determining what to replace based on the constrained cache resource. The algorithm dynamically selects the granularity of the entity to be cached that minimally uses the limited cache resource (i.e., bandwidth or space). The basic idea of the replacement policy is to model the cache as a two-constraint knapsack. RBC provides heuristics to convert the two-constraint knapsack to multiple single constraint knapsacks. The formulation of the algorithm enables it to be generalized for handling different data types and cache performance objectives (hit ratio, byte hit ratio etc.). We present extensive simulation results to evaluate our caching algorithm, and show that RBC outperforms other known caching algorithms. We have also instantiated our caching algorithm in a prototype proxy cache.

7.1.1 Related Work on Caching Algorithms

In order to review the extensive work on caching algorithms, we first discuss the traditional memory caching of fixed size pages. Next, we describe previous work on caching continuous media objects in memory. Finally, we describe proposed extensions of memory caching algorithms for disk caching to accommodate variable sized web documents (text and images only) and motivate the need for new algorithms to handle multiple data types in the WWW.

Traditionally, to reduce disk accesses and improve program performance, a subset of pages most likely to be accessed again are cached in memory [9, 27, 31, 37, 38, 84]. To maximize the number of accesses served from cache, when the cache capacity is exceeded the caching algorithm replaces pages to be accessed furthest in time. To estimate the time of the next access to a page, these algorithms use heuristics based on the recency or frequency of access.

- Recency-based algorithms: These algorithms exploit the *locality of reference* inherent in programs and hence, use the recency of access as an indicator of the time of the next access. LRU (Least Recently Used) is the most commonly used recency based algorithm. It maintains the time of last access for each page and replaces the least recently used page from the cache [31]. The CLOCK algorithm used in VM systems maintains the access count within a time interval for each page in cache and then replaces those that have not been accessed within that interval [87].

- Frequency-based algorithms: These algorithms are suited for skewed access patterns in which a large fraction of the accesses go to a disproportionately small set of *hot* objects [31]. LFU (Least Frequently Used), which maintains a reference count for each page in the cache and then replaces the least frequently used page, is the most commonly used algorithm [23]. A variant of LFU called FBR (frequency based replacement) considers only non correlated accesses [100] [1].

Frequency and recency based algorithms form the two ends of a spectrum of caching algorithms. The limitation of frequency based algorithms is that they do not adapt to changing arrival patterns and bursty accesses. On the other hand, recency based algorithms do not consider long term popularity of pages. Several algorithms that try to balance these tradeoffs have been proposed. Two of these algorithms are LRFU (Least Recently Frequently Used) and

[1] All accesses that occur within the correlated time interval are treated as a single access. This is to mask multiple low level accesses as they do not indicate the long term popularity of the page.

LRU-k [78, 88]. LRFU records the total number of accesses to an object and assigns a weight for each access. The weight (a value less than 1) of an access decreases with age. By selecting weights appropriately, LRFU balances recency and frequency of access [78]. LRU-k extends LRU by using the time of the k^{th}-to-last access for replacement decisions. The page whose k^{th}-to-last access occurred least recently is a candidate for removal. By appropriately selecting the value of k, LRU-k balances recency and frequency of access; a small k value favors recency and a large value favors frequency [70, 88]. In addition to these algorithms, several other algorithms that utilize the knowledge of user access patterns have been developed [105]. These include the hot-set and the locality-set models used for optimizing query processing in databases; as well as algorithms that exploit sequential access patterns by prefetching and immediate page discarding [23]. Other efforts include mechanisms that enable application controlled caching policies. These include LRU-SP for file caching with global guarantees [14]; informed caching and prefetching for files using the access pattern hints from the application [94]; and Mach VM page caching with application specified hints (PREMO pager) [82].

All these caching algorithms focus on minimizing response times for accesses predominantly on textual/numeric and static image data. Since continuous media (CM) accesses are sequential, periodic and require real-time guarantees, minimizing response times is not the primary goal. Consequently, the objective of caching algorithms for CM data is to maximize the number of requests that can be provided guaranteed service.

Caching for CM data has been studied in the context of memory caches [28, 29, 30, 89]. Since CM data have periodic and sequential accesses, the buffer allocated to a block can be reassigned, after use, to the next access. However, if another user is accessing the same data, within a short time interval, the block can be retained in the buffer for the later user. Examples of caching algorithms that are optimized for CM accesses include *interval caching* [29, 30] and *distance caching* [89]. These algorithms exploit the sequentiality of CM accesses and cache *intervals* formed by pairs of consecutive accesses to the same CM object. For any pair of consecutive requests to the same CM object, the later request can read the data buffered by the earlier request. The two requests that form such a consecutive pair are called the *writer* (or preceding request) and the *reader* (or following request). For instance, in Figure 7.1, arrows

S_1 through S_6 represent various requests for CM objects 1 and 2; and b_1 through b_4 denote the intervals formed between pairs of consecutive requests. For the interval b_1 consisting of the request pair $[S_1, S_2]$, S_1 is the writer and S_2 the reader. Whereas S_1 accesses data blocks from the server and retains them in the cache buffers, S_2 reads the blocks from the cache and frees the buffers. The amount of data buffered, which is the size of the interval, depends on the inter-arrival time between the pair of requests and the data access rate. For example, if the inter-arrival time between S_1 and S_2 is 10 seconds and the data access rate is 2 blocks/sec, then the size of the cached interval b_1 is 20 blocks. Note that the interval is formed only after the arrival of S_2. On the arrival of S_2, S_1 stops discarding the used buffers; S_2 accesses the first 10 seconds of data from the server; after 10 seconds, S_2 continues to read the next set of consecutive blocks from cache and frees the buffers for reuse by S_1. To maximize the number of requests served from cache for a given cache size, the interval caching algorithm orders the intervals in ascending order of their storage space requirements and discards those that cannot fit in the cache. In Figure 7.1, the ordering of intervals increasing size is b_1, b_4, b_3, b_2. Generalized interval caching extends the definition of an interval to include small sized objects.

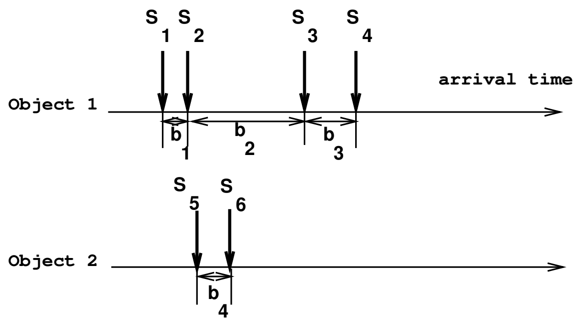


Figure 7.1: Interval Caching for Continuous Media

Observe that all of these algorithms for memory caches assume that the system is limited by disk performance. However, in the context of the WWW, where a request involves retrieving objects from remote servers, the network becomes a bottleneck. Additionally, the web caching environment differs from memory caching in that: (1) the cached object have varying sizes (instead of fixed size pages), (2) the cached data could be shared among clients, and (3) the cached objects are of different types. Current disk-based web caching algorithms are adaptations of traditional memory caching [34, 36, 39]. For example, commonly used access attributes for replacement policies are: (1) time of last reference among a set of objects (Squid Cache) [122], (2) the time-of-day of access [96], (3) the size of the object (or its logarithm) [39], (4) the number of accesses to an object over a time interval as the primary key and the time for last access as the secondary key (e.g., the HyperG) [35], (5) the logarithm of object sizes as the primary key and the time for last access as the secondary key (LRU-MIN) [36]), (6) time for last access per object group where the logarithm of object sizes is used to group objects [68], and (7) the estimate of the average page loading delay and retrieval time [10, 126].

Since most of the data accessed on the web today contains text and static images, the above described algorithms seem adequate. Recent studies, however, predict a shift towards continuous media data in the future [50]. Currently continuous media data is accessed similar to text and images using the download-and-play mode. Due to the large size of CM data, this results in large space usage and wait time at the client. Streaming mode of access addresses this issue by enabling the client to display the data received in real-time with only a small start-up latency [67]. Given these trends the web caches will need to serve continuous media data in streaming mode, reserving cache bandwidth for each object. Since disks have limited bandwidth, memory caching algorithms for CM objects cannot be directly adapted for web caches. For example, the interval caching policy, used for memory caching of CM data, caches only the shortest deterministic intervals to minimize the space usage, but incurs the overhead of writing a block to the cache for each block read from the cache. Consequently, the effective cache bandwidth available for reads, in the worst case, can reduce to 50% of the total bandwidth. This limits the number of user requests that can be served from the cache. At the other extreme, traditional caching techniques (e.g., frequency-based caching), that cache the entire object, amortize the writing overhead over a large number of reads, but can only store a few objects in the cache due to the large storage space requirement of CM objects. Given that disk caches are limited by both space and bandwidth, in order to maximize the hit ratio, web cache management algorithms have to efficiently manage both these resources.

In summary, to overcome the dynamically occurring overloads at web servers, objects are cached on disks at intermediate proxy servers. Since disks are limited by both bandwidth and space, in addition to multiplexing the cache space among multiple data types, cache management algorithms will be required to efficiently utilize both cache space and bandwidth. None of the existing caching algorithms achieve all of these objectives.

The rest of this chapter is organized as follows. In Section 7.2, we describe our resource based caching algorithm. Results of our simulations are presented in Section 7.3, and finally, Section 7.4 summarizes our results.

7.2 Resource Based Caching (RBC)

Consider the hierarchical information delivery architecture of the WWW, with a remote server storing different types of objects. The objects stored at the remote server are classified into two categories: continuous (i.e., audio and video) and non-continuous (e.g., textual and numeric data, imagery, etc.) media objects. The resource requirement of each object O_i, $i \in [1..N]$ is defined as the tuple $\langle s_i, b_i \rangle$, where s_i and b_i, respectively, denote the size and access bandwidth. Let an intermediate server maintain a cache of size S and bandwidth B. The atomic unit of an object in cache is called an entity. The entity granularity (e.g., entire file, block, group of blocks) is not fixed but depends on the type of object. CM entities are further classified into *dynamic* or *static* entities. For dynamic entities (e.g., intervals), the contents of the entity changes with time, while for static entities (e.g, entire objects)the contents are fixed. Let $\mathcal{E} = \{E_1, E_2, ..., E_n\}$ denote the set of entities in the cache, and let $\widehat{s}_i$ and $\widehat{b}_i$, respectively denote the cache space and bandwidth occupied by entity E_i. The cache space and bandwidth is shared by both continuous and non-continuous media objects. Each CM object requires bandwidth reservation, while non-CM objects are served on a best-effort basis. In order to avoid starvation of non-CM objects, a fraction of the cache bandwidth is pre-allocated for the entire class of non-CM objects (the remaining cache bandwidth is shared by both CM and non-CM objects).

In what follows, we describe the main components of the RBC algorithm which consist of defining the entity characteristics, selecting the desired entity to cache, and determining which, if any, entity to replace.

7.2.1 Entity Characteristics

The characteristics of an entity are its: (i) granularity, (ii) resource requirement, and (iii) caching gain.

7.2.1.1 Granularity of Cached Entity

Depending on its type (continuous or non-continuous media), an object can be cached partially (e.g., block, group of blocks) or in its entirety.

- *Continuous media objects:* Due to the sequential and periodic nature of access, a request for a CM object can be served from the cache by maintaining a set of consecutive blocks forming an interval between two successive requests to the same object. An interval consists of a single reader and writer. In general, if multiple users access the same object concurrently, a set of adjacent intervals could be grouped together to form *runs*, with a single writer and multiple readers. This introduces a range of possible entities that can be cached; from an interval to single-writer multiple-reader runs, to the entire object[2]. Figure 7.2 shows the runs r_1 to r_4, that are formed for a set of users accessing the same data object. The run r_1 is a simple interval of writer S_1 and reader S_2, while run r_4 consists with writer S_1 and readers S_2, S_3, S_4, S_5. Since run caching not only amortizes the write overhead over multiple readers but also uses less space than the entire object, it is useful when both space and bandwidth of the cache are equally limited.

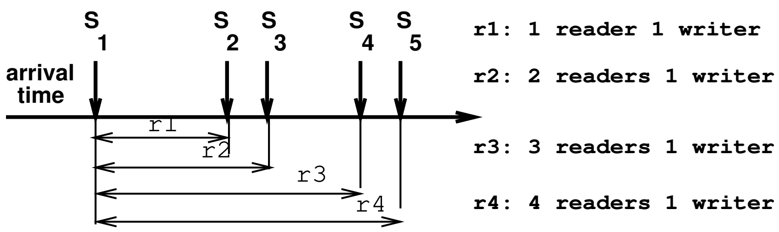


Figure 7.2: Run Caching for Continuous Media

[2]Note, in interval caching grouping of readers was done only if the previous request was already being read from cache.

- *Non-continuous media objects:* Since the HTTP/1.0 protocol does not support random block accesses, non-CM objects in the WWW are normally accessed, and hence, cached in their entirety.

7.2.1.2 Estimating Resource Requirement

The resource requirement of an entity is given by the cache space and bandwidth occupied by it.

- *Continuous media objects:* If an entity E_i consists of the entire object O_j, then its cache space occupancy equals the object size, i.e., $\widehat{s}_i = s_j$. On the other hand, if E_i is a dynamic entity (interval or run) of duration t_i, with an access bandwidth b_i, then

$$\widehat{s}_i = \min\{s_j, t_i * b_i\}$$

  The bandwidth occupancy of an entity is the cumulative access bandwidth required by its concurrent readers and writers. An interval has exactly one reader and one writer; a run, on the other hand, has multiple readers and one writer. If the entity is an entire object, then it could have multiple concurrent readers and at most one writer (if the object is being concurrently written to cache). Thus, for entity E_i with access bandwidth b_i per user, r_i concurrent readers and w_i ($w_i = 0 \; or \; 1$) writers, the bandwidth occupancy is

$$\widehat{b}_i = (r_i + w_i) * b_i$$

- *Non-continuous media objects:* Since non-CM objects are cached in their entirety, if entity E_i, stores object O_j, then its space occupancy is given by $\widehat{s}_i = s_j$. Given that non-CM objects are served on a best-effort basis, we will assume for the purpose of the caching policy that, for all non-CM objects the entity's bandwidth occupancy $\widehat{b}_i = 0$. The cache bandwidth consumed for accessing them is allocated from the reserved bandwidth pool for non-CM objects.

7.2.1.3 Caching Gain

There are many metrics for measuring the overall performance improvement due to caching. Two such metrics are: (i) *Hit Ratio (HR)* which denotes the ratio of number of accesses from cache to the total number of accesses, and (ii) *Byte Hit Ratio (BHR)* which equals the ratio of the total number of bytes accessed from cache to the total number of bytes accessed. The caching gain of an entity depends on the metric used to measure cache performance.

- *Continuous media entities:* To maximize BHR, the caching gain g_i of a CM entity E_i is defined as the total bandwidth saved (i.e., the bytes transferred per second). For an entity with access bandwidth b_i and an estimated number of concurrent readers $\widehat{r_i}$, the gain is given by

$$g_i = \widehat{r_i} b_i \tag{7.1}$$

  For a dynamic entity (interval or a run), $\widehat{r_i}$ is known at the time the run or interval is formed, and is equal to the number of concurrent readers r_i. For a static entity (i.e., the entire object), on the other hand, the estimated number of concurrent readers is computed from the interarrival time between requests (also referred to as the time-to-reaccess or TTR). Thus, if a static entity has access probability p_i and λ is the request arrival rate in the system, then the interarrival time between requests is $\frac{1}{\lambda p_i}$. Knowing that the entity consists of the entire object, each user will be serviced from the cache for a duration of $\widehat{s_i}/b_i$. The estimated number of concurrent readers is given by the ratio of the time duration of the object to the interarrival time between requests. Hence,

$$\widehat{r_i} = \frac{\widehat{s_i}/b_i}{1/(\lambda p_i)} = \lambda p_i \widehat{s_i}/b_i \tag{7.2}$$

  To maximize HR, the gain of a CM entity is defined as the number of requests per second, which is the inverse of the interarrival time between requests. Hence,

$$g_i = \lambda p_i = \widehat{r_i} b_i / \widehat{s_i} \tag{7.3}$$

- *Non-continuous media entities:* For a non-CM entity E_i, the caching gain for maximizing BHR is defined as the bytes transferred per second, i.e., the ratio of the total number of bytes transferred to the interarrival time between requests. The gain is given by

$$g_i = \frac{\widehat{s}_i}{1/\lambda p_i} = \widehat{s}_i \lambda p_i \tag{7.4}$$

where p_i is the access probability and $1/\lambda p_i$ is the interarrival time between requests. For maximizing HR, the gain of a non-CM entity is defined as the number of requests per second, i.e.,

$$g_i = \lambda p_i \tag{7.5}$$

Note that depending on the metric used to measure cache performance, RBC selects the appropriate gain function.

Observe that the value of the caching gain depends on the estimate of the time-to-reaccess or the access probability of an object. Since $TTR = \frac{1}{\lambda p_i}$, determining either is sufficient. The long term access probability could be known apriori without knowing the exact access sequence (e.g., VOD applications with slow changing access patterns), or estimated by maintaining access statistics. Techniques used to determine the access probability or TTR are outlined below.

- *Reference Count (REFCNT)* — In this technique, the reference count is used to estimate the access probability (e.g., LFU, FBR use reference count [23]). However, the reference count measure does not capture the recency of objects.

- *Time of Access (REFTIME)* — In this technique, the history of access times is used to estimate the interarrival time [88]. The LRU technique uses only the time of the last access, while the LRU-k uses the time of the k^{th}-to-last access. A typical value selected for k is 2 (i.e., LRU-2). One of the drawbacks of this technique is that it requires the history of the last k accesses to be maintained per object, which imposes significant overhead for large values of k. Also, selecting a large value of k to favor frequency of access ignores the recency of the previous k-1 accesses. A small k value favors recency of access, but ignores the long term frequency.

To overcome the drawbacks of the above techniques, we designed the WLRU-n technique in which the mean time-to-reaccess (MTTR) is computed as the weighted sum of the interarrival time between the previous accesses. Let the access times of the last n accesses be $t_n, t_{n-1} ... t_1$, where t_i is time of the last i^{th} access (i.e, t_1 is the most recent access time and t_0 is the current time). Let the weighting function, $W(i)$, be decreasing, $W(i) < W(i-1)$. Then $MTTR = \Sigma^{i \geq 0}(t_i - t_{i+1}) * W(i)$ Observe that for $W(i) = \alpha W(i-1)$, $\alpha \leq 1$ and $W(0) = 1 - \alpha$, the history information of the previous access times need not be maintained. Thus MTTR at time t_0 is, $MTTR(t_0) = (1-\alpha)(t_0 - t_1) + \alpha MTTR(t_1)$. The averaging factor α can be tuned to bias for or against recency.

Having defined the set of entities that can be cached and their characteristics, we now present the procedure for (i) selecting the granularity of the cached entity, and (ii) determining which, if any, of the cached entities to replace, that underly the RBC algorithm.

7.2.2 Selection of Cached Entity

When a request arrives for an object not in cache, the RBC algorithms selects the granularity of the entity to cache. This selection depends on the state of the cache and the entity characteristics.

The *state of the cache* is defined as a pair $(\mathcal{U}_s, \mathcal{U}_b)$, where $\mathcal{U}_s$ and $\mathcal{U}_b$ denote the utilization of the cache space and bandwidth, respectively[3]. Specifically,

$$\mathcal{U}_s = \frac{1}{S} \sum_{E_i \in \mathcal{E}} \widehat{s}_i; \qquad \mathcal{U}_b = \frac{1}{B} \sum_{E_i \in \mathcal{E}} \widehat{b}_i; \quad 0 \leq \mathcal{U}_s, \mathcal{U}_b \leq 1.$$

To ensure that both the cache space and bandwidth are effectively utilized, RBC selects the granularity of the entity (entire object, interval, etc.,) so as to minimally use the resource that is currently over-utilized. This prevents either space or bandwidth to be saturated while the other remains under-utilized. Observe that if the state of the cache is represented as a point within a unit square (where the x- and y-axis denote the space and the bandwidth utilization, respectively (see Figure 7.3), then all the points close to the diagonal capture the state where $\mathcal{U}_s \approx \mathcal{U}_b$. If the selection is made such that the state of the cache moves towards the diagonal, then it achieves an equitable utilization of cache space and bandwidth. Recall that for a CM

[3] The cache bandwidth B in the equation represents the total bandwidth available for CM objects.

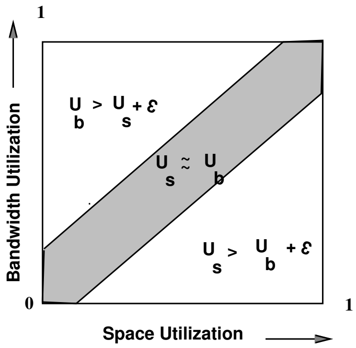


Figure 7.3: Cache State

object there are different possible entity granularity (entire object, interval, run). The selection methodology is described below:

- If the bandwidth utilization of the cache is greater than the space utilization , i.e., $\mathcal{U}_b > \mathcal{U}_s + \epsilon$, then to minimize the wasted bandwidth, the entity with the lowest write bandwidth overhead is selected[4]. The write overhead for entity E_i is $\frac{w_i}{r_i+w_i}$, where r_i is the number current readers and w_i is the number of writers. Note that for an interval, due to the continuous discarding of blocks by the reader, the write overhead is 50%.

- If the space utilization of the cache is greater than the bandwidth utilization, i.e., $\mathcal{U}_s > \mathcal{U}_b + \epsilon$, then to minimize the space usage, the entity with the lowest space usage $\widehat{s_i}$, is selected.

- If the bandwidth and space utilization of the cache are approximately equal, i.e., $\mathcal{U}_s - \epsilon < \mathcal{U}_b < \mathcal{U}_s + \epsilon$, then both the space and wasted bandwidth overheads are minimized. The entity selected has the largest effective bandwidth to space overheard ratio. The effective bandwidth is the bandwidth that is available for reads. The entity with the largest ratio has the lowest write bandwidth overhead per unit space used. The ratio is given by $\frac{1-\frac{w_i}{r_i+w_i}}{\widehat{s_i}}$.

[4] ϵ, the approximation bound, is a tunable parameter.

7.2.3 Cached Entity Replacement Algorithm

Having determined the entity to cache, RBC determines which, if any, of the entities present in the cache should be replaced. Unlike processor memory caching, in RBC the accessed entity is not always cached.

To identify the candidates for removal, RBC orders the accessed entity and those present in the cache by *goodness*, which depends on (i) the caching gain g_i, (ii) the resource usage $\langle \widehat{s}_i$ and $\widehat{b}_i \rangle$ and, (iii) the cache state $(\mathcal{U}_s, \mathcal{U}_b)$. All the entities in the cache are sorted in ascending order of their goodness values. The selection of entities for removal proceeds from the one with the worst goodness value till sufficient resources have been released to accommodate the new entity. To resolve a tie among entities with equal goodness values, a *best-fit* selection is made based on the cache resource used by each entity. The best-fit selection is used to ensure that the decrease in the caching gain by the removal of entities is minimum.

The computation of the goodness value depends on whether the cache, with respect to the entity E, is: (i) *unconstrained*, (ii) *space constrained*, (iii) *bandwidth constrained*, or (iv) *bandwidth and space constrained*. For simplicity of description, we assume that the cache performance objective is to maximize the byte-hit-ratio. Similar arguments and analysis can be carried out for maximizing hit ratio and other performance objectives.

1. **Unconstrained**: The cache is said to be unconstrained if the cache space and bandwidth available exceed the requirements of the entity to be cached. In this scenario, the entity is placed in cache and the state of the cache is appropriately updated.

2. **Space Constrained**: The cache is said to be space constrained if the available bandwidth is sufficient to accommodate entity E (i.e., $\widehat{b} \leq (1 - \mathcal{U}_b) * B$) but the available space is not (i.e., $\widehat{s} > (1 - \mathcal{U}_s) * S$). In this case, each entity E_i is assigned a *goodness* value $\mathcal{G}_i$, which represents the gain per unit space used, or the *gain density*. Thus, for caching gain g_i and space used $\widehat{s}_i$ the goodness is given by

$$\mathcal{G}_i = g_i / \widehat{s}_i \tag{7.6}$$

Recall that for the objective of maximizing BHR, the gain $g_i = \widehat{r_i} * b_i$, which for entire objects reduces to $g_i = \lambda p_i \widehat{s_i}$ (Equation 7.1, 7.2). Thus, the goodness equation reduces to $\mathcal{G}_i = \lambda p_i = 1/TTR$. The goodness value maps to the inverse of the time-to-reaccess the object (similar substitutions can be made for intervals). Observe that this goodness value is identical to the metric used for traditional space constrained processor caches that replace the data block that has the largest time-to-reaccess.

The cache replacement algorithm in the space constrained case is described in detail in Figure 7.4(a).

3. **Bandwidth constrained**: The cache is said to be bandwidth constrained if the available space is sufficient to accommodate the entity E (i.e., $\widehat{s} \leq (1 - \mathcal{U}_s) * S$) but the available bandwidth is not (i.e., $\widehat{b} > (1 - \mathcal{U}_b) * B$). We define *goodness* value of a continuous media entities E_i, in this case to be the gain per unit cache bandwidth used, that is

$$\mathcal{G}_i = g_i / \widehat{b_i}$$

Recall that to maximize BHR, $g_i = r_i b_i$ and $\widehat{b_i} = (r_i + w_i) * b_i$, thus $\mathcal{G}_i = \frac{r_i}{r_i + w_i}$. Thus for maximizing BHR, the goodness ordering maps to the ratio of total bandwidth saved to the cache bandwidth used. A ratio of 1, indicates that an entity uses all the cache bandwidth for reading from the cache and no bandwidth is wasted in writing to the cache. This is the case when the object is not being concurrently written to the cache. The goodness value measures to what degree the write overhead is amortized over the gain from cache reads. A larger goodness value has a smaller write overhead per reader and hence is better. The bandwidth constrained scenario occurs only for CM objects and non-CM objects are not considered for removal as they are served on a best-effort basis. The cache replacement algorithm in the bandwidth constrained case is similar to the one described in Figure 7.4(a).

4. **Bandwidth and Space Constrained**: The cache is said to be bandwidth and space constrained if neither the available space nor the available bandwidth is sufficient to accommodate entity E_i (i.e., $\widehat{s_i} > (1 - \mathcal{U}_s) * S$ and $\widehat{b_i} > (1 - \mathcal{U}_b) * B$). In this case the

cached entities are ordered into two lists. An *s-list*, which orders entities by their gain per unit space (i.e, $\mathcal{G}_i = g_i/\widehat{s_i}$), and a *b-list* that orders by the gain per unit bandwidth (i.e, $\mathcal{G}_i = g_i/\widehat{b_i}$). The algorithm removes entities from either of the two lists until enough resources are released for the new entity. Let S_{free} and B_{free}, respectively, represent the currently available space and bandwidth. RBC selects an entity at the top of the s-list if $S_{free}/\widehat{s_i} < B_{free}/\widehat{b_i}$, otherwise the one at the top of the b-list is selected. After that entity is marked for possible removal, the values of S_{free} and B_{free} are re-computed, and the list selection process is repeated. In this procedure, the entities selected for removal are those that have the worst goodness value in either the bandwidth constrained case or the space constrained case. The cache replacement algorithm in the dual constrained case is described in Figure 7.4(b).

To avoid the computational overhead of handling two lists, a single goodness value can be used. The original gain function is weighted by the fractional bandwidth wasted in writes. The goodness value is the weighted gain per unit space and is given by, $\mathcal{G}_i = \frac{g_i(1-w_i b_i/\widehat{b_i})}{\widehat{s_i}}$. A detailed discussion of such an optimization, however, is beyond the scope of this dissertation.

7.2.3.1 Properties of the Entity Replacement Algorithm

The entity replacement algorithm has the following properties:

Property 1 *When multiple entities ordered by $\mathcal{G}_i$ values are removed from cache, their combined goodness is less than that of the entities remaining in the cache.*

Proof: Let entities E_i, E_j, E_k be ordered by goodness. Let bs_i, bs_j, bs_k be the constrained resource usage, respectively. Hence, $\frac{g_i}{\widehat{bs_i}} < \frac{g_j}{\widehat{bs_j}} < \frac{g_k}{\widehat{bs_k}}$. Now, if entities i and j are removed from cache, and entity k remains in cache, their combined goodness is the ratio of the combined gain to resource occupied. Hence, by definition and the above inequality,

$$\frac{g_i + g_j}{\widehat{bs_i} + \widehat{bs_j}} < \frac{g_k}{\widehat{bs_k}}$$

```
space _constrained_policy (new entity E_i)
        ∀ entities E_k in cache
          if (𝒢_k < 𝒢_i)
            removelist = removelist '+' E_k
          order cachelist in ascending 𝒢 values; i < k, 𝒢_i ≤ 𝒢_k
        while (space_free < ŝ_i)
              space_free = space_free + ŝ_k;
              mark E_k for deletion; k++
        if (space_free ≥ ŝ_i)
          second pass release any extra space marked
          delete marked entities
          return success
        else return failure
```

(a) Space Constrained

```
bw _space_constrained_policy (new entity E_i)
    ∀ entities E_k in cache
      if (g_k/b̂_k < g_i/b̂_i) bwlist = bwlist '+' E_k
      if (g_k/ŝ_k < g_i/ŝ_i) splist = splist '+' E_k
      order splist and bwlist by increasing goodness
    while (bw_free < b̂_i & space_free < ŝ_i )
          if (empty splist & bwlist )
            break;
          if ( bw_free/b̂_i < space_free/ŝ_i )
            select k from bwlist; remove k from lists
          else
              select k from splist; remove k from lists
          mark k for deletion
          update bw_free and space_free
    if (bw_free ≥ b̂_i & space_free ≥ ŝ_i)
      release any extra bandwidth or space marked
      delete marked entries
      return success
    else return failure
```

(b) Bandwidth and Space Constrained

Figure 7.4: Cache Replacement Policy

Corollary: If a new entity is added to the cache, its goodness value is greater than the combined goodness value of all the entities it replaces from the cache.

Property 2 *The best-fit choice to resolve a tie among entities with equal goodness values ensures that, for a given amount of resource required (space or bandwidth), the caching gain is maximized.*

Proof: If two entities, E_i and E_j, have goodness values, $\mathcal{G}_i = \mathcal{G}_j$ then $\frac{g_i}{\widehat{bs_i}} = \frac{g_j}{\widehat{bs_j}}$. Let the resource required be BS_{req}. If $\widehat{bs_i} > BS_{req}$ and $\widehat{bs_j} > BS_{req}$, then by best fit the entity selected for deletion has resource usage $bs_k = min(\widehat{bs_i}, \widehat{bs_j})$; the other entity with the gain of $max(g_i, g_j)$ remains. The best-fit selection can also be applied over groups of entities, if individual $\widehat{bs_i}$ values are smaller than the resource required. The combined entity will have the same goodness value as the individual ones; i.e.

$$\frac{g_i + g_j}{\widehat{bs_i} + \widehat{bs_j}} = \mathcal{G}_i = \mathcal{G}_j$$

and hence can be treated as a single entity.

7.2.4 Discussion

The problem of determining a caching algorithm that minimizes the load on the remote server and network for a cache with limited size and bandwidth, can be mapped to a two constraint knapsack problem. The RBC algorithm reduces this problem to two single constraint knapsack problems. For a single constraint cache (e.g., limited space), the caching problem reduces to the 0/1 knapsack problem. The 0/1 knapsack problem consists of filling a knapsack of fixed size S, with objects x_i each with a gain g_i and a size s_i, such that the total gain is maximized. The objective is to maximize $x_i g_i$ under the constraint that $x_i s_i < S$, where $x_i \in 0, 1$ depending on whether the object is selected or not. The 0/1 knapsack problem is known to be NP-complete. However, if the gain values for each object are identical, then the 0/1 knapsack has an optimal solution which consists of ordering the objects in ascending order of size [92].

For a fractional knapsack problem (objects are not whole but can be fractional), the optimal solution is the greedy ordering by gain density, g_i/s_i. For the 0/1 knapsack problem with varying values of g_i, selecting entities based on the greedy ordering of gain density is a simple approximation. Various approximate algorithms of the 0/1 knapsack problem with ϵ approximation factor exist, but they are all off-line algorithms [92]. The RBC algorithm uses the gain density approximation in ordering the entities for the different cache constraints. This approximation will be optimal if each entity has identical caching gain but varying resource requirement.

The objects in the cache are maintained as priority queues, one for each cache resource (i.e., an s-list and a b-list). The insertion in the queue is O(log(n)) operation where n is the number of entities in the cache. Each deletion is O(1), with a worst case of O(n) deletions. Note that the goodness value does not change until an object is accessed.

7.3 Experimental Evaluation

In order to compare the resource-based caching algorithm with other known algorithms, we have a developed a prototype system and an event-driven simulation model. The prototype is built on a cluster of AIX workstations, with one node being the proxy cache server and the remaining running the client code. The caching module at the cache server interacts with the local file-system and the remote server. The caching module maintains (i) multiple priority queues to implement the caching policy for each data type, (ii) data structures to store access information for each data type (access times, previous user list, reference count, reserved bandwidth etc.), and (iii) a hash list to map $\langle$filename, logical block number$\rangle$ to cached blocks. In on-going work, the caching module is integrated with the local multimedia file-system. To experiment with multiple configurations and understand the behavior of the caching algorithm, we have developed and experimented with an event driven simulation model, the results of which are described in the following sections. The workload used for the simulation model was derived after analyzing different web-server access logs.

7.3.1 Selecting Workloads

To determine the workload parameters, we analyzed the data gathered from NLANR/Squid cache, NCSA, NASA and Digital web-server logs [3, 76]. The workload is characterized by defining: (i) the distribution of requests among different objects types, (ii) the request arrival pattern or access distribution, and (iii) the object size distribution.

- *Access distribution across data types:* Table 7.1 shows the access distribution across different data types gathered from different web-server access logs [3, 76]. The current statistics indicate a predominance of text and image data accesses. However, it has been predicted that in coming years, with streaming video gaining popularity, these statistics will show a greater shift to CM data [50]. Given that web-server access-traces are mostly for text/image data, we use synthetic traces and access distributions across data types in our simulations.

Data Type	Hits %	Bytes %
Image	67	46
Text+html	15	10.6
Video	1	2
Audio	0	0
Other	17	39

NLANR Squid Cache

Data Type	Hits %	Bytes %
Image	48	36
Text+html	51	50
Video	0.1	6
Audio	0.2	3
Other	0.6	4

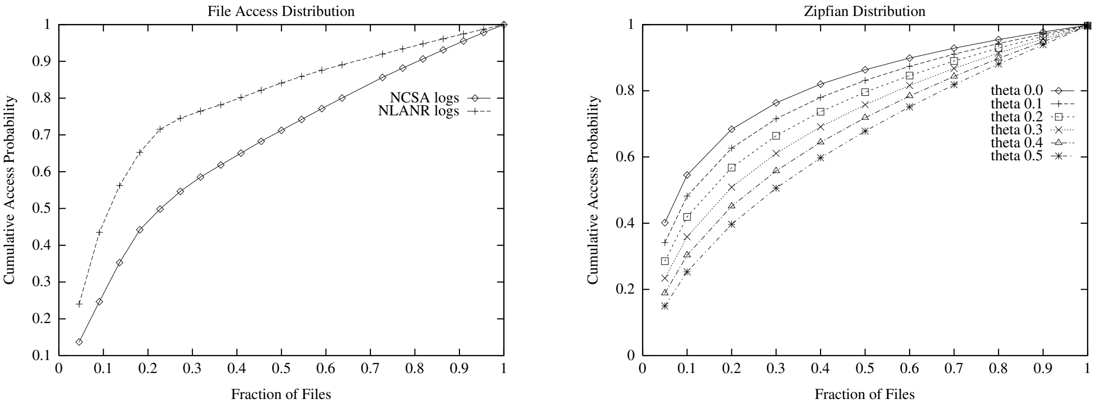
NCSA logs

Data Type	Hits %	Bytes %
Image	63	48
Text+html	30	18
Video	1	30
Audio	0.2	1
Other	2	2

NASA logs

Table 7.1: Web-server Request Distribution Data

- *Determining Access Skew:* The web-server access logs were also analyzed to determine the file access skew. Figure 7.5(a) depicts the file access skew graph for two web servers, NCSA and NLANR/Squid. It can be seen that about 70% of the accesses are restricted to 20% of the files, that is a 70-20 skew. Skews of the order of 80-20 and 90-10 are not uncommon [3]. It has been shown that the Zipf distribution can be used to accurately model access skew in environments where a small fraction of the objects account for a large fraction of the references (e.g., large digital multimedia libraries, web servers, etc.) [39]. In our workload, we selected the file access skew to be derived from a Zipfian distribution for different values of the skew parameter θ. A Zipf distribution yields high access skews at low values of θ, and tends to become more uniform at larger values of θ. Figure 7.5(b) shows the cumulative access probability of objects as a function of the fraction of objects accessed for a Zipf distribution with different values of θ. It demonstrates that, for a cumulative access probability of 75%, the fraction of objects accessed ranges from 26% to 60% as the value of θ increases from 0 to 0.4, which matches that observed in Figure 7.5(a).



(a) Access Log Distributions (b) Zipf Distribution

Figure 7.5: File Access Distribution

- *Object Sizes:* The file sizes at the NCSA/NASA servers shown in Table 7.2, consist of text and images objects ranging from 3 to 64KB and video and audio objects ranging from 100KB to 15MB. Although these figures show a relatively small-sized CM data files, they are more due to the limitations of current network bandwidth and viewing preferences. However, it can be seen from the table that the object sizes of text/images differ by an order of magnitude from audio and video. Therefore when selecting object sizes, in our workload, the short CM objects (e.g., audio size is around 1MB) are set to be order of magnitude smaller that the large objects (e.g., video range is 100MB), but are an order of magnitude larger than the text and image objects (100 KB). Each class of objects has a range of uniformly distributed sizes. The interarrival time between requests is derived from observed values in the web-server logs, and the mean rate λ is measured apriori.

Observe that, for CM objects, there is an upper bound on the maximum arrival rate. For instance, for a cache that can support at most 50 concurrent CM requests, each with a time duration of 15 minutes, the maximum arrival rate is 0.05[5].

Data Type	Size
Image	9-64 KB
Text+html	3-20 KB
Video	500KB - 15 MB
Audio	100KB - 300KB

Table 7.2: File Size Distribution

The other parameter that is varied in the experiments is the cache capacity, i.e., the space and bandwidth. Observe that the size and bandwidth requirements of objects, the values of mean request arrival rate λ, the Zipf skew parameter θ, as well as the desired value of hit ratio can be used to derive bounds on the cache size and bandwidth requirement. For example, for a Zipfian access skew with $\theta = 0.05$, a hit ratio of 0.75 can be achieved by maintaining 30% of the objects in the cache. Thus, for a database containing 100 objects, each with a mean size of 500MB, the approximate cache size requirement to achieve a hit ratio of 0.75 is 15GB. Similarly, if we assume that each CM request has a bandwidth requirement of 0.5MB/sec and a playback duration of 15 minutes, then an arrival rate of $\lambda = 0.05$ implies that there can be 50 concurrent requests. In such a scenario, to achieve a hit ratio of 0.75, the cache bandwidth must be at least $0.75 * 50 * 0.5 = 18.75$MB/sec. Note, that the space and bandwidth values assumed for cache are not representative of a single disk but that of a logical disk cache that could be mapped to a disk array.

In what follows, we analyze the behavior of the resource based caching algorithm and compare it with several known algorithms for varying workload parameters.

[5] This can derived from Little's Law of $N = \lambda T$.

7.3.2 Analysis of the RBC Algorithm

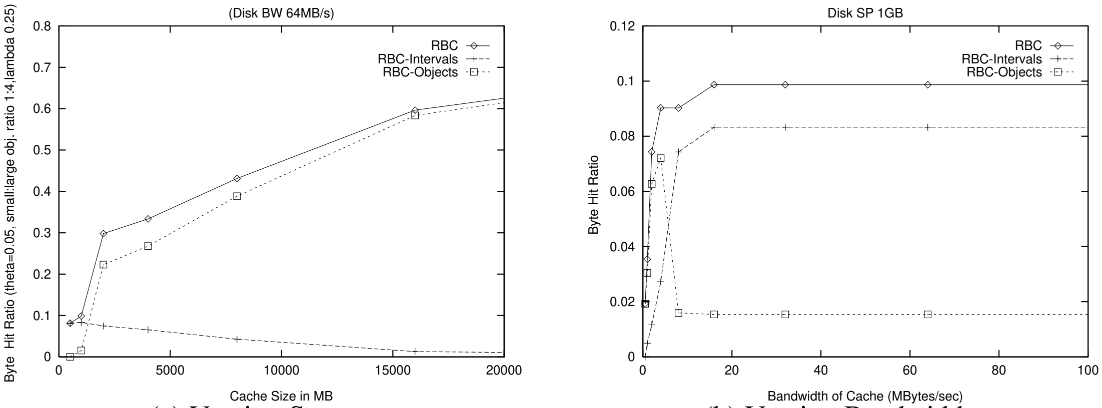
We analyze the behavior of the RBC algorithm for the objective of maximizing BHR, with varying cache space and bandwidth capacity. For this analysis, the workload is restricted to CM objects of varying size (short audio clips to large video streams) and bandwidth requirements; the granularity is restricted to entire objects and intervals. The algorithm uses the WLRU-n technique for estimating the time-to-reaccess for different objects which is used in the goodness computation based on the resource requirement and cache state. Figures 7.6(a,b) illustrate how the RBC algorithm selects the granularity of the cached entity as the cache resources vary. Figure 7.6(a) shows that, when the cache size is small (less than 2GB) for a fixed bandwidth of 64MB/sec, the RBC algorithm selects more intervals (marked in the figure as RBC-intervals) than entire objects (marked in the figure as RBC-objects)[6]. As the cache size increases and the cache bandwidth becomes the bottleneck, more of the cache is filled with entire objects. In fact, when the cache size is large ($>$ 20GB) only entire objects are selected for caching.

Figure 7.6(b), on the other hand, shows that, for a cache size of 1GB, when the cache bandwidth is low (less than 5MB/sec), a large proportion of the cache is filled with entire objects so as to amortize the writing overhead over a large number of reads. As the cache bandwidth increases, however, the cache size becomes the constraint and the number of intervals selected increases.

7.3.2.1 Extending Intervals to Runs

As outlined in Section 7.2.1.1, when multiple users are accessing the same object, a set of adjacent intervals could be grouped together to form *runs* and considered for caching. When the bandwidth and space of the cache is limited, it is profitable to group intervals together to amortize the write overhead over multiple readers and use less space. Figures 7.7(a,b) compares the hit ratios yielded by two versions of the RBC caching algorithm: one that considers only intervals and the other that considers runs (or groups of intervals). When the bandwidth is limited (32MB/sec), the RBC algorithm which caches runs of intervals yields around 25% higher

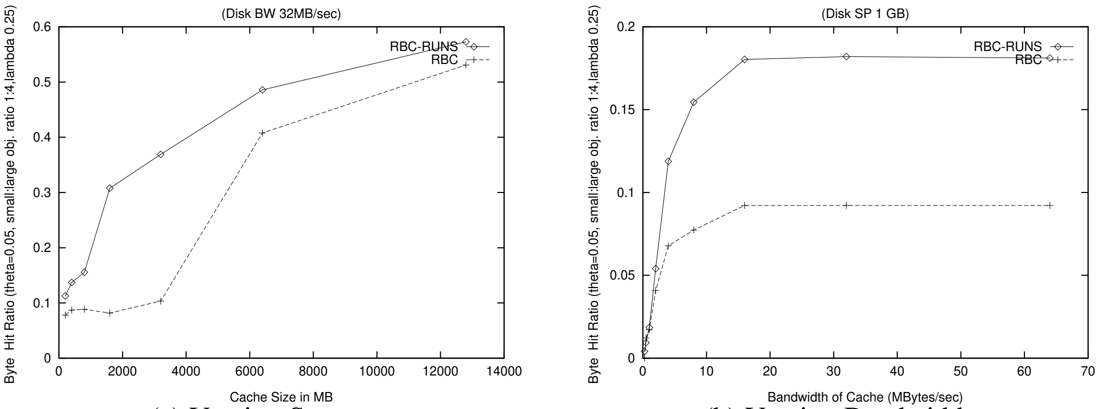
[6]The intervals selected by RBC define the granularity of the entity selected for caching and should not be confused for generalized interval caching [30]



(a) Varying Space (b) Varying Bandwidth

Figure 7.6: Analysis of the RBC algorithm. In both figures the large object size ranges from 5MB-1GB and the small objects range from 125KB to 1MB. The access rates are uniformly distributed between 0 to 6 Mb/sec.

hit ratios (see Figure 7.7(a)). On the other hand, in Figure 7.7(b) when the cache size is fixed (1GB), RBC with run caching improves the hit ratio by around 70%. This is due to the improved bandwidth usage for a much lower space usage compared to entire objects or intervals. For the following experimental results, by default, RBC caching includes runs unless otherwise stated.



(a) Varying Space (b) Varying Bandwidth

Figure 7.7: Effect on run caching on the performance of RBC algorithm. In both figures the large object size ranges from 125MB-1GB and the small objects range from 5MB to 100MB. The access rates are uniformly distributed between 2 to 6 Mb/sec.

7.3.3 Comparison of RBC and other Algorithms

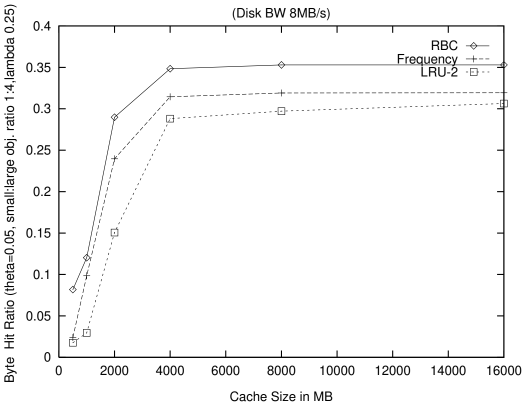
To evaluate the performance of RBC, we compare it with the most commonly used algorithms, LRU-2, frequency and LRU-MIN that base their replacement policy on recency, frequency and object size respectively. The algorithms are compared with varying cache space and bandwidth for different performance metrics (e.g, byte hit ratio and hit ratio), and different object types (continuous media and mixed text and continuous media).

7.3.3.1 Continuous Media Objects

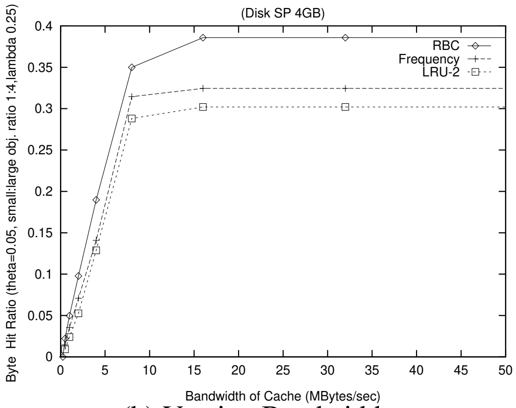
To evaluate the performance of RBC, we compare its BHR values with other traditional caching algorithms. In this comparison the workload is restricted to CM objects of varying size and bandwidth requirements; the granularity consists of entire objects, intervals and runs. The RBC algorithm is compared with (i) frequency-based caching of entire objects using WLRU-n, and (ii) recency-based caching using LRU-2. Whereas the size of the cache was varied from 1 to 1024 times the size of the smallest object in the system, and the bandwidth of the cache was varied from 1 to 128 times the mean data rate requirement of the objects[7]. The default data rate requirement of all the CM objects was assumed to range from 2-6 Mb/sec. The sizes of the short and long were chosen to range from 5MB-100MB and 125MB-1GB, respectively (corresponding to MPEG-2 compressed video streams of duration 20 seconds to 1 hour, respectively). The access skew among objects was assumed to be Zipf with $\theta = 0.05$, and the ratio of accesses between small and large CM objects was set to 1:4.

Figure 7.8(a) and 7.8(b) show the variation in the byte hit ratios for the different cache algorithms with varying cache size (with fixed disk bandwidth 8MB/sec) and varying cache bandwidth (with fixed space 4GB), respectively. They demonstrates that, once the cache bandwidth (space) saturates, the byte hit-ratio reaches a plateau, and is unaffected by further increase in cache size (bandwidth). Figure 7.8(a) shows that for a fixed cache bandwidth, when the space increases, the bandwidth becomes a constraint. The RBC algorithm forms an envelope over both recency and frequency caching algorithms, with around 20% higher hit ratio.

[7]Note that the space and bandwidth values assumed for cache are not representative of a single disk but that of a 'logical disk cache' that could be mapped to a disk array.



(a) Varying Cache Size



(b) Varying Bandwidth

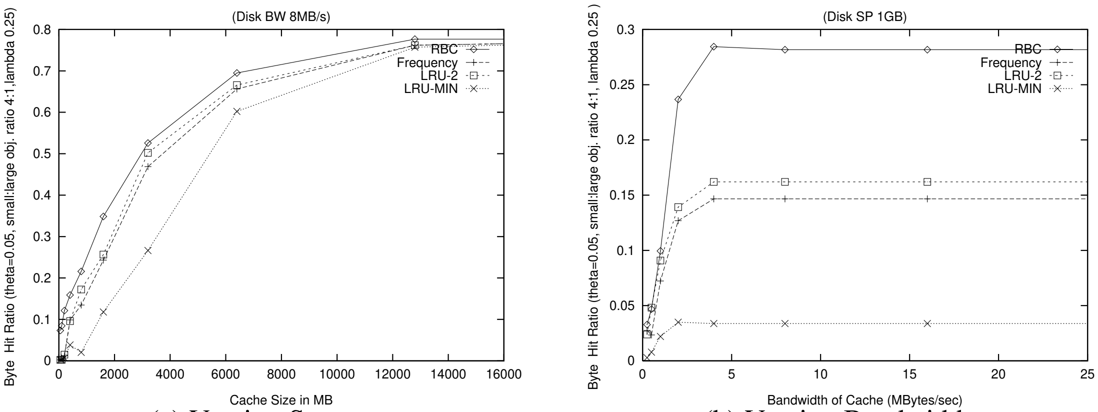
Figure 7.8: Comparison of caching algorithms - a snapshot. In both figures the large object size ranges from 125MB-1GB and the small objects range from 5MB to 100MB. The access rates are uniformly distributed between 2 to 6 Mb/sec.

Figure 7.8(b) shows that for a fixed cache size, when the bandwidth increases, the cache space becomes the constraint, and the hit ratios saturate; the RBC algorithm remains an envelope of either policies (atleast 29% higher than all the other policies).

In the space constrained case, for the objective of maximizing BHR, the goodness value maps to inverse of TTR, which is identical to the ordering used by the other algorithms. The main difference being the computation of TTR and the selection of entity granularity. Only RBC selects the granularity of the cached CM entities based on the space utilization of the cache, selecting either intervals, runs or entire objects. In the bandwidth constrained case, RBC selects for replacement entities that have a large write overhead while other algorithms use the TTR ordering. RBC also distinguishes between objects that are being written to the cache and the ones that are already in the cache. Consequently, RBC has much higher hit ratios compared to the other policies when both space and bandwidth are limited.

7.3.3.2 Mixed Continuous and Non-continuous Media

In the previous experiments the workload was restricted to CM objects that had both space and bandwidth requirements. Non-CM objects on the other hand, do not require any retrieval rate guarantees, and hence do not reserve any bandwidth. Figures 7.9(a,b) compare the byte hit



(a) Varying Space (b) Varying Bandwidth

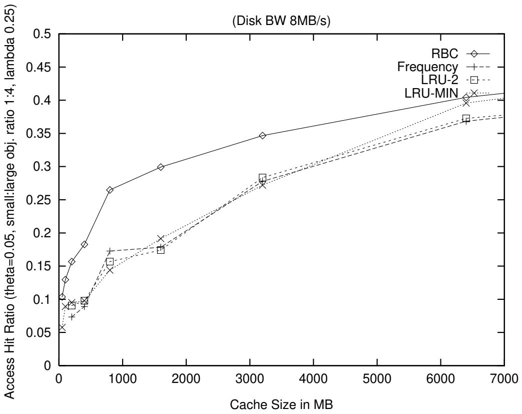
Figure 7.9: Mixing non-CM and CM data. In both figures the large object size ranges from 5MB-1GB and the small objects range from 125KB to 1MB. The access rates are uniformly distributed between 0 to 6 Mb/sec.

ratios yielded by the LRU-MIN, LRU-2, frequency-based and the RBC algorithm for mixed CM and non-CM objects. LRU-MIN was selected as a representative web caching algorithm that considers the time of last access and the size of the objects. Since interval caching is not suited for non-CM objects it was not used for the comparison. For this experiment, the sizes of non-CM objects were assumed to range from 100KB to 1 MB, while the sizes of CM objects ranged from 5MB to 1 GB. The data rate requirement for CM objects was assumed to vary in the range 2-6Mb/s. The ratio of accesses is 80% for non-CM objects and 20% for CM objects.

As Figure 7.9(a) demonstrates, RBC caching algorithm yields byte higher hit ratios (10-50%) than the LRU-2 and frequency-based algorithm. For the space constrained case shown in figure 7.9(b) RBC yields a byte hit ratios that are 6 times higher than LRU-MIN and 80% higher than frequency of recency based caching. Note that while the number of references to CM objects are much smaller, they contribute much more to the byte hit ratio due to their size. However, since LRU-MIN deletes all objects larger than the one to be inserted in LRU order, it has the poorest performance. Since the RBC algorithm considers the cache resource available, it performs better than LRU-2 or frequency-based algorithms.

The mixing of non-CM objects with CM objects does not affect the behavior of the algorithms when bandwidth is not a constraint. When bandwidth is a constraint, however, the performance of other algorithms is much poorer compared to RBC. Since other algorithms do not consider the bandwidth occupied by objects, they may unnecessarily replace non-CM objects which do not release any reserved bandwidth.

7.3.3.3 Hit Ratio as Performance Objective



(a) Varying Space (b) Varying Bandwidth

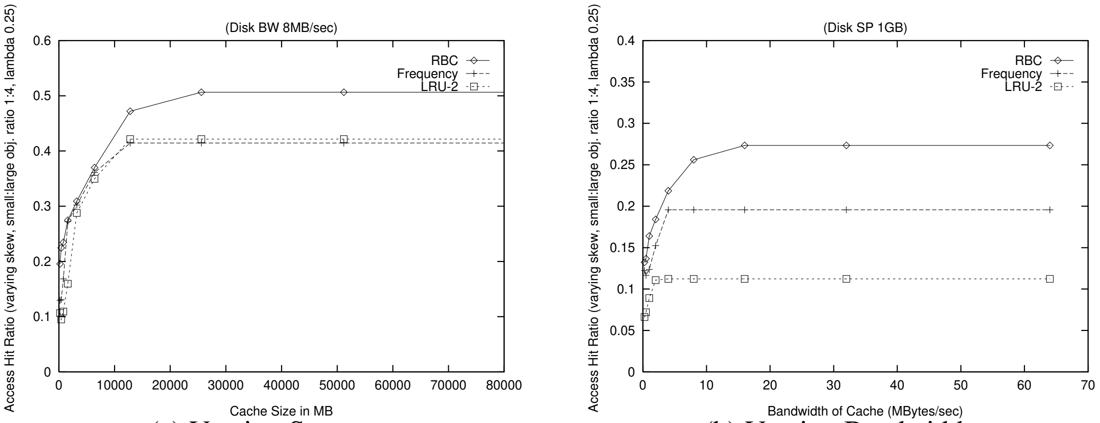
Figure 7.10: Performance Objective as Access Hit Ratio. In both figures the large object size ranges from 5MB-1GB and the small objects range from 125KB to 1MB. The access rates are uniformly distributed between 0 to 6 Mb/sec.

In the previous experiments the performance objective was chosen to be the byte hit ratio. It was shown in Section 7.2.1.3, that for BHR the ordering in the space constrained scenario mapped to the inverse of the time-to-reaccess. Since the object size contributes proportionately to the byte hit ratio, the space usage of the object was not considered. When the cache objective is to maximize hit ratio (based on the number of accesses) the goodness value of entities in the RBC algorithm maps to the access rate per unit resource used. Figure 7.10 compares the hit ratio of RBC, frequency caching, LRU-2 and LRU-MIN. RBC has much higher hit rate values compared to these algorithms (36%). This is due to the fact that, unlike BHR, a large sized object uses more cache space, limiting the number of objects in the cache. None of the other algorithms considered the space and bandwidth usage of the entities to be cached. Figure 7.10(a) shows the bandwidth constrained case (BW limited to 8MB/sec) for varying space where the RBC algorithm performs 36% better than LRU-2 and the frequency-based algorithm.

LRU-MIN shows a better performance compared to LRU-2 and frequency-based caching since it deletes the larger size objects first. Figure 7.10(b) depicts the space constrained scenario (space limited to 1GB) for varying bandwidth. The RBC algorithm achieves a 90% higher hit ratio compared to the other algorithms.

7.3.4 Effect of Temporally Changing Skew

In the previous experiments the access distribution among the objects (Zipfian with a skew parameter θ) remained constant. However, it has been observed in the different web server access logs that the hotness of objects varies with time. The popularity of objects sometimes varies with the time-of-day. In order to capture the varying popularity of an object with time, the access distribution in Figure 7.11(a,b) followed multiple Zipfian distributions. The objects that were hot in the initial reference sequence gradually become cold. The access hit ratio of the RBC, frequency and LRU-2 algorithms is shown in Figure 7.11. In this experiment the granularity for RBC is restricted to entire objects and interval. The RBC algorithms has 20-35% higher hit ratio compared to the other schemes. This is due to the fact that RBC balances both recency and frequency information in determining the TTR values compared to frequency or recency based algorithms.

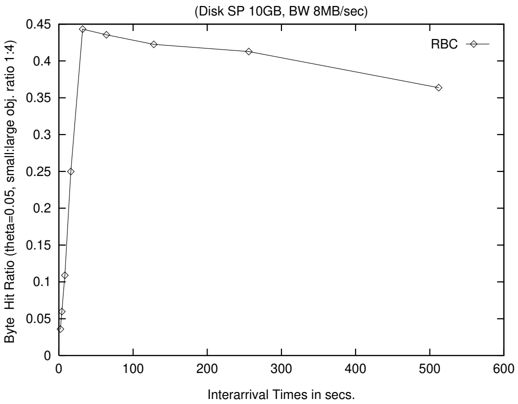


(a) Varying Space (b) Varying Bandwidth

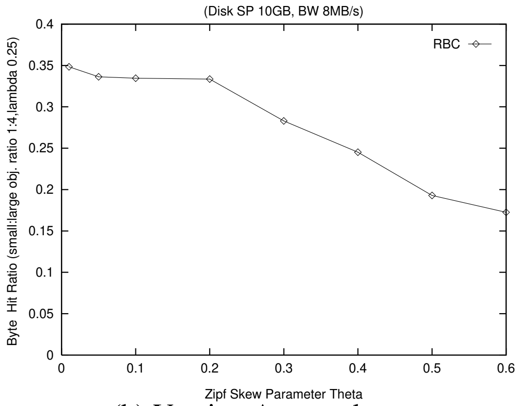
Figure 7.11: Varying skew with time. In both figures the large object size ranges from 5MB-1GB and the small objects range from 125KB to 1MB. The access rates are uniformly distributed between 0 to 6 Mb/sec.

7.3.5 Varying Workload Characteristics

The values of the performance objective, HR or BHR are also affected by the access skew and the request arrival rates. Figure 7.12(a) shows the effect of varying the request inter-arrival times. It demonstrates that very small inter-arrival times cause the cache to be in a constant state of upheaval resulting in thrashing and hence, low hit ratios. Increasing the inter-arrival time decreases the number of concurrent accesses to an object, and hence, increases the average interval length. This results in a smaller number of intervals to be cached, and hence, reduces the byte hit ratio. Figure 7.12(b) shows the effect of varying the access skew. Typically when the access skew is high a large fraction of the accesses are to a small set of objects and caching the "hot-set" will result in a large hit ratio. As the access skew becomes more uniform (with increasing θ), the byte hit ratio falls as there are fewer concurrent accesses.



(a) Varying Inter-arrival Times



(b) Varying Access skew

Figure 7.12: Effect of Workload. In both figures the large object size ranges from 125MB-1GB and the small objects range from 5MB to 100MB. The access rates are uniformly distributed between 2 to 6 Mb/sec.

7.4 Concluding Remarks

The WWW employs a hierarchical data dissemination architecture in which hyper-media objects stored at a remote server are served across a wide-area network to geographically distributed clients, and cached at intermediate proxy servers on-demand. To minimize the load imposed on the network and the occurrence of remote server hot-spots, cache management algorithms will be required to maximize the data transferred from the proxy or intermediate hierarchical caches. These cache algorithms differ from processor caches in that they handle multiple data types, each with different cache resource usage, for a cache limited by both bandwidth and space.

In this chapter, we presented a resource based caching algorithm that achieves these objectives. To do so, the algorithm: (1) selects the granularity of the entity to be cached based on the current cache resource usage; and (2) determines which, if any, of the entities present in the cache should be replaced. The procedure for determining the entities to be replaced is governed by the relative usage of the cache resources by the entity and their contribution to the cache performance objective. By judiciously selecting the entities to be replaced, the algorithm maximizes the optimization criteria for a cache with limited bandwidth and space. Through extensive simulations, we demonstrated that the RBC algorithm achieves higher hit ratios as compared to several existing algorithms under widely varying workloads and cache configurations.

Chapter 8

Conclusions

"Where shall I begin, please your Majesty?" he asked.

"Begin at the beginning," the King said, gravely,

"and go on till you come to the end: then stop."

—Lewis Carroll, Alice's Adventures in Wonderland

In this dissertation we designed an infrastructure for a scalable wide-area information system with support for managing a large number of heterogeneous objects, maintaining low response times and guaranteeing high availability. To achieve the above goal we have developed, designed and implemented: (i) scalable clustered servers and (ii) a scalable distributed caching architecture.

8.1 Summary of Contributions

Current web servers were built using single node systems, providing a best-effort service for different types of objects with no availability guarantees to the user. This approach violated the basic design principles for scalable systems. Our scalable server design, based on *clustered servers*—a group of independent nodes connected by a scalable interconnect, has the following properties: (i) has no central resource, (ii) requires no global information or shared state,

(iii) supports incremental growth, (iv) balances load across various nodes/disks, (v) supports requirements of heterogeneous data types, and (vi) guarantees data availability.

To address the design principles for scalability we built techniques for data placement that: (i) *network stripe* the data across nodes for load balancing, and (ii) use a balanced random data placement to support incremental growth. Further, we designed an analytical multi-queue model of the clustered server. The model has two benefits. First, it was used to analyze architectural choices and determine the optimal striping parameters (e.g., stripe unit size, striping width, etc.) for the different data types. Second, it provided insights into the interactions of the interconnection network with the storage disks and its combined effect on the placement parameters. Our analysis showed that three factors affected the stripe unit size: (i) disk parameters (throughput and response time variance), (ii) interconnect bandwidth, and (iii) read-ahead buffer size. Disk throughput favors a large size to amortize seek and rotational latency overheads, but disk response time variance is lower for a smaller size. A slow interconnect favors a small size to reduce interconnect variance, and a fixed read-ahead memory size favors a small size to increase number of read-aheads. The optimal stripe unit size balances the above tradeoffs.

To load balance the client connections across the nodes, the existence of multiple nodes should be transparent to the client. Conventional approaches balance the number of client connections across nodes, either by: (i) providing a list of node names to the client, or (ii) resolving a single name to multiple addresses. These approaches have two limitations: (i) they are susceptible to node failures, and (ii) they provide poor load balancing. To achieve fine-grained load balancing, our approach provided a single name/address to the client and transparently routes the data to different nodes. The novel aspect of this technique is that it is limited to control-path routing (from the client to server) while the data path (from the server to the clients) circumvents the router. This eliminates router bottlenecks.

Increasing the number of nodes and disks in a cluster increases the overall probability of failure. To guarantee availability of data, redundant information is stored; either a replica of each block or parity information for a group of blocks (e.g., RAID). Existing parity placement techniques do not scale to large number of users. We design parity placement schemes—random parity and multi-stride parity placement, that have the following properties: (i) balance the during normal and failure mode operations, (ii) minimize the storage and computation overhead, (iii) support optimizations for different data types, (iv) scale to a large number of users. To determine the optimal parity group size, for best performance, we extend our analytical multi-queue model to include failure mode operations.

Geographically distributed servers and bursty accesses increase network delays and congestion. This limits the system scalability and the performance observed by clients. Caching at intermediate nodes (called proxies) improves the scalability and performance by reducing: the network/server traffic, and the distance traversed on the network. We design and implement a distributed cache architecture that has the the following goals: (i) improve performance, (ii) increase scalability, and (iii) support different data types. To address these goals, our design consists of: distributed caches with local location hints, a self-configuring hierarchy for meta-data propagation, and cache replacement policies for multiple data types.

Hint Caches: Based on measurements of several caches on the Internet and analysis of several traces of web traffic, we derive five basic design principles for large-scale caches: (1) minimize the number of hops to locate and access data, (2) do not slow down misses, (3) share data among many caches, (4) cache data close to clients. Although these principles may seem obvious in retrospect, we find that current cache architectures routinely violate them at a significant performance cost. Using these principles, we design, simulate, and implement three strategies for organizing and managing large-scale, distributed caches that:

- Separates data paths from meta-data paths and maintains a hierarchy of meta-data that tracks where copies of data are stored.
- Maintains local location hints so that caches can locate nearby copies of data without suffering network latencies.
- Uses direct cache-to-cache data transfers to avoid store-and-forward delays inherent in conventional cache hierarchies
- Push caches data closer to the clients that have not yet referenced the data but are likely to do so in the near future.

We have added the protocol for maintaining location-hints to Squid version 1.1.20. There are three primary interface commands between Squid and the hint cache. One is an *inform* command, which tells the hint cache that a copy of a given object is now stored locally, and the hint cache should advertise it. The second is an *invalidate*, which tells the hint cache that that copy is no longer present; this causes the hint cache to advertise the non-presence, and to use the next best location (if one is known) of that object. The third is *find nearest*, which asks the hint cache to report the location of the nearest copy of an object, if one is known.

Experimental results show that by using the above strategies, the response times improves by a factor of 1.3 to 2.3 compared to conventional cache organizations.

Self-configuring meta-data Hierarchy: The goal of the self-configuring meta-data distribution hierarchy is to provide a fast and scalable way for a node to determine where to go for data that is not locally cached. Four aspects of the design improve scalability and performance: hierarchically propagating hints, pruning the propagation of location hints, limiting the number and size of local hints, and automatically configuring the hierarchy. The design and implementation of the algorithm for self-configuring the hierarchy improves scalability by: (i) automatically reconfiguring on failure, (ii) distributing load among the caches by creating different hierarchies for different object groups, (iii) maintaining locality of nodes within subtrees, and (iv) automatically configuring on incremental growth.

Cache replacement algorithms for heterogeneous data: There are two factors that affect proxy cache resource management. First, objects stored in the cache have different sizes and occupy different amounts of cache space. Second, continuous-media objects are accessed in streaming mode instead of traditional down-load-and-play, thereby require disk bandwidth reservations. Current proxy cache replacement algorithms, based on extensions of traditional memory caching, do not adapt to the resource requirements of heterogeneous data types. We design the *Resource-based Caching* (RBC) algorithm that makes replacement decisions based on: the resource usage of different objects and the benefit they add toward improving cache performance. The key aspect of RBC is selecting the unit of object to cache and determining what to replace based on the available cache resource. The idea is to model the cache as a two-constraint knapsack; use heuristics to convert the two-constraint knapsack to multiple single constraint knapsacks; use the gain density approximation to solve a single constraint knapsack. We show that the performance of RBC is always an envelope of various other cache algorithms, ranging from 10-200% improvement for varying access parameters and cache capacities.

8.2 Future Directions

In this dissertation, we addressed issues for designing scalable servers that support heterogeneous data types, and a scalable distributed cache architecture for delivery. However, several more issues need to be resolved that present avenues for future research.

The performance improvement due to caching in a scalable delivery architecture is lowered by poor consistency management. Efficient consistency management reduces communication or validity misses. In the current web environment, consistency is achieved through client polling. Conversely, in traditional distributed filesystems, consistency is handled entirely by the server (e.g., by call-back mechanisms). While the former is not suitable for applications with frequent updates and shared objects, the latter does not scale for large systems. Other solutions like *Leases* let consistency be the responsibility of the server, but guarantee it only for a limited time period. The consistency requirement of an application depends on the percentage of updates, the scale of the application (number of users) and the number of writers.

Currently web clients navigate multiple servers for information through search engines and repeatedly poll servers to get up-to-date information. Although techniques like *if-modified-since* requests and information channels reduce the amount of data transferred and automate the polling process, they do not do true push. The publisher/subscriber model is one step towards enabling subscribers to get the information they need without polling. We would like to extend the application of push caching algorithms for distributed proxy caches using the publisher-subscriber model.

When pushing information, security is an important concern for wide-area applications. In our design we have implicitly assumed trusted agents or the ability to authenticate and verify information. In future we would like to extend our design of a scalable wide-area information system to address the above mentioned issues.

Bibliography

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] M. Arlitt and C. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996. http://www.cs.usask.ca/projects/discus/discus_reports.html.

[3] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of SIGMETRICS 96*, pages 126–137, May 1996.

[4] C.R. Attanasio, M. Butrico, C.A. Polyzois, S.E. Smith, and J.L. Peterson. Design and Implementation of a Recoverable Virtual Shared Disk. IBM Research Report RC 19843, T.J Watson Research Center, Yorktown Heights, New York, 1994.

[5] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[6] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered striping in multimedia information systems. *Proceedings of the 5th SIGMOD*, May 1994.

[7] S. Berson, L. Golubchik, and R. Muntz. Fault tolerant design of multimedia servers. *Proceedings of the 6th SIGMOD*, May 1995.

[8] M. Blaze. *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

[9] M. Blaze and R. Alfonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proceedings of International Conference on Distributed Computing Systems*, June 1992.

[10] J. Bolot and P. Hoschka. Performance Engineering of the World Wide Web. In *Proceedings of WWW Journal, available online at http://www.w3.org/pub/WWW/Journal/3/s3.bolot.html*, pages 185–195, Summer 1996.

[11] W. Bolsoky and et. al. The Tiger Video Fileserver. *Proceedings of NOSSDAV*, April 1996.

[12] M. M. Buddhikot, G. M. Parulkar, and J.R. Cox. Design of a large scale multimedia storage server. *Journal on Computer Networks and ISDN Systems*, 1995.

[13] B. Callaghan. Webnfs: The file system for the internet. Technical report, Sun Microsystems Whitepaper, April 1997.

[14] P. Cao, E. Felten, and K. Li. Implementation and Performance of Application Controlled File Cache. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[15] V. Cate. Alex– a global filesystem. *Proceedings of the Usenix Filesystem Workshop*, May 1992.

[16] J. Chandy and N. A. L. Reddy. Failure evaluation of disk array organizations. *Proceedings of ICDCS*, May 1993.

[17] Chran-Ham Chang and et.al. High performance TCP/IP and UDP/IP networking in DEC/OSF/1 for alpha AXP. *Proceedings of the 3rd IEEE Symposium on High Performance Distributed Computing*, pages 36–42, August 1994.

[18] E. Chang and A. Zakhor. Scalable video data placement on parallel disk arrays. *Proceedings of Storage and Retrieval for Image and Video Databases II*, February 1994.

[19] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[20] H.-J. Chen and T.D.C. Little. Physical storage organizations for time-dependent multimedia data. *Proceedings of the fourth international conf. on the foundations of data or ganizations and algorithms*, October 1993.

[21] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-Performance, Reliable Secondary Storage . *ACM Computing Surveys*, June 1994.

[22] R. Chillarege, S. Biyani, and J. Rosenthal. Measurement of failure rate in commercial software. *IBM Research Report, RC-19889*, December 1994.

[23] H. Chou and D.J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Proceedings of the 11th VLDB Conference*, 1985.

[24] Inktomi Corporation. The Economics of Large-Scale Caching. http://www.inktomi.com/Tech/EconOfLargeScaleCache.html, 1997. http://www.inktomi.com/Tech/EconOfLargeScaleCache.html.

[25] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150–160, May 1994.

[26] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[27] M. D. Dahlin, R. Wang, T. E. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance1994. In *Proceedings of the Operating Systems Design and Implementation Symposium*, 1994.

[28] A. Dan, D. Dias, R. Mukherjee, D. Sitaram, and R. Tewari. Buffering and caching in large scale multimedia servers. In *Proceedings of IEEE COMPCON*, pages 217–224, March 1995.

[29] A. Dan and D. Sitaram. Buffer Management Policy for an On-Demand Video Server. IBM Research Report RC 19347, T.J Watson Research Center, Yorktown Heights, New York, January 1993.

[30] A. Dan and D. Sitaram. A Generalized Interval Caching Policy for Mixed Interactive and Long Video Environments. In *IS&T SPIE Multimedia Computing and Networking Conference*, San Jose, CA, January 1996.

[31] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *ACM SIGMETRICS*, pages 143–152, May 1990.

[32] Digital Equipment Corporation. Digital's Web Proxy Traces. ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html, September 1996.

[33] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[34] A. Bestavros et. al. Application-Level Document Caching in the Internet. In *Proceedings of Workshop on Services and Distributed Environments*, June 1995.

[35] K. Andrews et. al. On second generation hypermedia systems. In *Proceedings of ED-MEDIA, World Conference on Educational Multimedia and Hypermedia*, June 1995.

[36] M. Abrams et. al. Caching proxies: Limitations and potentials. In *4th International World-Wide Web Conference*, pages 119–133, December 1995.

[37] M. Feeley et. al. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[38] R. H. Patterson et. al. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[39] S. Williams et. al. Removal policies in network caches for world-wide web documents. In *ACM SIGCOMM*, pages 293–305, August 1996.

[40] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 201–212, December 1995.

[41] S. Floyd and V. Jacobson. The Synchronization of Periodic Routing Messages. *IEEE Transactions on Networking*, 2(2):122–136, April 1994.

[42] A. Fox, S. D. Gribble, Y. Chawathe, and Eric A. Brewer. Cluster-based scalable network services. *Proceedings of Symposium on Operating Systems Principles*, October 1997.

[43] C. Freedman and D. DeWitt. The SPIFFI scalable video-on-demand server. *Proceedings of SIGMOD*, June 1995.

[44] A.H. Frey and R.C. Mosteller. File-based redundant parity protection in a parallel computing system. U.S. Patent, (5130992), 1990.

[45] S. Gadde, J. Chase, and M. Rabinovich. Directory Structures for Scalable Internet Caches. Technical Report CS-1997-18, Duke University Department of Computer Science, November 1997.

[46] S. Gadde, M. Rabinovich, and J. Chase. Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 93–98, May 1997.

[47] S. Ghandeharizadeh and et. al. MITRA: A Scalable Continuous Media Server. *Journal of Multimedia Tools and Applications*, January 1997.

[48] S. Ghandeharizadeh and S. H. Kim. Striping in Multi-disk Video Servers. *Proceedings of SPIE High Density Data Recording and Retrieval Technologies conference*, Oct 1995.

[49] S. Ghandeharizadeh, S. H. Kim, and Shahabi. C. On Configuring a Single Disk Continuous Media Server. *Proceedings of ACM SIGMETRICS*, 1995.

[50] G. A. Gibson, J.S. Vitter, and J. Wilkes. Storage and I/O Issues in Large-Scale Computing. *ACM Workshop on Strategic Directions in Computing Research, ACM Computing Surveys*, 1996. http://www.medg.lcs.mit.edu/doyle/sdcr.

[51] Steve Glassman. A Caching Relay for the World Wide Web. In *Proceedings of the Third International World Wide Web Conference*, pages 69–76, May 1994.

[52] J. Goldick, K. Benninger, W. Brown, C. Kirby, C. Maher, D. Nydick, and B. Zumach. An AFS-Based Supercomputing Environment. In *Proceedings of the Thirteenth Symposium on Mass Storage Systems*, pages 127–132, April 1993.

[53] S. Gribble. UC Berkeley Home IP HTTP Traces, July 1997. Available at http://www.acm.org/sigcomm/ITA/.

[54] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[55] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[56] D. Gross and C. M. Harris. *Fundamentals of Queueing Theory*. John Wiley and Sons, 1985.

[57] J. Gwertzman. Autonomous Replication in Wide-Area Networks. Masters thesis, Harvard University, April 1995. Available as Tech Report TR-17-95.

[58] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[59] Roger Haskin. Personal Communication, 1994.

[60] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. *Proceedings of ASPLOS*, 1992.

[61] M. Holland, G. A. Gibson, and D. P. Siewiorek. Fast, on -line failure recovery in redundant disk arrays. *Proceedings of FTCS*, 1993.

[62] M. Holland, G. A. Gibson, and D. P. Siewiorek. Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays. *Journal of Distributed and Parallel Databases*, July 1994.

[63] J.H Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *SIGOPS*, pages 1 – 33, 1987.

[64] J. Hsieh, M. Lin, J.C.L. Liu, David D.C. Du, and T. M. Ruwart. Performance of a mass storage system for video-on-demand. *Proceedings of INFOCOM-1995*, March 1995.

[65] J. Hsieh et.al. Performance of a mass storage system for video-on-demand. *Journal of parallel and distributed computing*, submitted.

[66] C. Kesselman I. Foster. Globus: A metacomputing infrastructure toolkit. *Proceedings of the Workshop on Environments and Tools*, August 1996.

[67] VXtreme Inc. Server-less Versus Server-based Video Streaming. In *available online at http://www.vxtreme.com/developers/wp970101.html*, 1997.

[68] S. Irani. Page Replacement with Multi-Size Pages and Applications to Web Caching. In *Proceedings of the Twenty-ninth ACM Symposium on Theory of Computing*, 1997.

[69] F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor Group Membership Protocols: Specification, Design and Implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 2 – 11, Princeton, NJ, October 1993. IEEE Computer Society.

[70] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of VLDB*, 1994.

[71] J. Kay, M. Dahlin, R. Tewari, and H. Vin. Introducing the Pushcache. June 1998.

[72] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[73] T. Kroeger, D. Long, and J. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[74] G. Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation*. PhD thesis, The University of California at Los Angeles, 1997.

[75] Thomas T. Kwan, Robert E. McGrath, and Daniel A. Reed. Ncsa's world wide web server: Design and performance. *IEEE Computer*, pages 68 – 74, November 1995.

[76] T.T. Kwan, R. E. McGrath, and D. A. Reed. User Access Patterns to NCSA's World Wide Web Server. Dept. of Computer Science Research Report available online at http://www-pablo.cs.uiuc.edu/Papers/WWW.ps, University of Illinois at Urbana-Champaign, February 1995.

[77] A. Laursen, J. Olkin, and M. Porter. Oracle Media Server: Providing Consumer Based Interactive Access to Multimedia Data. *Proceedings of SIGMOD*, May 1994.

[78] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU Replacement Policy: A spectrum of block replacemnet policies . In *Seoul National University Technical Report SNU-CE-AN-96-004*, March 1996.

[79] M. Lewis and A. Grimshaw. The core legion object model. *Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[80] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[81] C. Maltzahn, K. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[82] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accomodate User-level Page Replacement Policies. In *Proceedings of Usenix Mach Workshop*, pages 17–29, 1990.

[83] P. Mockapetris and K. Dunlap. Development of the Domain Name System. *Computer Communications Review*, 18(4):123–133, August 1988.

[84] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your cache ain't nuthin' but trash. In *Proceedings of the Winter 1992 USENIX Conference*, pages 305–313, January 1992.

[85] D. Neal. The Harvest Object Cache in New Zealand. In *Proceedings of the Fifth International World Wide Web Conference*, May 1996.

[86] NetCraft. Web Server Survey. URL < http://www.netcraft.com >, 1998.

[87] V.F. Nicola, A. Dan, and D.M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proceedings of ACM SIGMETRICS*, pages 35–46, 1992.

[88] E. J. O'Neil, P. E. O'Neil, and Gerhard Weikum. The LRU-k page replacement algorithm for database disk buffering. In *Proceedings of International Conference on Management of Data*, May 1993.

[89] B. Ozden, R. Rastogi, and A. Silberschatz. Buffer replacement algorithms for multimedia storage systems. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 172–180, June 1996.

[90] B. Ozden, R. Rastogi, and A. Silberschatz. Disk Striping in Video Server Environments. *Proceedings of International Conference on Multimedia Systems and Computing*, June 1996.

[91] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 22–36, July 1996.

[92] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*. Prentice Hall, 1982.

[93] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Chicago, May 1988.

[94] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 79–95, December 1995.

[95] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: Replicated database services for world-wide applications. *SIGOPS European Workshop*, 1996.

[96] J.E. Pitkow and M. M. Recker. A simple yet robust caching algorithm based on dynamic access patterns. In *Proceedings of 2nd International WWW Conference*, pages 1039–1046, October 1994.

[97] C. Plaxton, R. Rajaram, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[98] Attanasio C. R. and Smith S.E. (a virtual multiprocessor implemented by an encapsulated cluster of loosely coupled computers). *IBM Research Report RC18442*, 1992.

[99] N. A.L. Reddy and J. Wyllie. I/O Issues in a Multimedia System. *IEEE Computer*, 27(3):69 – 74, March 1994.

[100] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of ACM SIGMETRICS*, May 1990.

[101] A. Rousskov. On Performance of Caching Proxies. http://www.cs.ndsu.nodak.edu/ rousskov/ research/cache/squid/profiling/papers, 1996.

[102] L. A. Rowe, D. A. Berger, and J. E. Baldeschwieler. The Berkeley Distributed Video-on-Demand System. *Proceedings of Sixth NEC Research Symposium*, 1996.

[103] L.A. Rowe and B.C. Smith. A continous media player. *Proceedings of the 3rd Intl. NOSSDAV workshop*, November 1992.

[104] H. Sandhu and S. Zhou. Cluster-Based File Replication in Large-Scale Distributed Systems. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 91–102, June 1992.

[105] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of Operating Systems Design and Implementation Conference*, October 1996.

[106] M. Satyanarayanan. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering*, January 1992.

[107] M. Schwartz. Formal Service Agreements for Scaling Internet Caching. In *NLANR Web Cache Workshop*, June 1997. http://ircache.nlanr.net/Cache/Workshop97/Papers/Schwartz/schwartz.html.

[108] N. Smith. The UK National Web Cache: A State of the Art Report. In *Proceedings of the Fifth International World Wide Web Conference*, May 1996.

[109] H. Takagi. *Queueing Analysis*, volume 1. North Holland, 1991.

[110] R. Tewari, D. Dias, R. Mukherjee, and H. Vin. Design and Performance Tradeoffs in Clustered Video Servers. *Proceedings of ICMCS*, June 1996.

[111] F. Tobagi, J. Pang, R. Baird, , and M. Gang. Streaming raid- a disk array management system for video files. *Proceedings of ACM Multimedia*, 1993.

[112] Gene Trent and Mark Sake. WebSTONE: The First Generation in HTTP Server Benchmarking. URL < http://www.sgi.com/Products/ WebFORCE/WebStone/paper.html >, February 1995.

[113] A. Vahdat and T. Anderson. Transparent Result Caching. In *Proceedings of the 1998 USENIX Technical Conference*, June 1998. To appear.

[114] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. Webos: Operating system services for wide area applications. Technical report, University of California, Berkeley, UCB CSD-97-938, March 1997.

[115] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating System Services for Wide Area Applications. Tech Report CSD-97-938, U.C. Berkeley Computer Science Division, March 1997.

[116] V. Valloppillil and K. Ross. Cache Array Routing Protocol v1.0. IETF Internet Draft, October 1997. http://www.etext.org/Internet/Internet-Drafts/draft-vinod-carp-v1-02.txt.

[117] M. van Steen, P. Homburg, and A.S. Tanenbaum. The architectural design of globe: A wide-area distributed system. Technical report, Vrije Universiteit, IR-422, March 1997.

[118] H.M. Vin and P.V. Rangan. Designing a multiuser HDTV storage server. *IEEE journal on selected areas of communications*, January 1993.

[119] H.M. Vin, S.S. Rao, and P. Goyal. Optimizing the placement of multimedia objects on disk arrays. *Proceedings of the Second IEEE International Conference on Multimedia Computing and Systems, Washington, D.C.*, pages 158–165, May 1995.

[120] H.M. Vin, P.J. Shenoy, and S. S. Rao. Efficient failure recovery in multi-disk multimedia servers. *Proceedings of FTCS*, June 1995.

[121] wais. Wide Area Information Servers. URL < http://www.think.com >, 1998.

[122] D. Wessels. Squid Internet Object Cache. In *available online at http://squid.nlanr.net/Squid*, May 1996.

[123] D. Wessels. Squid Internet Object Cache. http://squid.nlanr.net/Squid/, January 1998.

[124] D. Wessels and K. Claffy. Application of Internet Cache Protocol (ICP), version 2. Request for Comments RFC-2186, Network Working Group, 1997. http://ds.internic.net/rfc/rfc2187.txt.

[125] Network Wizards. Internet Domain Survey. URL < http://www.nw.com >, 1998.

[126] R. Wooster and M. Abrams. Proxy Caching that Estimates Page Load Delays. In *Proceedings of Sixth International WWW Conference, available online at http://proceedings.www6conf.org/HyperNews/get/PAPER250.html*, April 1997.

[127] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the Eightteenth International Conference on Distributed Computing Systems*, May 1998.

[128] P. Yu, M. S. Chen, and D. D. Kandlur. Design and Analysis of a Grouped Sweeping Scheme for Multimeida Storage Management. *Proceedings of Third International NOSSDAV Workshop*, Nov 1992.

[129] L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching. In *Proceedings of the NLANR Web Cache Workshop*, June 1997. http://ircache.nlanr.net/Cache/Workshop97/.