Global Memory Management for Multi-Server Database Systems

Shivakumar Venkatarman

Technical Report #1325

September 1996

GLOBAL MEMORY MANAGEMENT FOR MULTI-SERVER DATABASE SYSTEMS

By

Shivakumar Venkatarman

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

Abstract

Traditionally, database systems have been implemented using a client-server architecture. Under heavy loads, the server experiences both CPU and I/O bottlenecks. One promising solution to this problem is to replace a single processor server with a cluster of servers. The goal of this thesis is to develop buffer management algorithms that exploit the aggregate memory capacity in such a server cluster to attack the I/O bottleneck. The thesis addresses two key issues: 1) utilizing idle memory in the cluster, and 2) limiting data replication among the memories in the cluster to minimize intra-cluster network traffic and disk I/O, to minimize response time.

On the issue of utilizing idle memory in the cluster, we demonstrate the significant impact of data placement, namely, clustering and declustering, on how global memory is utilized. We design three page replacement algorithms, ClSv, Reserve, and Global, that are simple modifications to the client-server buffer management code. Through a simulation study using CSIM, and an implementation on the IBM SP/2, we evaluate the performance of the memory management policies that are combinations of the data placement and page replacement algorithms. Using synthetic workloads characteristic of those experienced by object databases, we demonstrate that when the data is declustered, simple memory management policies are sufficient to utilize the aggregate memory capacity of the server cluster. We show that Global combined with declustering gives the best performance.

To control data duplication in memory, we present a new algorithm, Hybrid, that dynamically controls the amount of duplication in global memory. We show that on workloads characteristic of those experienced by Web servers, the Hybrid algorithm correctly trades off intra-cluster network traffic and disk I/O to minimize the average response time.

Acknowledgements

I have been very fortunate to have had the opportunity to work with Prof. Jeff Naughton. He has been an outstanding Ph.d advisor. I thank him for his plentiful support and constant encouragement. His guidance and insightful suggestions were important in making my research interesting and exciting. I also had the privilege of working closely with Prof. Miron Livny. Several lively discussions with Miron provided me with a deep understanding of the various performance issues in database systems. These discussions always helped me overcome difficulties and develop better algorithms. I thank Prof. David DeWitt for being on my thesis committee, and for suggesting several improvements to the final draft of the thesis. I also extend my sincere thanks to Prof. Raghu Ramakrishnan and Prof. Teresa Adams for serving on my thesis committee.

There are several others I thank for supporting my Ph.d research. Mike Franklin's Ph.d thesis provided me with the initial background as well as ideas for my thesis topic. Mike Zwilling, C.K Tan, Nancy Hall, Josef Burger, and Dan Schuh were instrumental in developing SHORE, which was an excellent software platform for my implementation work. Mike Zwilling and Tan spent a lot of time teaching me the intricacies of the SHORE database system. They were quick to address and solve problems I had while using SHORE. Miron and the Condor team deserve special thanks for Condor, which made the simulation studies feasible. Markos Zaharioudakis was responsible for improving the server-server communication infrastructure in SHORE, which made my implementation task easier. The computer systems lab deserves credit for providing top notch computing support.

I would like to thank all the department secretaries, especially Lorene Webber, for her timely reminders of deadlines and for efficiently managing the paper work pertaining to the graduate school. Susan Dinan, Kathleen Comerford, and James were very efficient in dealing with the accounting department and arranging for quick travel reimbursements.

Joseph Albert, Jie-Bing Yu, Kurt Brown, Craig Freedman, Joseph Hellerstein, Navin Kabra, Mark McAuliffe, Manish Mehta, Karthik Ramasamy, Jignesh Patel, Viswanath Poosala, Srikant Ramakrishnan, Ambuj Shatdal, Praveen Seshadri, S. Seshadri, John Shafer, V. Srinivasan, Odysseas Tsatalos, Kristin Tufte, Janet Wiener, and several others, were always there to discuss new ideas, and made the computer science department a fun place to be. Lunch breaks were entertaining due to animated discussions with T.N Vijaykumar and Harish Patil. Overall, Madison was a great place to be, a home away from home, and several other people were also responsible for it: Ravi Vijayaraghavan, V.C Tirumalai, V.C Govindarajan, Ramanathan Santhanagopalan, Vinod John. A.C Narendran, Shyam Srinivas, Anand Varadachari, and several others.

Shanthi Adimoolam (my wife-to-be) deserves special credit for the encouragement and motivation in pursuing my research goals. Her love and constant support enabled me to pull through difficult times.

I thank my parents, Sarada and Venkataraman, sister Sripriya, and brother Ramkumar, for their love and support in everything I did.

I dedicate this thesis to my grandfather Prof. S. Lakshminarayanan; he has been my mentor, and a great source of inspiration in all my endeavors.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Traditionally, database systems are built using a centralized client-server architecture. Under heavy loads, the single server is a potential bottleneck. The database servers typically run on a Symmetric Multi-Processor (SMP), and the scaling abilities of an SMP based server is limited by the bus bandwidth that is approaching physical limits. Figure 1 illustrates the client-server architecture.

One solution that is employed to solve this problem is to split the database among many independent servers. The client is modified so that it processes data from multiple servers. An example of such an architecture is the client-server EXODUS [CDG+90]. In this architecture, the servers do not share their resources, nor do they communicate with each other. All the co-ordination necessary to run distributed transactions (2 phase commit, distributed logging, and distributed queries), takes place at the client. The client also has to be aware of changes to the server configuration. This gives rise to software complexity at the client. The transparency of the server architecture, obstacles to scaling due to load imbalances among the servers, and the software complexity at

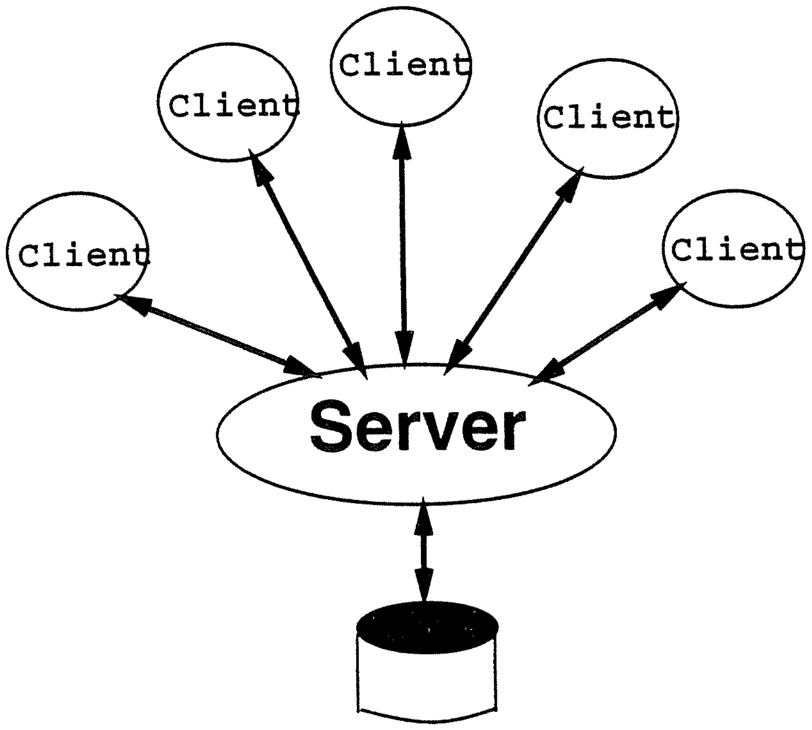

Figure 1: Client-Server Architecture with Single Server

the client, are some of the serious drawbacks of this architecture. Figure 2 shows this architecture.

An obvious solution that is increasingly becoming popular is to have a cluster of database servers on commodity processors, working closely together over a network as a database "super server". Building a system out of commodity parts, connected by a high speed network is an attractive solution, for its low cost due to large economies of scale, high scalability, and simplicity of the client software. An important advantage of this architecture is that, the complexity of dealing with multiple servers can be built into the servers rather than into the client. The details of the server architecture can be hidden from the client by providing the client with a single server interface to the

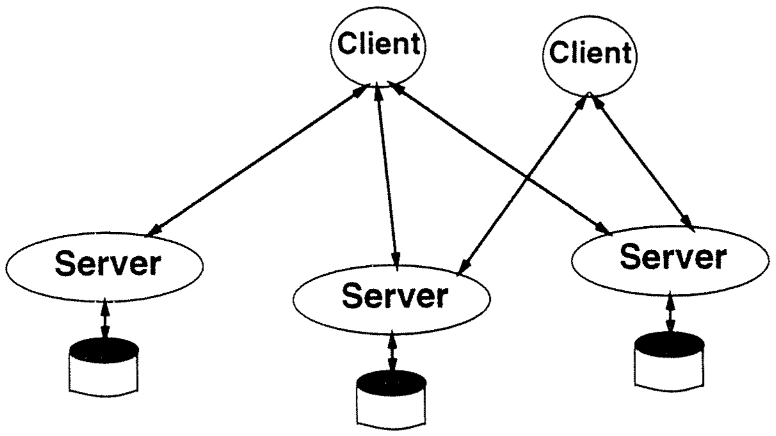

Figure 2: Client-Server Architecture with Multiple Servers

data at all the servers. An example of such a multi-server database system architecture is the Scalable Heterogeneous Object REpository [CDF+94] (SHORE) persistent object database system developed at the University of Wisconsin-Madison.

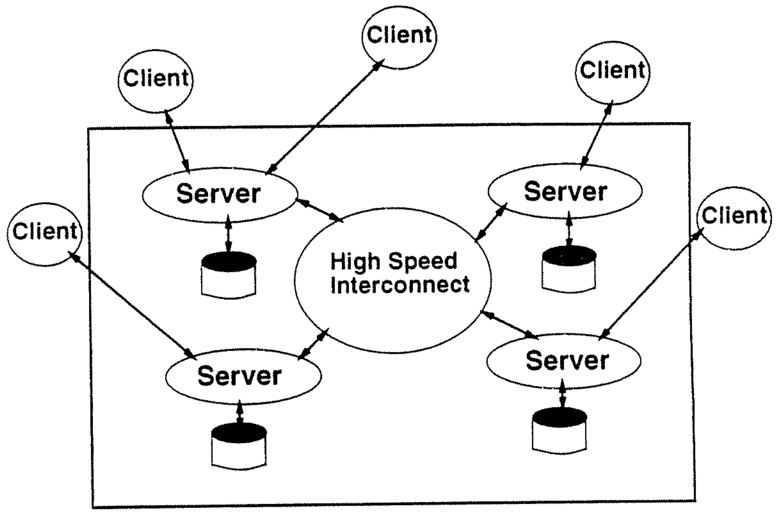

Figure 3: Multi-Server Architecture

SHORE employs a symmetric peer-peer communication architecture that facilitates replacing the single-server bottleneck with multiple servers. An important feature of this architecture is that, a client has access to the entire persistent object space through a single server. Figure 3 shows a multi-server architecture. The multi-server database system architecture is described in detail in Chapter 2. Table 1 summarizes some of the key architectural differences between a client-server architecture and a multi-server architecture.

Property	Client-Server	Multi-Server
Servers	Single or Multiple	Multiple
Scalability	Limited by Bus	High scaling potential
Communication	Bus	Switched Network
Memory	Centralized	Distributed
Nodes	Non commodity	Commodity
I/O architecture	Complex	Simple
Cost	Expensive	Low Cost
Client Complexity	High	Low
Availability	Low availability	Highly available
Resource Management	Centralized	Distributed

Table 1: Client-Server versus Multi-Server Architecture

1.1.1 Network Technology

One of the key factors that has accelerated this trend towards multi-server database system is the tremendous strides made with networking technology. Today, the availability of networks with very high bandwidths, low latencies, and at low cost, has made multi-server database system architectures feasible. For example, in a Myrinet [BCF+95] switched network, the link bandwidth is of the order of 140 Megabits(Mbits)/sec, and the round trip latency for transferring 8 kilobyte (KB) pages between two nodes is about 931 microseconds (μs). The prices of these networks are also very affordable, for example, the cost of a 8 port Myrinet switch is only 2,400 \$. The total cost for setting up the

Network Type	Machine Type	Software Protocol	Bandwidth Mbits/sec	Remote Mem 8 KB μs
Ethernet	Sparc 20	TCP/IP	10	6,700
ATM AN2	Alpha	DEC-ATM [TL93]	155	870
ATM AN2	Sparc 20	TCP/IP	100.2	-
Myrinet	Sparc 20	UIUC FM [PLC95]	140	551
Myrinet	Sparc 20	TCP/IP	85.2	-
SP/2	RS/6000 SP	MPI [GLS94]	360	480
SP/2	RS/6000 SP	TCP/IP	81.6	2,650

Table 2: Network Performance

network, including the network card, and the software, is only 1,500 $ per host connection. Table 2 illustrates the bandwidth and latency for these networks. The performance figures for the Asynchronous Transfer Mode (ATM) network is taken from the paper by Thekkath and Levy [TL93] and the Message Passing Interface (MPI) values for the IBM SP/2(Scalable POWERparallel2 Systems) [AMM$^+$95] are from the paper by Gropp et al. [GLS94]. We measured the bandwidth and the latency for other networks using the hardware and protocol specified in Table 2.

The bandwidths and latencies for the ATM and Myrinet switched networks are an order of magnitude better than the corresponding values for the Ethernet. Communication protocols have become simple in recent times due to the improvements in the network technology. The low error rates, ability to directly transfer data from user space to the network, reliable in order delivery of messages guaranteed by today's networks, has lowered the complexity of the communication protocols. This in turn has lowered the cost of messages. Table 2 compares the bandwidth of Transmission Control Protocol/Internet Protocol (TCP/IP) with those of the specialized protocols tailored to each network. Clearly, the bandwidth when TCP/IP is employed is lower than that of specialized network protocol. For example, the bandwidth of a Myrinet switch using Fast Messages(FM) [PLC95] from the University of Illinois is double that of TCP/IP on the same hardware.

Even though TCP/IP performs poorly when compared with the specialized network protocols, message performance using TCP/IP on today's network hardware is superior to that on the Ethernet. For example, the IBM SP/2 has a bandwidth of 81.6 Mbits/sec and remote memory access time of 2,650 μs for an 8 KB page. The Ethernet on the other hand, has a bandwidth of 10 Mbits/sec, and a remote memory access time of 6,700 μs.

The availability of high speed networks has made it feasible for a cluster of database servers to share, memory and disk resources over a network, and operate as a single database "super server". The low latency and high bandwidth networks enable servers to efficiently access data from remote memory instead of from the local disk. Utilizing remote memory becomes feasible, because the latency for remote memory access is an order of magnitude lower than accessing data from disk.

1.2 Memory Management

The improvement in network performance has not been matched by disks. The performance improvement of disks has been moderate. For example, the cost of a random disk I/O in a 30 MegaByte (MB) data set is approximately 10 ms on a raw disk of the SP/2, and the best sequential data transfer rate from disk to memory is about 8 MB/sec, while the cost of accessing an 8 KB page from remote memory on the SP/2 network is around 480 μs. The cost of accessing a page from remote memory is an order of magnitude faster than accessing it from the local disk. Future projections for disks and networks do not expect this trend to alter significantly.

Due to the enormous difference between local and remote memory access times, a critical factor that determines the performance of a multi-server database system is the effectiveness with which the aggregate memory is utilized. Efficiently managing distributed memory as a global resource will tremendously improve performance, as the aggregate memory in a multi-server is quite large. For example, in a cluster of 10 commodity database servers, with 100 MB of memory on each node, the total memory is 1 GigaByte (GB). Each server has 100 MB of local memory, and 900 MB of memory is accessible over the network for a small cost.

1.2.1 Memory Management Issues

Most of the research on distributed shared memory is devoted to the study of coherence and efficient access mechanisms to for the data distributed shared memory. An important aspect that has received very little attention until recently is the issue of efficiently utilizing the aggregate memory in a distributed memory system.

Since the memory in a cluster of database servers is fragmented among the servers, managing memory as a global resource is difficult. The fundamental hardship in managing memory that is distributed is the lack of global state information. It is difficult, complex, and expensive to maintain exact global state information at each server when memory is distributed. To maintain exact global state, a message has to be sent to all the servers every time a page is accessed or replaced at one server. Therefore, maintaining exact global state information, even with a high speed network, is expensive as it requires a large number of messages.

If the message cost is negligible, an ideal way of managing distributed memory would be to manage it as centralized memory, that is use LRU[1] replacement policy to manage aggregate memory. However, when message latencies are finite, implementing LRU policy on distributed memory is not practical, as the cost of maintaining exact global state information at each server is prohibitive.

In contrast, direct applications of client-server buffer management policy, where each server uses a "Least Recently Used" (LRU) replacement policy to manage its buffer pool is simple to implement, but results in sub-optimal performance. We refer to the policy where each server uses LRU replacement policy as the *client-server policy*. Memory load imbalance, and duplication of data among servers results in poor use of global memory. There is a need for managing memory as a global resource so that servers make use of remote memory to cache pages and reduce memory load imbalance, and avoid duplication, so that the global memory is used very efficiently.

Two key properties distinguish managing centralized memory using LRU and managing distributed memory with each server employing LRU. With centralized memory: 1)

[1]Throughout this thesis we assume that LRU is a good replacement strategy, given the fact that the detailed client access patterns are unknown. This is a reasonable assumption as LRU is used as the replacement strategy in the buffer pools of commercial database systems.

the page that gets replaced from memory is the LRU page, and 2) no page is duplicated in memory. By replacing the LRU page, hot pages are cached in memory and cold pages are discarded. By disallowing duplicates, a large portion of the database will be cached in memory, reducing disk I/Os. With distributed memory, neither of the above are true and this could lead to sub-optimal performance. The goal of this thesis is to duplicate these two qualities of centralized memory management on distributed memory so that distributed memory is managed efficiently. We address these two aspects in detail in the following sections.

1.2.2 Utilizing Idle Memory

If the aggregate memory is at a centralized location, and if LRU replacement policy is used to manage it, the memory will be fully utilized if there is sufficient demand for it. The page replaced from memory will be the least recently used page (we refer to the least recently used pages as idle pages). The belief is that, by replacing the LRU page, an idle page is replaced with a useful page. However, if memory is distributed, and if each server uses LRU policy to manage its memory (we refer to this policy as the client-server memory management policy), memory at some of the servers may not be utilized even though there is a need for it at other servers. For example, a page that is hot and that has been accessed recently may be discarded from memory at one server even though there is a page available at some other server that has not been accessed in a long time. This could hurt performance, because, pages that are idle remain in memory, while hot pages are discarded from memory. However, if some of the data is moved from servers with hot pages to replace cold pages at other servers, memory will be better utilized. The following example further illustrates the problem of unutilized memory.

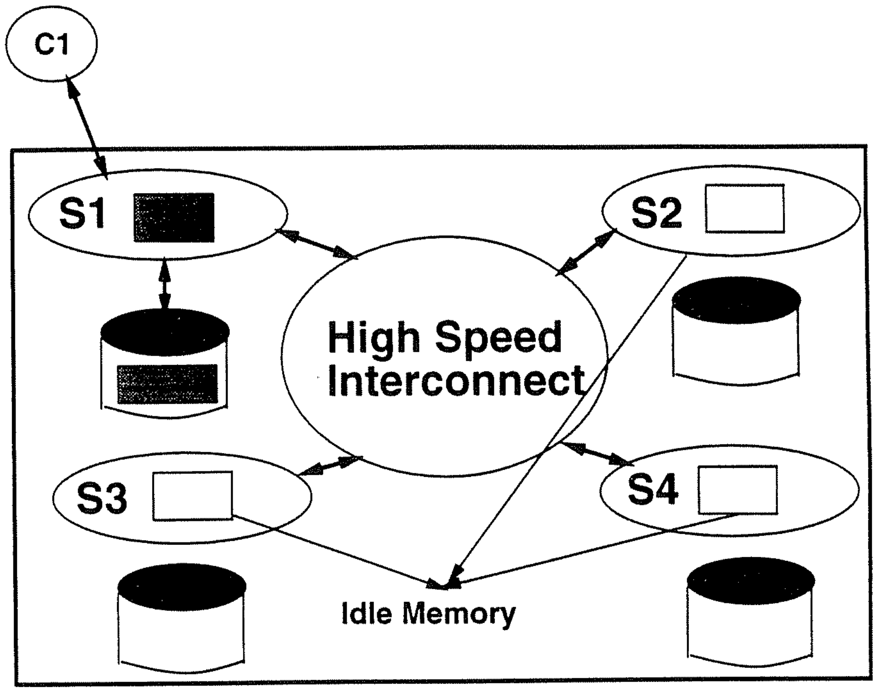

Figure 4: Utilizing Idle Memory

Figure 4 depicts a cluster of object database servers connected by a high speed network. Each server employs LRU to manage its memory. Object database clients, modeled after the OO7 object database benchmark [CDN93], execute methods that traverse pointers in a large object graph. The size of the object graph is typically larger than the size of memory of one server. C1 is one such client. When the client C1 connects to the server S1 that has the data accessed by the C1 on its local disk and executes such a method, I/Os are incurred at S1 even though there is idle memory available at servers S2, S3, and S4 to hold a larger portion of the data in memory. However, if the server

S1 uses a memory management policy that identifies idle memory at servers S2, S3, and S4, and moves some of the data from its memory to those servers, a large portion, if not all of the data accessed by C1 will be retained in global memory, and disk I/Os will be avoided.

A fundamental problem with exploiting global idle memory is the difficulty in identifying servers with idle memory. One of the goals of this thesis is to design memory management policies that identify and utilize idle memory efficiently. We study this in greater detail in Chapters 3 and 5.

1.2.3 Managing Duplicates in Memory

When client-server is used to manage memory, and two clients request read access to the same data page by connecting to different servers, the page gets duplicated in the buffer pools of both servers. Duplicate pages reduce the percentage of data in global memory, and this has the potential to degrade performance as it increases the number of disk I/Os. The following is an example illustrating the effects of duplicate pages in global memory.

The Figure 5 shows a web server that serves several clients that share access to files distributed among the servers. Round-robin is used to distribute the access requests (access requests to files),from the client, among the servers, for CPU load balancing. The clients connect, requests one or more files, receives the files, and then disconnect.

With client-server memory management, after a period of time, the memory image at

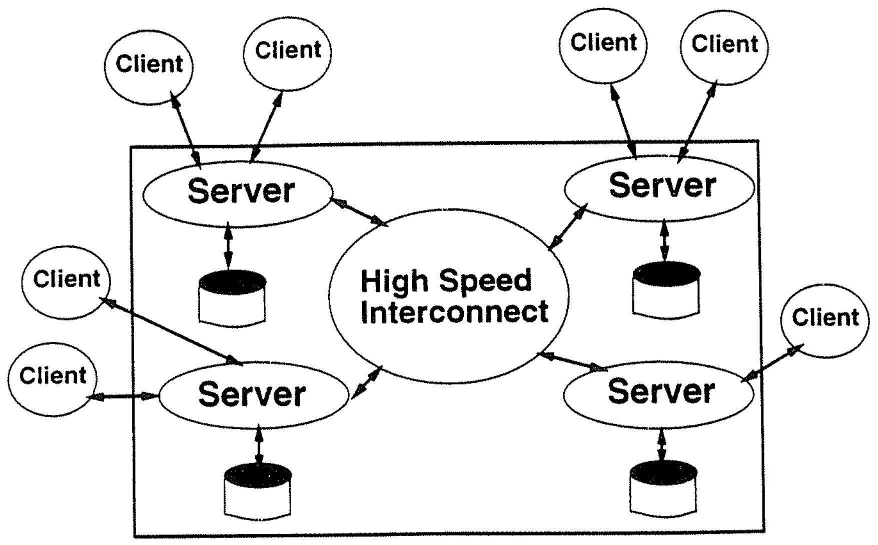

Figure 5: Managing Duplicates

each server begins to look alike as files get duplicated in the buffer pools of the servers. Due to duplication, instead of caching data of size N*M (where N is the number of servers and M the buffer pool at each server) in global memory, only a fraction, close to size M gets cached in global memory. In this situation, the servers will frequently go to the disks for data that could have otherwise been obtained quickly from another server's memory had there been no duplication. At the other extreme, a buffer management policy that prefers eliminating duplicate pages at each server will maximize the data in global memory, but would suffer from increased network traffic to fetch frequently accessed data from remote memory.

In general, neither client-server, nor the elimination of duplicates, provides the optimal response time. On the one hand, client-server duplicates pages, as a result uses global memory poorly resulting in disk I/Os. On the other hand, eliminating duplicates results in servers having to frequently go to another server for a hot page that could have been cached locally if duplicates are allowed, giving rise to increased intra-network traffic. The key challenge is to design a buffer management algorithm that controls duplication of data in global memory, maintain a good balance of the network traffic, and disk I/O, to minimize response time. In Chapter 5, we design and evaluate a page replacement algorithm that explores the tradeoff between disk I/O and message cost to control duplicate pages in memory to minimize response time.

1.2.4 Guiding principles

The following rules are used as guiding principles in designing the memory management algorithms. These principles help us in designing and implementing memory management algorithms that are practical and easy to implement.

- **Global State Information:** Since exact global state information is expensive to maintain, memory management policies have to be designed so that they use minimal, approximate, global state information. Approximate global state information has to be maintained efficiently, so that it closely reflects the actual global state .

- **Complexity:** An important principle that we follow strictly is to design memory management algorithms that do not need a complete overhaul of the buffer management code when moving from a single server to a multi-server database system.

- **Independence:** Our aim is to design memory management algorithms so that each server has complete control of its entire buffer pool. A server does not relinquish control of any portion of its buffer pool. We do not allow this as interference among the servers will complicate the code and will make a server susceptible to faults and crashes of other servers.

- **Data placement:** The presence of data on disk influences the data in memory at that server. We study the influence of data on memory management in further detail in Chapters 3 and 5.

1.3 Related Work

The related work described in this section addresses related research in Operating Systems, Computer Architecture, and Database systems.

1.3.1 Operating Systems

There is a vast amount of Operating System literature on memory management in distributed memory systems. Li and Hudak [LH89] present the implementation of shared virtual memory on a distributed memory system. They focus on providing efficient access to distributed shared memory and study several algorithms to maintain coherence in a distributed shared memory system. Carter et al. [CBZ91] present algorithms for maintaining coherence based on the applications access patterns. These papers have focused on efficiently maintaining memory consistency, for the most part ignoring the issues of paging between nodes or to disk.

Comer and Griffoen [CG90] describe a remote memory model in a cluster that has several workstations, disk servers, and remote memory servers. The remote memory servers are dedicated machines whose memory is used by nodes with heavy paging activity. Felten and Zahorjan [FZ91] use remote memory as an extension of the processor's memory, and also as a faster swap space. When a node becomes idle, it registers itself as a memory server. A processor that is in need of memory consults the registry and uses the memory at an idle node as a backing store. Schilit and Duchamp [SD91] use remote paging to enhance the performance of mobile computers. Their goal is to migrate data from portables that are in need of memory to fixed stationary servers. The data in the servers migrate from one server to another as the portables migrate. Iftode et al. [ILP93] have investigated the memory server model by introducing a remote memory server layer as a fast backing storage between local physical memory and disks. They discuss several design issues to support sequential and message passing programs in such an architecture.

Issues related to global memory management have been addressed in distributed systems such as Emerald [JLHB88], where methods for allowing objects to migrate among sites are addressed. Migration in this case is to improve performance to bring the objects closer to sites where they are being accessed, and to simplify distributed programming applications, rather than to avoid disk I/O.

Most of the studies in the Operating Systems literature focus on utilizing memory at designated memory servers for paging. They do not dynamically identify servers with idle memory and move pages from heavily loaded servers to lightly loaded servers. None of these studies address the issue of duplicate pages in memory, nor do they consider data placement to be an issue. Transaction related issues, such as locking and recovery are not addressed.

1.3.2 Computer Architecture

Kendall Square Research (KSR) [FBR93] used an "all-cache" shared memory architecture to manage the buffers of a parallel shared nothing system. KSR has specialized hardware to support data caching at a block level, that has a size of 128 bytes. It provides a shared memory abstraction on top of a highly scalable parallel architecture. Reinhardt et al. [RLD94] propose mechanisms that exposes low-level communication and memory-system mechanisms so that programmers and compilers customize policies for a given application on highly parallel architecture. The DASH project at Stanford [LLG$^+$92] implements similar cache-coherent shared memory policies entirely in hardware. The SHRIMP project at Princeton [BLA$^+$94] has studied ways to efficiently map the network interface to virtual memory to achieve low latency, and high bandwidth communication. in a multi-computer, to support shared memory abstraction.

The distributed shared memory literature for NUMA [LE90. LLG$^+$92, BLA$^+$94, RLD94] (non uniform memory accesses) and COMA [FBR93] (cache only memory accesses) machines, described above, study mechanisms to efficiently implement a shared memory abstraction so that applications on a parallel hardware make use of aggregate memory. Efficiently managing and exploiting the aggregate memory is left to the application. It is not clear how database applications use such a distributed shared memory systems for buffer management.

1.3.3 Utilizing Idle Memory

Franklin et al. [FCL92, FCL93] study ways to augment the memory of the server in a client-server database system by utilizing the memory and disk resources on client workstations. The key idea behind their work is that, the server uses the data cached in one client's memory to serve the request of another client. This is achieved by forwarding data in memory from one client to another. The server reads the page from disk only if it is absent in its memory and the memory at its clients.

Their study do not make an attempt to identify and utilize idle memory at client workstations to cache some of the server's data. An important difference between their study and ours is that, their study deals with memory management in an asymmetric client-server setting, as opposed to a symmetric multi-server architecture in our case. The client's are unreliable, while the servers in a multi-server architecture are reliable and trustworthy. Another important difference between their work and ours is that, their work utilizes memory at other client workstations by pulling data that is already present at those client's. We refer to this as the *pull* model. They make no effort to utilize the memory on client workstations that are idle. In this thesis, we study ways to *push* data from heavily loaded server to lightly loaded ones, and then *pull* the data back when needed. We refer to this model as the *push-pull* model.

Rahm [Rah92] presents the use of extended memory to improve the performance of transaction processing systems. Examples of extended memory includes non-volatile memory, and disk caches. This work differs from ours in that, we study ways to augment a servers memory with idle memory at other servers, while [Rah92] deals with ways of improving performance of one server with additional storage devices by reducing the co-relation between the server's memory and the storage device.

Many recent papers have studied the issue of utilizing idle memory in a cluster of workstations. In the Network of Workstations (NOW) project at U.C Berkeley, Dahlin et al. [DWAP94], evaluate algorithms that make use of remote memory in a cluster of workstations connected by a fast interconnect. Their approach to identify and utilize memory is to store a page that is the last copy in memory at a randomly chosen server. They show that the N-chance forwarding algorithm that forwards the last copy of the page from one server to a randomly chosen server, N times (2 times in the paper), before discarding it from global memory, efficiently utilizes aggregate memory. This study is done using file access traces for a collection of workstations connected by an ATM network.

Feeley et al. [FMP+95] describe an implementation of global LRU replacement policy in a cluster of workstations by maintaining approximate global state information at each workstation. This work was done contemporarily with our work. The global state is maintained in a state table at each server by gathering state information from all the servers periodically. This information is used by the servers to send discarded pages to be stored at the server with the global LRU page. They show that this algorithm performs well for a variety of workloads including the OO7 benchmark. In contrast, we make use of data placement to simplify the implementation of global LRU. The memory management algorithms that we describe only requires simple modification to the client-server buffer management code.

Data placement is very important for parallel shared nothing relational database systems (RDBMS) such as Gamma [DGS$^+$90] and Bubba [BAC$^+$90]. Data placement has been shown to be crucial in exploiting the CPU and memory resources of parallel relational database system by [Gha90, Meh94, CABK88]. They demonstrate the need for declustering relations to achieve high performance. These studies focus on the effect of data placement on parallel relational query processing. The data placement tradeoffs of RDBMS are not applicable to OODBMS, because, unlike RDBMS that typically execute queries specified in a set-oriented declarative language like System Query Language [MS93] (SQL) that can be parallelized, OODBMS typically execute arbitrary C++ [Str91] code that traverse the object graph. For such applications it is not clear how data placement impacts performance.

Previous work for managing idle memory in a cluster of database servers has studied the issue of data placement and memory management in isolation. In this thesis it is demonstrated that these two issues are related; we show that data placement combined with simple memory management policies are sufficient to effectively utilize the aggregate system memory.

1.3.4 Managing Duplicates

One of the early papers that alludes to the problem of duplication in a distributed shared memory environment is the work by Li and Hudak [Li86]. Their work primarily deals with maintaining coherence in a distributed shared virtual memory environment. Inefficient use of global memory due to duplication is identified as a problem, but is not studied in any detail, and no solutions are proposed.

A recent study on managing duplicates is the work by Leff et al. [LWY93]. They use an analytical model to study algorithms to manage duplicates for distributed memory systems. The algorithms work on the assumption that the servers have prior knowledge of the client's access patterns. The servers decide on the objects to replicate based on a static cost analysis. The authors propose three algorithms: optimal, greedy, and distributed. Optimal is used as a baseline for comparison; with the greedy algorithm each server manages its own memory independent of other servers. The distributed algorithm needs the knowledge of future access patterns to be implementable. Leff et al. [LWY92] present another study of duplication in a local area network, and Pu et al. [PLKC90] study the effect of duplicating objects on performance along similar lines.

1.3.5 Coherence, Locking, and Recovery

There are several papers that study coherence policies for client-server database systems. The papers by Carey et al. [CFLS91], Franklin et al. [FC92], and Wang and Rowe [WR91], and Wilkinson and Neimat [WN90] study the advantages and the potential pitfalls of offloading work from the servers to the clients by caching pages and locks across transaction boundaries at the client. Offloading work to the client has two important side effects: 1) it reduces the load on the server as there is reduced lock and page request from the client to the server, and 2) it reduces the path length for the transaction as the client's use the cached locks on pages. In order to maintain consistency of the cached pages, the server has to call back the locks at the clients that have cached them, before granting conflicting lock requests to other clients.

Rahm [Rah93] compares and evaluates various buffer coherency and locking policies for shared nothing database systems. Markos et al. [CFZ94] examine the tradeoffs between object and page locking schemes for client-server database systems. They describe a new locking algorithm that enables servers to lock pages, or objects; escalate locks to pages, and de-escalate back to objects, based on the read/write contention to the objects on a page.

In a data sharing environment, Dan and Yu [DY91] use an analytical model is used to study buffer management in a two level buffer hierarchy. Recent papers by Dan et al. [DY92, DY93] studies callback-style shared disk caching algorithms and investigates the performance gains that are available by avoiding disk writes when transferring dirty

Chapter 2

Multi-Server Database System

2.1 Architecture

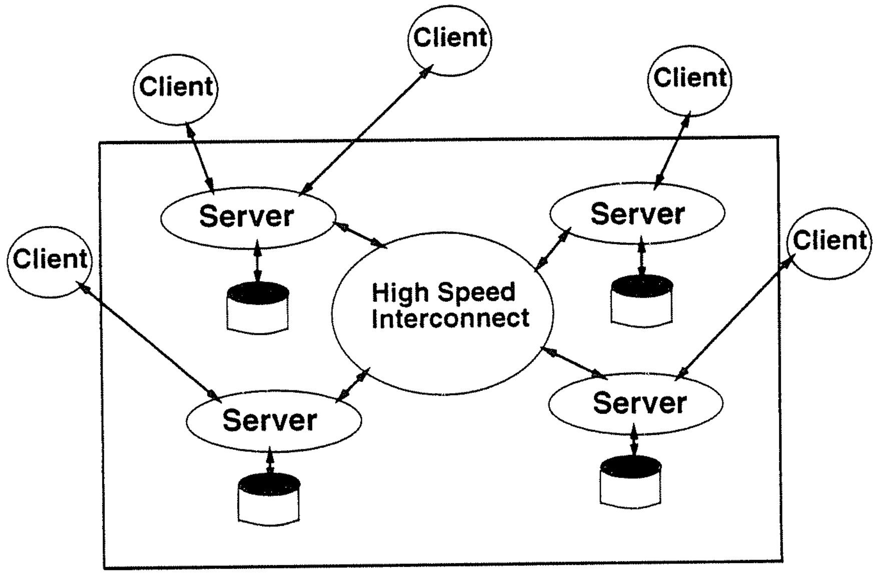

Figure 6: Multi-Server Database System

The multi-server architecture we consider is loosely based on the SHORE [CDF+94] database system architecture. A primary goal of SHORE is to provide a robust, high-performance, persistent object database system that is flexible enough to be employed in a wide range of applications and computing environments. To meet this requirement, the architecture is based on a novel *peer-to-peer* process structure shown in Figure 6.

The multi-server database system architecture consists of a group of communicating servers. The servers perform two basic functions: 1) they manage persistent objects they store, and 2) they provide applications with access to persistent objects that are managed either locally, or by other remote servers. A key concept behind this architecture is that an application is given access to the entire distributed persistent object space through direct interaction with a single server known as its *primary server.* In Figure 7, server S1 is a primary server of client C1. Servers that manage persistent data items are the *owners* of those data items. In Figure 7, server S2 is the owner for the data resident on its local disk. Owners are responsible for maintaining transaction consistency for the data items that they own.

There are several advantages of presenting a single server interface to the client when moving from a single server to a multi-server database architecture. The main advantage is that it hides the client from the server architecture. The servers may undergo reorganization without the client being aware of it. The servers can be made to coordinate to better utilize the memory resources. The complexity of managing distributed resources, and distributed data, is handled by the servers. For example, two phase commit, adding servers, distributed logging, etc, can be masked from the client.

2.2 Page-Server Architecture

Client-server database systems are classified into query or data shipping architectures, based on the unit of interaction between the servers and the client. In a query shipping architecture, the client ships the query to the server, the server processes the query, and responds with the results of the query. In a page shipping architecture, clients request data from the servers and the server responds with the data objects. At commit time, updated data is moved from the client to the server. All the data processing takes place at the client. Further, the unit of data transfer between the client and the server in a data shipping architecture could be either an object or a page.

Most commercial relational database systems have adopted the query shipping architecture [KCWW92]. The advantages and disadvantages of the query shipping architecture are summarized in [HF86, fADF90].

Commercial client-server object database systems, Observer [HZ87], O2 [Deu91], ObjectStore [LLOW91], and client-server EXODUS [CDG^{+}90], employ the data shipping architecture. They use a page as the unit of transfer between the client and the server. This is called the page-server architecture. Performance tradeoffs between page and object shipping architectures are examined by DeWitt et al. [DFMV90]. The page server architecture is employed by most object database servers. The pages requested by a client are transferred to the client workstations where they are processed. Processing data at the client makes use of the computing resources at the client workstations.

In a multi-server architecture there are two options to choose the site where the data is to be processed, the data is either processed at the client, or at the server. If the processing is done at the client, then the data transfer between the primary server and the client may take place over a network that is much slower than the multi-server interconnect. The performance benefits of having a high speed network connecting the servers may be negated by a slow network connecting the client to the servers. An alternative that is attractive, is to process the data entirely at the primary server. This alternative has the advantage that clients will be able to make use of memory at other servers over a high speed network, but has the disadvantage that the resources at the client are not fully utilized. Maximizing memory usage among the servers is useful only if the clients are connected to the server over a high speed network, or if the processing takes place at the server. For the purpose of this thesis, we assume that the data is processed at the server. While this is not standard in OODBMS, some commercial systems like Versant [Tec91] and Gemstone [Be89] support this.

2.3 Access path

This section describes the path taken by a page request that originates at the primary server until the page becomes resident in memory at the primary server. When a primary server requests a page, the following sequence of events takes place to bring a page into the buffer pool at the primary server.

If the requested page resides in the primary server's memory, the request is satisfied locally. If the page is absent, the primary server sends a request to the owner of the page. Note that the primary server itself may be the owner. Each server maintains a directory structure in which it keeps track of the copies of the pages that it owns in global

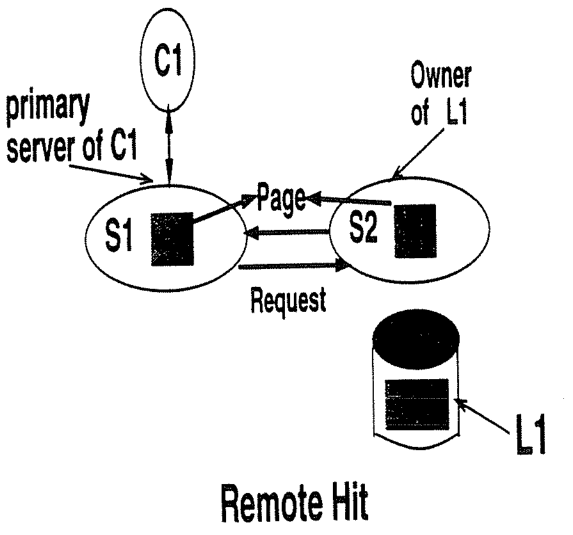

Figure 7: Remote Hit

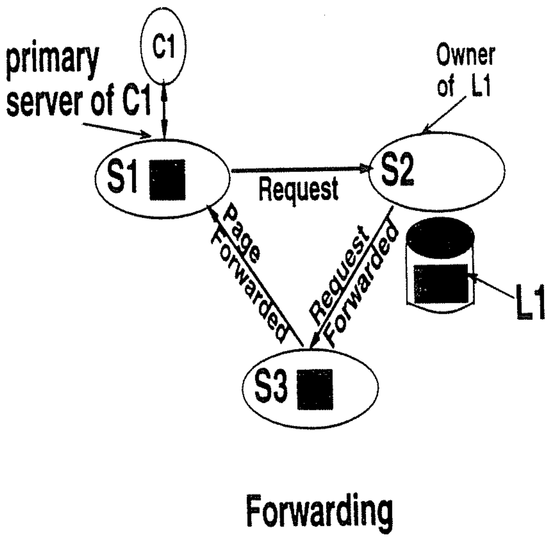

Figure 8: Forwarding

memory. The directory structure is used by the owner to maintain coherence, locking, and recovery information (dirty page list, log sequence numbers) about these pages.

Upon receiving a request from the primary server, the owner can do one of two things: 1) always read the page from disk and send the page to the requesting server, or 2) look up the directory entry, and request the page to be sent to the requesting server, if it is present in memory at some other server. The technique in option 2 is known as *forwarding*. *Forwarding* has been shown to make good use of data already present in global memory in studies by Franklin et al. [FCL92], and Dahlin et al. [DWAP94]. We therefore use option 2, *forwarding*, instead of always reading the page from disk. With *forwarding*, the owner reads the page from disk only if the page is absent in global memory.

When the owner reads the page from disk and sends it to the primary server, it retains a copy of the page in local memory. Franklin et al. [FCL92] show that getting rid of this page quickly from the owner's memory utilizes global memory better, since this page is not in use locally and there is a copy elsewhere in memory. This page is marked as hated (referred to as a *hate hint*) in the owner's memory and is placed on a Last-In First-Out (LIFO) list. When a page is needed for replacement, the hated pages are replaced first.

Meanwhile, the primary server makes room for the requested page by locating a page for replacement. The replacement algorithm is crucial in determining how effectively memory is utilized. For example, to utilize global idle memory, the replacement algorithm must choose between local pages, pages owned by servers that have idle memory, and pages owned by heavily loaded servers. To manage duplicates in global memory, the replacement algorithm has to choose between pages that are duplicates that can be fetched easily from remote memory, and single pages that must be read from disk when re-accessed. Page replacement algorithms are important in determining how global memory is utilized. We design and study page replacement algorithms that utilizes idle memory and controls duplicates in the following chapters.

After a page is chosen for replacement, the primary server has two choices: 1) it can discard the page and notify the owner about it by piggy backing this information with other messages, or 2) it can always send the page back to the owner so that the owner can retain it in its memory. We pick option 2, because, by shipping a page back to the owner, idle memory at the owner can be replaced with a useful page.

After making room for the page, the primary server receives the requested page and places the page in its buffer pool.

2.4 Coherence, Locking, and Recovery

The locking and coherency strategy that we adopt is a slight variation of call back locking, described in Wang and Rowe [WR91]. Call back locking is a pessimistic lock-based protocol, that has been shown to perform well for a wide range of workloads for client-server database systems in [FCL92]. Call back locking uses invalidations to ensure "read multiple, write one" memory coherence.

The salient feature of the call back locking strategy adapted for a multi-server architecture is that, each server is responsible for granting lock requests to the page that it owns. A primary server writes or reads a page only after obtaining the necessary locks from the owner. Once a lock is obtained, the lock for the page is cached at the primary server, until either the page is discarded from the primary server's memory, or the lock is invalidated by the owner. When a primary server first requests the page (read/write), the lock request for the page is granted along with the page request. The lock is retained at the primary server across transaction boundaries and granted to other transactions at the primary server without having to go to the owner.

When the owner grants a read lock to the primary server, the owner makes sure that there is no other server holding a write lock on that page. If some server has a write lock on the page, then the owner requests that server to downgrade the write lock to a read lock before granting the read lock to the primary server. If a write lock is requested by the primary server, the owner invalidates other copies of the page in global memory and by calling back the read locks and write locks on the page. It then grants a write lock

Location	Access Time μs
Primary server	150
Owner's memory	2,800
Remote memory	5,500
Local Disk I/O	9,900
Remote Disk I/O	11,500

Table 3: Memory Hierarchy

on the page to the requesting server.

Client-server ARIES [MN94], or client-server EXODUS recovery [FZT+92], or the recovery schemes described in White and DeWitt [WD95], for client-server database systems, can be applied, without changes, to the multi-server database system. The multi-server system can be treated exactly as a client-server database for recovery purposes, because, each server maintains its own logs, and checkpoint pages in its buffer pool, and treats other servers as clients for the data they own.

2.5 Memory Hierarchy

The memory hierarchy consists of the primary server's memory, the owner's buffer pool, followed by the memory of other servers, and finally the data on disk. For the purposes of this study, we do not distinguish between local disk I/O and remote disk I/O, as the disk I/O cost is significantly higher than the message cost. Table 3 illustrates the memory hierarchy along with the typical access times to each memory layer on the IBM SP/2 using TCP/IP. Notice that, the cost of a disk I/O is three times the cost of accessing the data from the owner's memory.

This chapter has described the architectural frame work for designing and implementing the memory management algorithms. Chapter 3 uses a simulation to study how global idle memory is utilized in the multi-server. This is followed by an implementation study in Chapter 4. Managing duplicates in global memory is presented in Chapter 5.

Chapter 3

Idle Memory:Simulation Study

3.1 Data Placement and Memory Management

When we began this research, our first step to solve the problem of utilizing idle memory was to devise efficient ways to identify servers with idle memory and transfer hot pages from heavily loaded servers to replace cold pages on lightly loaded servers. To locate servers with idle memory, each server must maintain global state information to keep track of servers with idle memory and ship discarded pages to those servers. The challenge is to identify servers with idle memory by maintaining minimal global state information.

One solution to utilizing memory at all the servers is for each client to make use of memory at all the servers evenly. A simple way of doing this will be for each client to direct its accesses evenly to all the servers so that the pages in memory at each server is accessed with equal probability. Since databases have persistent data, one way of achieving this is to distribute the data uniformly among the disks of all the servers. The rational for doing this is that, when the data is accessed by the client, it is brought into memory at the owner, before it is shipped to the primary server. This can be used to exploit the memory at the owner, and this is illustrated by the following example.

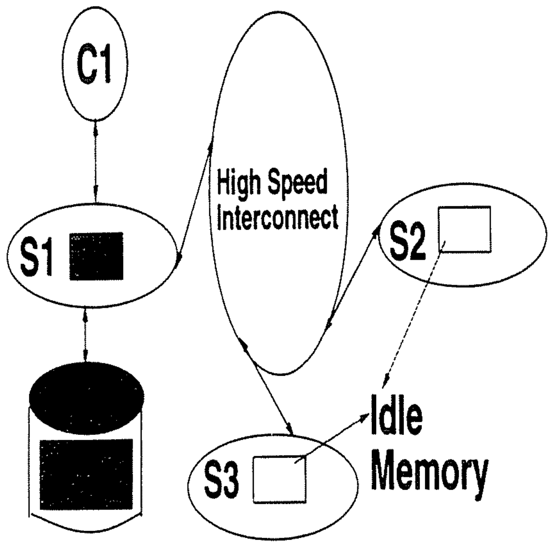

Figure 9: Memory at S2 and S3 Idle

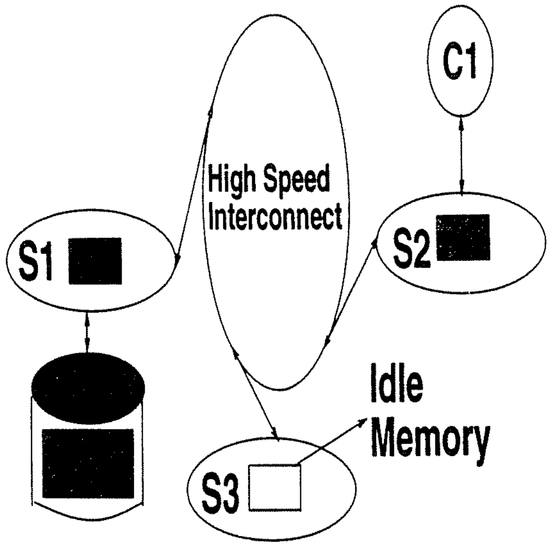

Figure 10: Memory at S3 Idle

Figure 9 shows a client C1 connected to server S1 of a multi-server database system. Server S1 has all the data that the client accesses on its local disk. Client-server memory management (LRU at each server) is used to manage the memory of the multi-server. If the data accessed by client C1 is resident at server S1, and is larger than server S1's memory, disk I/O's become necessary even though there is idle memory at servers S2 and S3. Instead, if the client C1 connects to server S2 and accesses data at server S1 remotely, then the memory at server S2 will be put to use, if the page replacement algorithm avoids duplication between the owner and the primary server. Figure 10 illustrates this. This shows that, in order to exploit memory of two servers, it is better to access data from a remote server than to access the data locally.

An extension of this idea is to evenly distribute the accesses by client C1 among all the servers. If the data is distributed, the data accessed by the client C1 are uniformly directed to the data at all the three servers. The memory at all the three servers will be

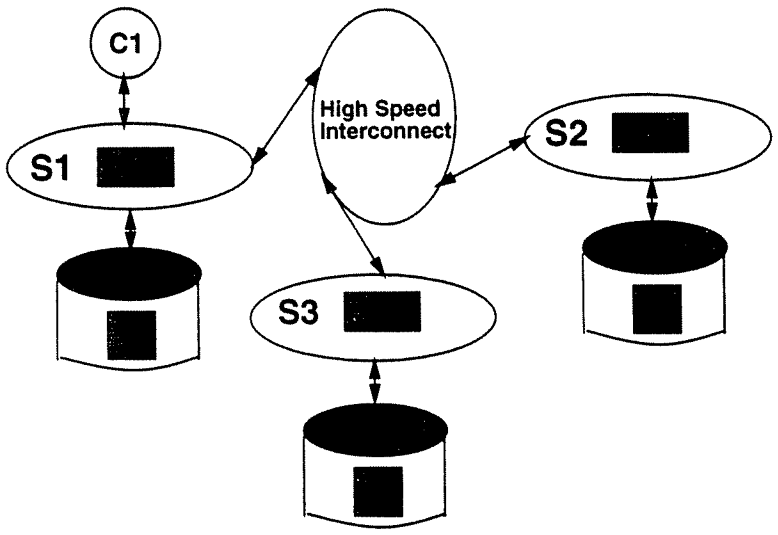

Figure 11: No Idle Memory

utilized if the page replacement algorithm avoids duplicating pages between S1, and the memory of servers S2 and S3. Figure 11 shows such a configuration.

The above example illustrates that the uniform distribution of data, combined with simple modification to the page replacement algorithms, to avoid duplication between the primary server and the owner, fully utilizes global memory.

By distributing the data among the servers, accesses from the client are uniformly directed to all the servers. Although, this helps utilize aggregate memory efficiently, it is insufficient to efficiently utilize global memory. Client-server page replacement algorithm must be modified so that it uses the declustered data set to make use of global memory. The following are some of the problems that must be tackled by the page replacement algorithms.

1. The page replacement algorithm must prevent duplication of data pages in memory between the primary server and the owner.

2. When there are multiple clients, some servers are heavily loaded and the others are lightly loaded. Pages owned by lightly loaded servers, when discarded, are more likely to be retained in memory than those of heavily loaded servers. If all the pages are discarded with equal probability, and this is the case with client-server page replacement, disk I/Os become necessary when pages from heavily loaded servers are reaccessed, even though there is idle memory available at other servers.

3. Since each node is a server for the data that it owns and a client for the data that it retrieves from other servers; the servers have to strike a balance between the local disk data and the client data that it caches.

In order to use global memory, the page replacement algorithms must be designed so that they solve these problems. In the following sections we describe data placement and page replacement algorithms and analyze how global idle memory is utilized for various combinations of data placement and memory management policies.

3.2 Data Placement

OODBMS provide mechanisms to group related objects, for instance Objectivity provides containers, Ontos has segments and files, and SHORE provides pools. For uniformity we call these object collections as units of locality or locality sets. For example, in a CAD database, the objects in the private workspace of a user represents a locality set. A typical multi-user database system has several locality sets. The decision on how to place locality sets at the servers is static and must be carefully made as data re-organization on disk is very expensive. We consider two data placement alternatives, namely, *clustering* and *declustering*.

3.2.1 Clustering

Each locality set is clustered in its entirity at one server; different locality sets maybe placed at different servers. If a server has multiple disks, the pages of the locality set are striped across all the disks on that server. Figure 12 illustrates five locality sets distributed among three servers, and each locality set is placed in its entirety at one server.

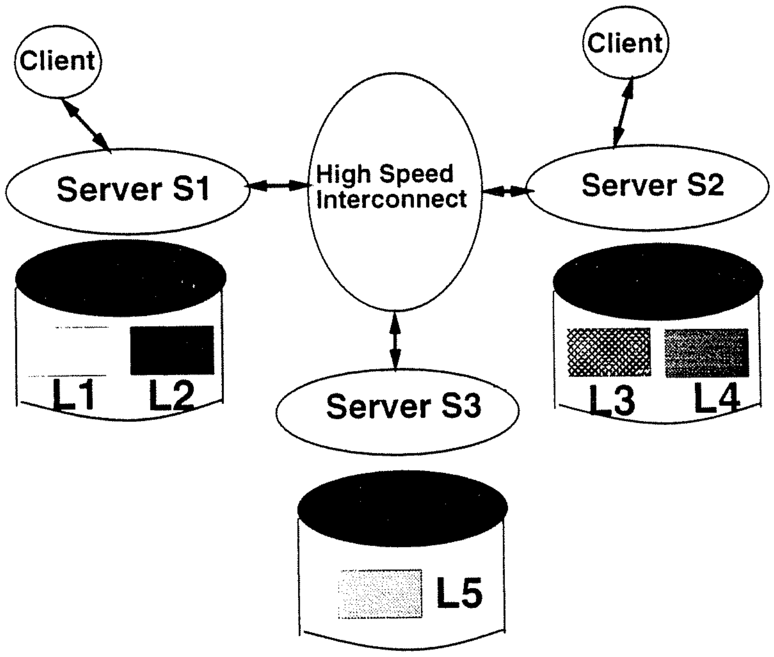

Figure 12: Clustered

The placement of the client application with respect to the locality set from which it accesses data is important in determining how memory is utilized. The earlier example in this chapter, illustrated by Figures 10 and 9, demonstrates that a client accessing data from a remote server makes use of memory at two servers, the primary server and the owner, while the client that accesses data from the primary server's disk only makes use of the memory at the primary server.

In order to evaluate the impact of client placement on how memory is utilized, we evaluate two client placement strategies, namely, *Clus-Diff* and *Clus-Same*.

1. **Clus-Same:** Clients connect to servers that store their data and access all the data locally as shown in Figure 9.
2. **Clus-Diff:** Clients access data by connecting to a server other than the server where its data is resident as shown in Figure 10.

3.2.2 Declustering

When data is declustered, the pages of the locality set are uniformly distributed across all the servers. For example, round-robin could be used to distribute the pages of a locality set. If a server has more than one disk, the pages may be striped across all the disks at that server. The objects on each page are assumed to be optimally clustered on a page to maximize the hits on each page[1]. Figure 13 shows five locality sets declustered among three servers.

One of the requirements of declustering, to make use of idle memory at all servers,

[1]clustering of objects on pages is not necessary for the algorithms

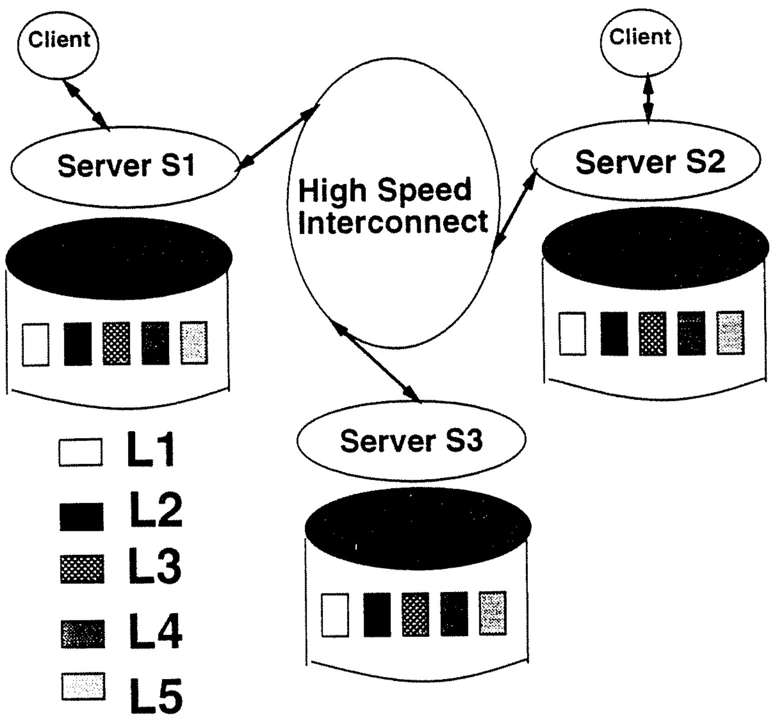

Figure 13: Declustered

the client must direct direct its accesses uniformly to all the servers. Therefore, the declustering strategy must distribute the hot and cold pages of the locality sets evenly among the servers. There are several ways one could decluster the pages of a locality set. One could use round-robin, hashing, or range partitioning to decluster the locality set into groups of equal number of pages, to distribute the data among all the servers.

In this thesis, we use round-robin to decluster the pages of the locality set. We assume that round-robin evenly distributes the hot and cold pages of the locality set among the servers. This is not unreasonable, because, barring pathological cases the hot and cold pages are likely to to distributed uniformly among the servers with round-robin. If round-robin does not distribute the hot and cold pages of the locality set uniformly among all the server, a scheme that randomizes the distribution of pages better, like hashing, could be used to decluster the data. The only criteria required of the declustering strategy is that it should distribute the hot and cold pages of the the locality set evenly among the servers so that accesses from the client are directed uniformly to all the servers.

3.3 Page Replacement Algorithms

In this section we describe the page replacement algorithms.

- **LRU:Base**

  In a client-server object database system, the clients and servers manage their buffer pools using an LRU replacement policy. When a server reads the page from its disk it retains a copy of the page in its local buffer pool. If the page is requested in the read mode, the owner retains a copy of the page in its memory; however, if the page is requested in the write mode, the owner sends the page to the requesting server without retaining a copy in its memory (since we use the "read multiple, write one" technique to ensure consistency of the pages). When a server replaces a page, it is shipped back to the owner. The owner retains the page in its memory if it has hated pages, or if the LRU time stamp[2] on the page shipped to it is lower than the LRU time on the page in its buffer pool.

- **LRU with Hate Hints:ClSv**

[2]we assume that a global time stamp is available, if it is not available, we describe an algorithm to maintain that in Chapter 4

This algorithm is similar to the Base algorithm, except that the owner marks the page that it reads from disk and retains locally as "hated". This is a modification proposed to the Base algorithm by Franklin et al. [FCL92] and we refer to this algorithm as the ClSv algorithm. This is, in effect, a hint to the buffer replacement policy that this page is a good candidate for replacement, because, the page is not in use at the owner and there is a duplicate copy elsewhere in memory. In the absence of invalidated page frames, the page frames marked hated are replaced in LIFO order, before the remaining pages are replaced in LRU order.

- **Hate with Reservation:Reserve**

Figure 14 illustrates a snap shot of the memory image at server S1 and S2 at run time, while employing ClSv on a declustered locality set L1 accessed by C1. The size of the locality set accessed by the client, in this example, is smaller than the combined memory of S1 and S2. The client directs its accesses uniformly to the pages of the locality sets at both servers. Therefore, the memory at server S1 contains two classes of pages: a) pages owned by the local server S1 accessed by C1, and b) pages owned by S2 accessed by C1. When S1 discards a local page, the page must be read from disk the next time it is accessed. However, if S1 discards a page owned by S2, the page will be retained in memory at S2, since it has a lower elapsed time (recall that a page that is discarded is shipped back to the owner, and the owner retains the page). If ClSv is the replacement policy, the pages owned by S1 and S2 are discarded with equal probability from S1's memory. When a local page that is discarded is reaccessed, it causes disk I/O at server S1. This will be avoided if the idle memory at server S2 were utilized. In order to utilize idle memory at server S2, S1 must retain local pages and preferentially discard pages owned by S2. By doing this, local disk I/Os will be reduced, and idle memory at server S2 will be utilized.

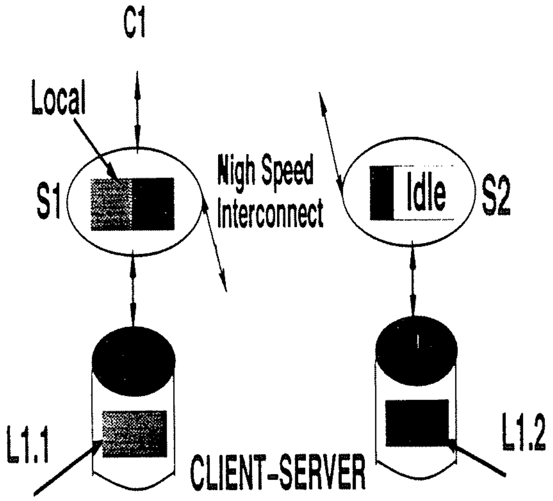

Figure 14: ClSv

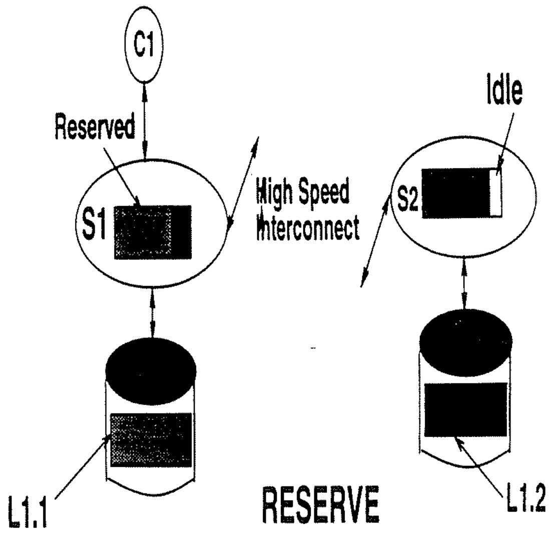

Figure 15: Reserve

A solution to reduce local disk I/O's and make use of idle memory at server S2, in this example, would be to reserve a portion of S1's memory for local data. Figure 15 shows the memory images of server S1 and S2 when a large portion of S1's memory is reserved for local data. Reserving memory at server S1 reduces the idle memory at server S2 at it moves some of the data owned by server S2, from server S1 to server S2. Reserving pages reduces the buffer space at server S1 for caching pages owned by S2. This increases the message traffic from S1 to S2 to fetch pages from memory at server S2. The memory that has to be reserved for the best performance is a function of the disk I/O latency, the message latency, and the workload. In order to focus our study on the issue of data placement on memory management we conduct experiments by fixing the amount of pages reserved at 60%.

This policy is intended to demonstrate the performance impact of reserving memory for local data. In Chapter 4 we study a page replacement policy, Global, that dynamically reserves memory for local and remote pages.

- **Optimal Algorithm:"Optimal"**

  The "Optimal" algorithm views the aggregate system memory as centralized memory shared by all the servers. An LRU replacement policy is used to replace pages in memory[3]. All servers have access to the global memory as if it were local to each server and incur no message costs to access it. The "Optimal" algorithm, although not practical to implement, is used as a baseline for comparing the performance of other algorithms.

3.3.1 Policies

Table 4 summarizes the six policies that are various combinations of memory management and data placement strategies. The performance evaluation of these policies is presented in the following section. We evaluated the "Optimal" algorithm when the locality set was declustered and when it was clustered. The memory utilization was unaffected by the data placement strategy, but the disk I/O performance was worse when the data was

[3]since we consider LRU to be the best one could do to manage memory in the absence of prior knowledge of access patterns

Policy	**Memory Mgmt Algorithm**	**Data Placement Strategy**
Base-Decl	Base	Declustered
ClSv-Decl	ClSv	Declustered
Resv-Decl	Resv with 60% Reservation	Declustered
Base-Clus	Base	Clustered
ClSv-Clus	ClSv	Clustered
Optimal	Optimal	Clustered

Table 4: Policies

declustered. Briefly, when data is declustered, each client directs its access to any of the disks, and this results in queuing at a disk when more than one client directs request to the same disk simultaneously. This increases the average disk I/O time. However, when the data is clustered, each client directs its request to only one disk, therefore, the queuing does not arise. Therefore, we use clustering to evaluate the performance of the "Optimal" policy.

3.4 Simulation Model

In this section, we describe the simulation model, and follow this with a description of the workload.

3.4.1 System Model

The simulator is written using CSIM/C++ [Sch90] process oriented simulation package. The simulator models the disk, network, CPU, and the buffer pool. The simulator uses the CPU instructions for various segments of the buffer management code, instruction

Parameter	Value
Cpu MIPS	30
Service Discipline	Processor Sharing
Page Size	8 KiloBytes
Link Bandwidth	15 MB/sec
Message Cost	550 μs for 8 KB
Message Buffers	160 KB/node
Memory/node (M)	8 MB/node

Table 5: Hardware Parameters

overhead for messages, disk I/Os, table lookups, and instructions for locking/unlocking. The simulation experiments are run using Condor [LLM88], which provides abundant CPU cycles by exploiting CPU cycles of idle workstations.

Physical Model

The hardware platform simulated is a shared nothing, distributed memory architecture, with a database server on each node. The number of physical nodes is controlled by the parameter N. For most experiments the number of nodes is fixed at 10. Each processing node has a 30 MIP CPU, four disks, and 8 MB of memory. We use a buffer pool of only 8 MB per node because, experiments with this memory size take a long time run (around 60 minutes for each data point), and larger memory sizes further increases the simulation time. Additionally, it is the ratio of the database size to the memory size that is important in determining how global memory is utilized. We demonstrate this with an experiment in Section 4.5.7. The hardware parameters of a node are summarized in Table 5. The disk that is modeled is a Fujitsu M2266. The disk parameters are shown in the Table 6.

Parameter	**Value**
Disks/Node	4
Disk Capacity	1 GB, 5.25"
Pages/Cylinder	83
Cache Size	4 Pages/Context
Cache Context	8
Transfer Rate	3.09 MB/sec
Rotation	16.667 ms
Seek Time [BG88]	$0.618*\sqrt{SeekDistance}$
Settle Time	2.0 ms

Table 6: Disk Parameters

We emulate the interconnect of a modern multicomputer. The interconnect is modeled as a network with bidirectional FIFO links with a link bandwidth of 15 MB/sec. We do not model network contention as it is not much of an issue in a real network due to the high network bandwidth. Each node has a finite number of message buffers (about 20) to send and receive messages. The messages incur a fixed protocol overhead, a per byte instructional overhead, and a network delay. With these parameters, the message cost to send a 8 KB page from one node to another takes about 550 μs.

Table 7 summarizes the CPU costs and execution parameters that are used in the simulations. The instruction count for the execution parameters are from Franklin et al. [FCL92].

Parameter	**Value**
Lock	300 Inst
Table Lookup	100 Inst
Fault	4,000 Inst
Per Page	30,000 Inst
Msg Protocol	11,000 Inst
Lock Timeout	500 ms

Table 7: Execution Parameters

3.4.2 Workload

The workloads simulated in the experiments reflects those experienced by object database systems. Typical CAD applications consists of, clients processing data from large object graphs. Some examples of such object database applications include the CAD design rule checker, the VLSI router, aircraft simulation for computing air pressure on the aircraft, and the OO7 benchmark.

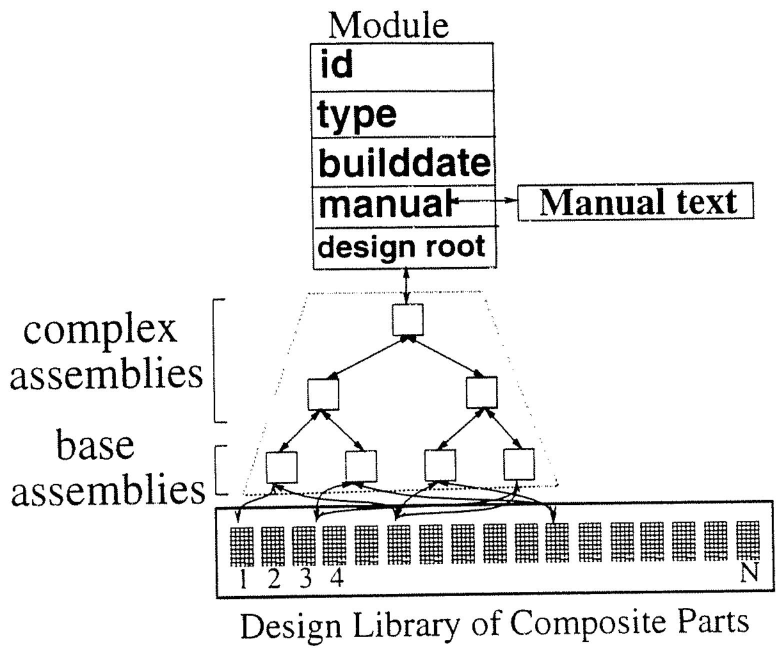

Figure 16: OO7 Module

We model our workloads loosely on the OO7 benchmark. The OO7 [CDN93] benchmark is an object database benchmark developed at the University of Wisconsin that models workloads experienced by CAD database. The schema of the OO7 benchmark consists of a module object that points to an assembly hierarchy. The base of the assembly hierarchy points to composite part objects. Each leaf node of the assembly is associated with three composite parts that are randomly chosen from the set of composite parts. Each composite part object consists of a private atomic parts graph, consisting of 20 atomic part objects, that are interconnected to each other with connection objects. The composite part also has a document object associated with it. Figure 16 illustrates the schema for OO7, and Figure 17 shows the structure of one of the composite part object. Methods, referred to as traversals, executed by the OO7 benchmark traverses the assembly hierarchy, and then traverses the composite parts attached the base of the assembly hierarchy. Some traversals perform a Depth First Search (DFS) traversal on the atomic parts graph present in each composite part, while others just touch the root atomic part object. Each traversal is long running, and it accesses a large number of composite parts, from the module. We model such an application using a synthetic workload that maps the traversal to the accesses to the pages in the module. This is explained in the following paragraph.

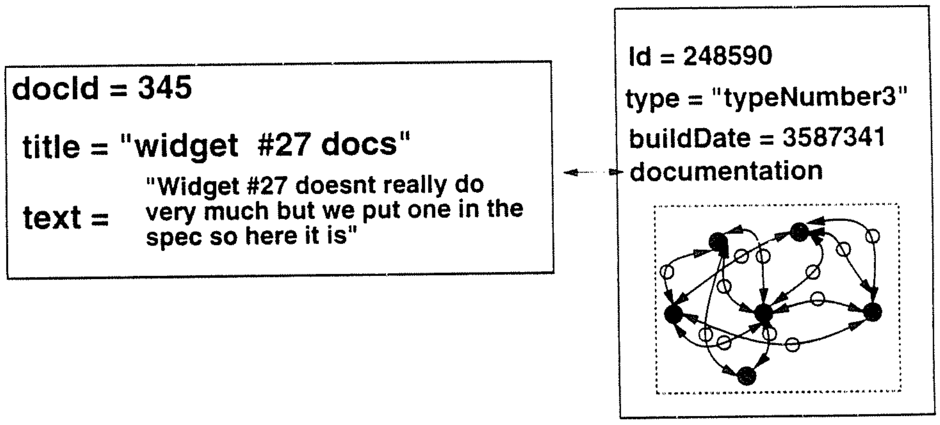
docId = 345
title = "widget #27 docs"
text = "Widget #27 doesnt really do very much but we put one in the spec so here it is"
Id = 248590
type = "typeNumber3"
buildDate = 3587341
documentation

Figure 17: Composite Object

In our workload, each client is associated with a private locality set. The private locality set can be visualized as a module in the OO7 benchmark. The private locality set consists of several pages, and each page in the locality set roughly corresponds to a composite part object. The atomic part graph and documents associated with a composite part object are assumed to clustered together on the page. We do not model the traversal in the assembly hierarchy, but model the accesses to the composite part objects and the traversal in the atomic part graph. In the OO7 schema, the composite parts are randomly assigned to the base assemblies; therefore, the traversals in the benchmark access composite parts at random from the module. We characterize the accesses to the composite parts by the benchmark traversal as random accesses to the pages in the locality set by using a uniform probability distribution. We also model the traversals in the atomic part graph and connection objects by associating a CPU overhead for processing each page that is fetched by the client. The amount of CPU overhead for processing a page is a parameter that can be controlled.

In addition to these parameters, each locality set in the workload has a hot and a cold region. The probability that an access is directed to the hot region is given by P_{hot}. Accesses to pages within the hot and cold region are uniform. To model sharing among several clients, we maintain a shared locality set that is accessible to all the clients. The probability of directing accesses to the shared locality set is governed by P_{shared} parameter. This workload characterization of object databases is similar to that used by [FCL92]. The probability of the client updating a page is specified by the write probability P_{write}. We explore the effects of writes in only one experiment, since the study of how idle memory is utilized is mostly orthogonal to the presence or absence of writes. The important effect of write is the coherence messages, log messages, and writes to disk, and the effects of these are orthogonal to how idle memory is utilized.

Name	**Default**
Number of Large Clients	1
Number of Small Clients	0
Large Locality Set (Hot) D_{large}	M to $40 * M$
Small Locality Set (Hot) D_{small}	$0.4 * M$
Cold Region for D_{large}	$5 * D_{large}$
Cold Region for D_{small}	$5 * D_{small}$
Shared Locality Set	$3 * M$
Hot Probability	1.0
Sharing Probability	0.0-1.0
Write Probability	0.0-0.2
Request Size	1 Page
Transaction Size	500
Think Time	0 ms
Cold Time	150 seconds
Sim Time	400 seconds

Table 8: Workload Parameters

We use two kinds of clients to model object database workloads. Large clients, that access data from a large private locality set, typically larger than the size of a server's memory, and small clients that access data from a small private locality set that is comparable to the size of a server's memory. The size of the locality set is measured as a multiple of the memory size of one server. The association between the client and the locality set that it accesses is dynamic. For each simulation run, the client randomly selects a locality set to operate upon. To ensure the statistical validity of our results we run our experiments 15 times with different seeds, verifying that the response times for these runs are within 2 percentage points of the mean in all cases. The Multi

Programming Level (MPL) at each node is restricted to one. Experiments with higher MPL values are studied in Chapter 5.

During the course of execution, a client uses a uniform probability distribution to choose a page from the private locality set. On receiving the page, the client uses 30,000 instructions to process the page. We break up the accesses of the client to the pages of the locality set into transactions. A client runs several transactions against the locality set. In each transaction, the client accesses 500 pages, corresponding to the large transaction size for the OO7 benchmark. At commit time the logs are sent to the owners of the pages updated by the transactions. The simulations were run for a period of time specified by Sim Time. To avoid transient startup effects, each run includes a startup time specified by Cold Time to warm up the buffer pool before collecting statistics.

The workload parameters and their default values are summarized in Table 8. The policies that we evaluate are summarized in Table 4.

3.5 Performance

In this section we evaluate the six policies that are summarized in Table 4. Briefly, the policies are combinations of memory management and data placement strategies described in Section 3.3.1.

The primary performance metric is the average request response time of the large client. The response time of the large client is the average over all the requests for the large client. The response time of the small client is not interesting, as the small clients cache their entire locality set in local memory and incur no I/O or message cost.

Parameter	**Expt 1**	**Expt 2**	**Expt 3**	**Expt 4**	**Expt 5**
Large Clients	1	1	1	**1 to 10**	5
Locality Size (Multiple of M)	4	**2 to 50**	12	3 and 12	3
Nodes	10	10	**5 to 40**	10	10
Sharing Prob	0	0	0	0	**0 to 1.0**
Message Cost	**0 to 8ms**	550μs	550μs	550μs	550μs

Table 9: Experiment Parameters

3.5.1 RoadMap

In moving from a single server architecture to a multi-server, the least that one expects is for a single application to perform better with a larger aggregate memory. In the first three experiments, we study how a single large client makes use of the aggregate memory in the multi-server. We study how memory load imbalance is handled by introducing multiple large clients. Finally, we study the effect of sharing on performance. Table 9 summarizes the parameters used in the experiments. The parameter in bold refers to the one that is varied for that experiment. The default values for the workload parameters are shown in Table 8.

In the first experiment, we study the tradeoff between I/O gains and message costs for two options: 1) when the client is connected to a server that has the data clustered locally, and 2) when the client is connected to the server other than the one where the data is clustered. In the second experiment, the ability of the policies to make use of aggregate system memory is explored by varying the size of the locality set. We introduce hot/cold accesses, and small clients, to study their effect on the client's response time. The effect of scaling the system to a large number of nodes is explored in the third experiment. In the fourth experiment, we evaluate the effect of uneven use of memory by increasing the number of large clients. Sharing has the effect of duplicating pages in global memory. Although we do not provide for any specific mechanisms to control duplication, we explore its effect on how idle memory is utilized in the fifth experiment. The effect of duplication on global memory is studied in Chapter 5.

3.5.2 Expt 1:Client's Placement

When the locality sets are clustered, the placement of the client has a significant impact on how memory is utilized. There are two options for placing the client with respect to its locality set: 1) the client accesses the locality set by connecting to the server that owns it, or 2) the client accesses the locality set from a remote server. We refer to the former policy as Clus-Same, and the latter as Clus-Diff. When data is declustered client placement has no effect, as all the servers are identical, since each server stores a portion of the locality set on its node. We use the ClSv page replacement algorithm to compare Clus-Same, Clus-Diff, and ClSv-Decl.

In this experiment one client accesses a locality set that is four times the size of one server's memory. The performance trade off between Clus-Same and Clus-Diff depends heavily on the ratio of the message cost and disk I/O. To study this, we plot the response time as a function of the message cost.

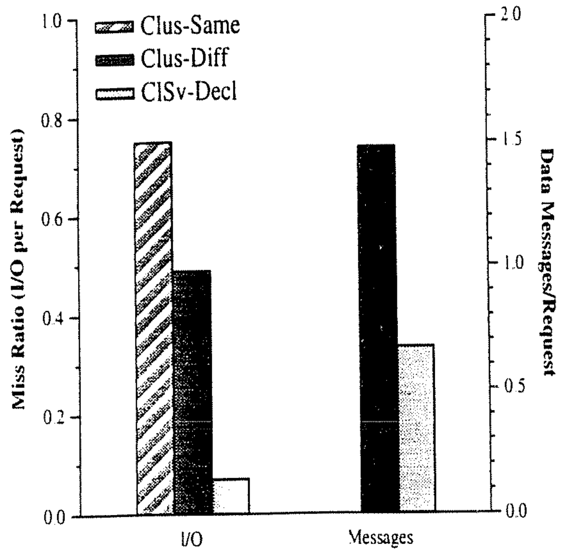

Figure 18: Placement: I/O

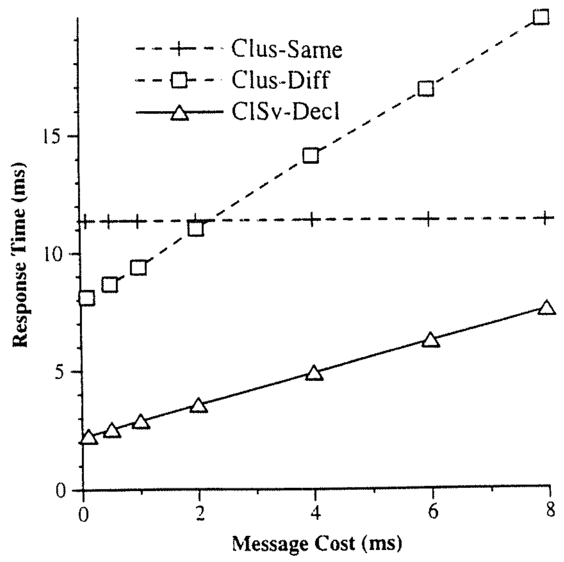

Figure 19: Placement:Response Time

Figure 18 shows the miss ratio (number of I/O per request), and the number of data messages per request. Clus-Same has the highest miss ratio followed by Clus-Diff, and ClSv-Decl has the lowest miss ratio. Clus-Same has the highest miss ratio since the client only make use of the memory at the primary server and therefore caches only 25% of the locality set in memory. Clus-Diff makes use of memory at the primary server and the owner and therefore caches 50% of the data in memory. ClSv-Decl has the lowest miss ratio as it makes use of memory at all the servers (since the locality set is declustered). Although we expect ClSv-Decl to have zero I/Os for small locality set sizes, because it makes use of the aggregate memory that is larger than the size of the locality set, we find that its miss ratio is non zero. This is because, with Clsv-Decl, the primary server discards the local pages and remote pages with equal probability. When a local page is discarded, a disk I/O is incurred to fetch the page into memory the next time it is accessed.

Turning our attention to message costs, Clus-Same incurs no message overhead as all data accesses are local. Since Clus-Diff and ClSv-Decl have the same number of local misses, we expect them to have the same message overhead. However, we notice that Clus-Diff has twice the message overhead of ClSv-Decl. The higher message overhead for Clus-Diff is due to the messages required to ship pages back to the owner. With Clus-Diff, the entire locality set is at the owner, and the size of the locality set is four times the size of memory at the owner. Therefore, almost every page discarded by the primary server is shipped back to the owner. On the other hand, very few pages that are discarded are sent back to the owner for ClSv-Decl. This is because most pages at the primary server are duplicates of pages in owners memory, as the memory at the owner is more than twice the size of the fragment of the locality set on its disk.

Figure 19 shows the response time as a function of the message costs. When the message cost is low, and the locality set is clustered, the worst thing that can happen is for the client to be hosted by the server that owns the locality-set. Clus-Diff out performs Clus-Same, and ClSv-Decl has the lowest response time. For message costs greater than 2 ms, the cost of retrieving and discarding data from the remote server outweighs the gain of using the additional server's memory to save disk I/Os. Therefore, Clus-Diff has a response time that is worse than that of Clus-Same. ClSv-Decl has a very low miss ratio, and a small message overhead; therefore, the message cost has to be very high before the response time becomes worse than Clus-Same. Clus-Diff has a larger response time slope than ClSv-Decl, as the message overhead for Clus-Diff is higher than that for

ClSv-Decl.

Todays multicomputer interconnects, network switches, and fiber optic networks have high bandwidths, and low latencies. Since message costs are inexpensive, it is better for the client to be hosted by the server other than the one where the data is declustered. This experiment demonstrates that Clus-Diff is superior to Clus-Same, therefore, we use Clus-Diff as the client placement strategy in the rest of the experiments. In the rest of the experiments, in order to make the choice of primary server simple, we use a random placement of clients and the locality set. The probability that a client ends up at the server where the locality set is clustered is very small. We run experiments with different random seeds, for different client placements, and take the average of the response times.

3.5.3 Expt 2: Data Size

In this experiment, we study the ability of one client to make use of the aggregate memory resources of the multi-server. We study this by varying the size of the locality set. One client accesses data from a private locality set, the size of the locality set is varied as a multiple of one server's memory. For example, when the size of the locality set is 10, the size of the locality set equals the size of aggregate memory, since there are ten servers. Figure 20 shows the miss ratio as a function of the size of the locality set. The graph has two distinct sets of lines. The lines corresponding to the miss ratio for the policies that decluster are very low when compared to those for the policies that cluster. When the locality set is declustered, memory at all the servers is used to cache the locality set, but when the locality set is clustered, the memory of at most two servers is used to cache the pages. Therefore, all the policies that decluster retain a greater portion of the database in memory, and therefore have a lower miss ratio.

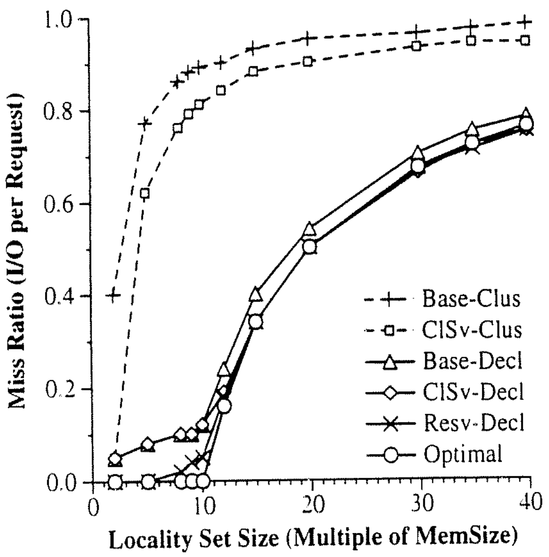

Figure 20: Data Size: I/O

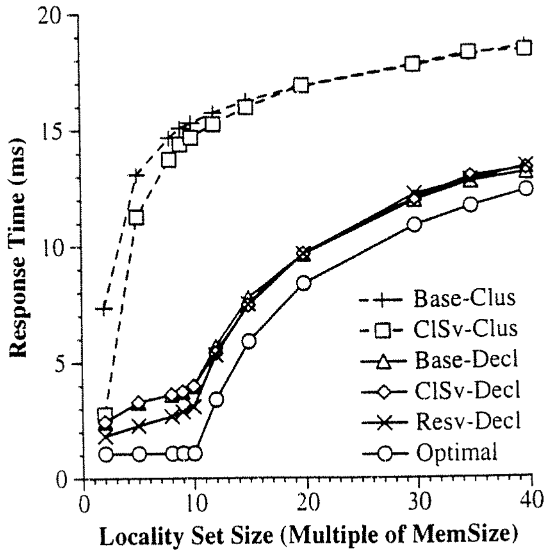

Figure 21: Data Size:Response Time

Among the policies that decluster, ClSv-Decl and Base-Decl have a higher miss ratio than Resv-Decl. This is because, with ClSv-Decl and Base-Decl, when the primary server discards a local page, disk I/O is incurred the next time the page is accessed. Resv-Decl incurs no local disk I/O as it reserves memory for local pages and discards pages owned by remote servers. Since remote servers have idle memory, the discarded pages are cached in memory at the owner. Therefore, no disk I/Os are incurred when these pages are reaccessed. The policies that employ the ClSv algorithm have a lower miss ratio in comparison to the policies that employ the Base algorithm, because, the use of hate hints eliminates the duplication between the primary server and the owner(s) for ClSv. The miss ratio for the policies that decluster remains constant until the size of the locality set exceeds 10. This corresponds to the point where the size of the locality set is equal to the size of aggregate memory. From this point the miss-ratio increases sharply, because, disk I/O's become necessary even when the aggregate memory is fully utilized, because, the size of the locality set is much larger than the size of aggregate memory.

The response time graph shown in Figure 21 essentially reflects the miss ratio behavior. The policies for which the locality set is declustered have a much lower response time than the policies where the locality set is clustered. The response times for the policies that use Base and the ClSv algorithms are about the same even though Base has a higher miss ratio than ClSv. Resv-Decl has the best response time, and is close to optimal, as it uses idle memory at remote servers very efficiently. In the remaining experiments we do not present the response time of the Base policies as ClSv is superior to it.

Expt 2.1: Hot-Cold

In the previous experiment the accesses of the large client are directed to a locality set that is uniformly hot. In order to study the effect of cold accesses, that experiment is repeated with the 90% of the client accesses being directed to a hot region of the locality set and the remaining 10% to the cold region. The size of the hot region is varied, and the size of the cold region is set at 5 times the size of the hot region.

Figure 22 shows the response time as a function of the size of the hot region of the locality set. Once again, the policies that decluster have a much better response

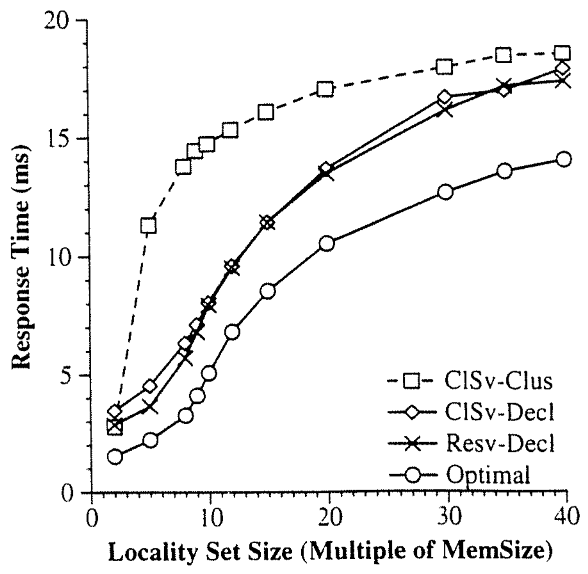

Figure 22: Hot-Cold: Response Time

time than the policies that cluster. The only difference in the response time between a uniformly hot locality set and a hot/cold locality set is the higher response times of all policies. The reason for this is the disk I/Os resulting from the access to the cold region of the locality set. We do not model cold accesses in the rest of the experiments as the only effect it has is to reduce the idle memory at that server available to cache the pages of the hot locality set.

Expt 2.2:Small Clients

The purpose of this experiment is to study the response time of the large client as the amount of idle memory to cache pages of the large client is reduced. In this experiment, all the servers, except the one associated with a large client, serve a small client. The small client accesses a locality set that occupies 40% of a server's memory. The presence of the small client at a server, reduces the idle memory to cache the locality set of the

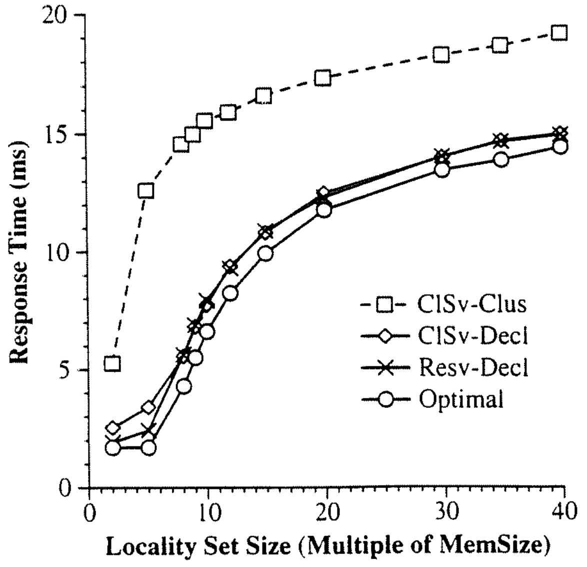

Figure 23: Small Clients: Response Time

large client. Figure 23 shows the response times of the large client when there are 9 small clients. All the servers except the server with the large client, host a small client. The relative performance of the policies remains the same. The policies that decluster perform better than ClSv-Clus. Resv-Decl has the best response time, which is close to that of the "Optimal" policy. However, the response time of all the policies are higher in the presence of small clients. This is because, the presence of small clients reduces the idle memory available to cache pages of the large client.

Expt 2.3: Writes

In this experiment 80% of the client accesses are reads and 20% of the accesses are updates. Figure 24 shows the response time as a function of the size of the locality set. As before, the response time of ClSv-Clus is worse than the response times for the policies that decluster. In comparison to those in Expt 2, the response times for all

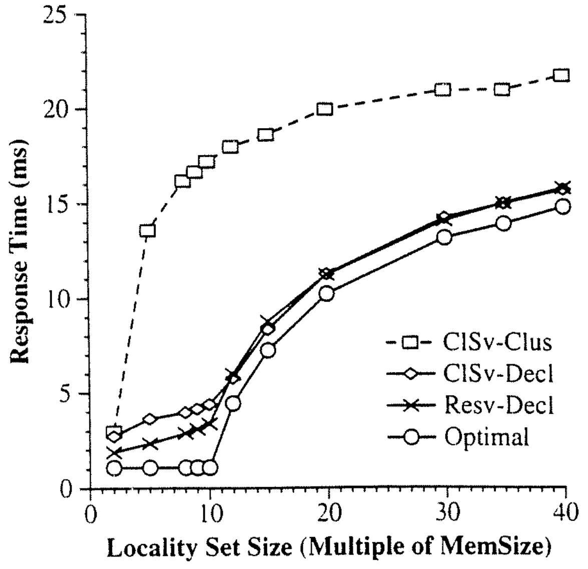

Figure 24: Writes: Response Time

the policies are higher , because dirty pages when discarded, must be written to disk. The effect on response time is more pronounced for ClSv-Clus than Resv-Decl or ClSv-Decl, because, a lot more pages are discarded from global memory by ClSv-Clus. This experiment essentially shows that the ill effects of writes will be minimized by caching pages in memory longer. Briefly, when a dirty page is cached for a long time in memory across transaction boundaries, the page when written to disk reflects the updates by several transactions to the objects on the page. However, if the page is in memory for a shorter period of time, the page will be fetched into memory every time it is updated, and written to disk every time it is discarded from global memory. Therefore, prolonging the life of a dirty page in global memory, by caching more pages in global memory, reduces the number of writes, as it has the effect of batching updates on a page to disk. This is exactly why the effect of writes is less pronounced on Resv-Decl, and ClSv-Decl, when compared to ClSv-Clus.

3.5.4 Expt 3: Number of Nodes

In this experiment, we study the effect of increasing the number of servers on the response time of a single client accessing a large locality set. The size of the locality set is fixed at twelve times the size of memory, and the number of servers caching the pages of the locality set is varied. When the locality set is declustered, increasing the number of servers, for a fixed sized locality set is equal to the effect of increasing memory to cache the locality set.

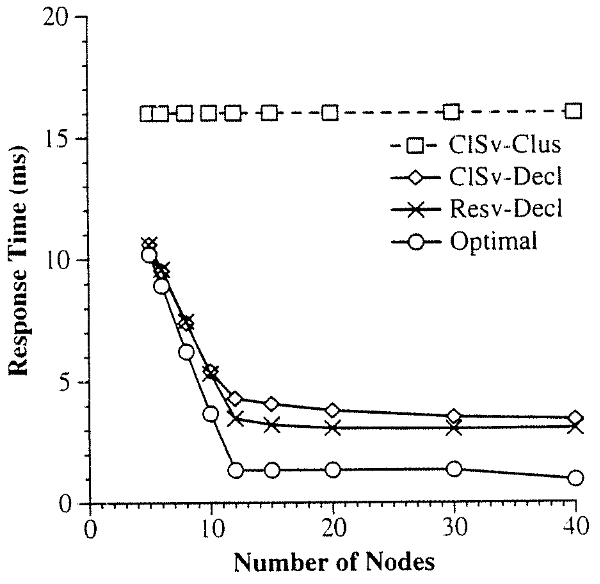

Figure 25: Number of Nodes: Response Time

Figure 25 shows the response as a function of the number of nodes. The response time for policies that decluster decreases sharply at first before flattening out. The knee of the curve represents the point where the size of the locality set is equal to the size of the aggregate memory. Beyond this point, addition of nodes does not reduce the response time as all pages are cached in global memory. Among the policies, Resv-Decl has the best response time and ClSv-Clus has the worst response time. The response time for ClSv-Clus is unaffected by the number of nodes as it only makes use of memory at the primary server and the owner. The response for ClSv-Decl is marginally higher than that of Resv-Decl, because, ClSv-Decl incurs local disk I/Os, while Resv-Decl does not. The response time gap between Resv-Decl and ClSv-Decl gets smaller as the number of nodes increases, because the fraction of the accesses to the local data reduces as the number of node increases. This is because, the fraction of the data resident at the primary server decreases as the number of nodes increase.

3.5.5 Expt 4: Multiple Large Clients

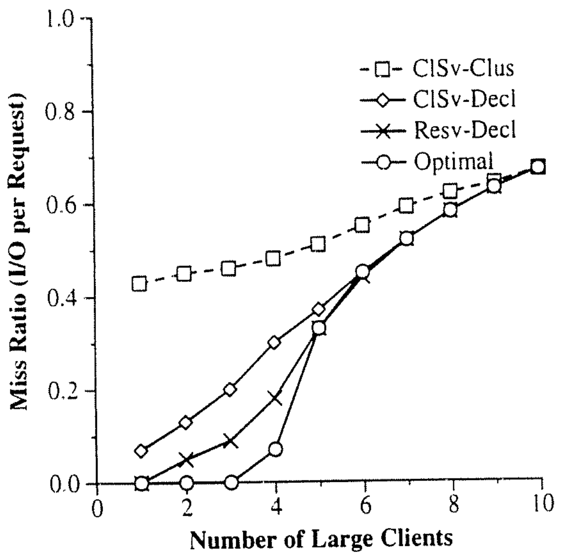

Figure 26: Many Clients: I/O

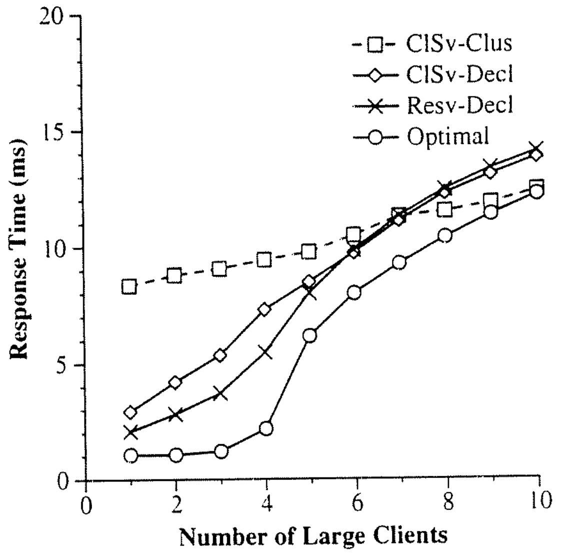

Figure 27: Many Clients: Response Time

In the first three experiments we show that one client makes good use of global memory when data is declustered, and Resv-Decl has the best response time. When there are more clients, ClSv-Clus begins to make use of the memory of more servers, while the idle memory available to cache the pages of the locality set decreases for the policies that decluster the data. Therefore, we expect the performance gap between declustering and clustering to shrink as the number of clients is increased.

To study this, the number of clients is varied from 1 to 10 (one client per node), with each server serving at most one client. Each client accesses data from a private locality set that is three times the size of a server's memory. Therefore, when there are more than three clients, the total size of the locality sets, accessed by all the clients, exceeds the size of aggregate memory.

Figure 26 shows the miss ratio as a function of the number of clients. The miss ratio for ClSv-Decl and Resv-Decl are much lower than that for ClSv-Clus. Initially, the miss ratio for the policies that decluster are very low compared to the policies that cluster. When there are more than three clients, the miss ratio increases sharply as the sum of the size of the locality sets of the client is larger than the size of aggregate memory. The miss ratio of the "Optimal" policy equals that of the policies that decluster, because, all the policies make use of memory at all the servers, and there is no idle memory that remains unutilized by the policies that decluster. The miss ratio of the policies that decluster and ClSv-Clus becomes equal when there are 10 clients, as each server has a client (even when the policies cluster data) associated with it, and this client makes use of memory at that server. Therefore, the memory at all the servers is fully utilized.

Policies that decluster have a lower miss ratio than the policies that cluster, because, with declustering each client makes use of the memory at its primary server and shares the use of memory at other servers that do not serve a client. When data is clustered a client makes use of memory of at most two servers, the primary server and the owner. Resv-Decl has a lower miss ratio than ClSv-Decl as it reserves memory for pages that belong to the local disk. Resv-Decl prevents local disk I/Os by reserving memory for local pages and pushing data owned by remote servers to make use of idle memory at those servers.

Figure 27 shows the response time as a function of the number of clients. The response time graph essentially reflects the miss ratio behavior. The response times of ClSv-Clus is nearly four times that of the Resv-Decl when there are 2 clients. However, when the number of clients is closer to 10, the response time of Resv-Decl and ClSv-Decl are marginally higher than that of ClSv-Clus. This is surprising given the fact the miss ratio for the policies that decluster is lower than that of ClSv-Clus. The reason is that the average disk access times is higher when data is declustered (15.4 ms) compared to 13.57 ms when the data is clustered. When the data is declustered, a request from a client goes to any of the servers. As a consequence, there are several instances when more than one client directs its request to a page on the same disk. The resulting queuing delay increases the average disk I/O time. This does not happen when data is clustered, because, accesses from a client are always directed to the same server. Our experiments with larger number of disks per node show that the queuing is reduced by increasing the number of disks per node.

To summarize, the policies that decluster, ClSv-Decl and Resv-Decl have much better performance than ClSv-Clus when the system is lightly to moderately busy (less than 8 clients). When the system is heavily loaded, ClSv-Decl and Resv-Decl have a marginally higher response time than ClSv-Clus, due to higher disk response times.

Expt 4.1:Large Locality Set

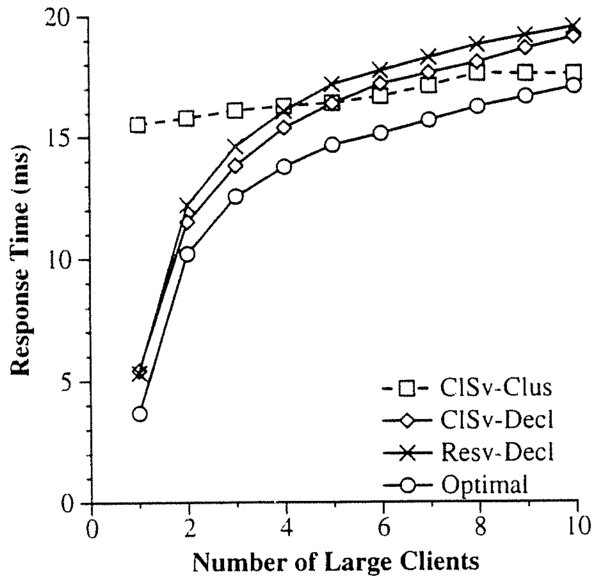

Figure 28: Large Data Set: Response Time

We repeat the above experiment with each client accessing a locality set that is twelve times the size of memory. The response time graph is shown in Figure 28. ClSv-Decl and Resv-Decl perform much better than ClSv-Clus when there are few clients, but its response time increases faster as the number of clients are increased. This is because the sum of the sizes of the locality sets is much larger than the size of aggregate memory. ClSv-Decl has a worse response time than ClSv-Clus when there are a large number of clients. Resv-Decl does not perform better than ClSv-Decl as the aggregate memory is fully utilized even with ClSv-Decl. The only difference is that Resv-Decl incurs remote disk I/O, while ClSv-Decl incurs local disk I/O. This is the reason for the marginally higher response time for Resv-Decl over ClSv-Decl.

3.5.6 Expt 5:Sharing

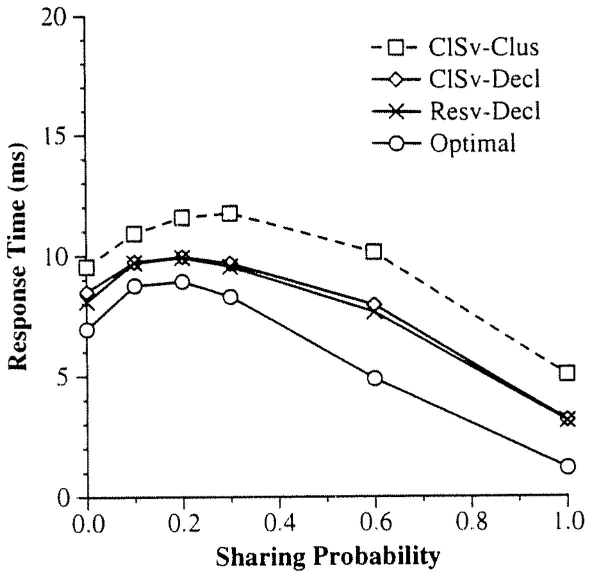

Figure 29: Shared: Response Time

In this experiment, we explore the effect of sharing on idle memory. Five large clients share access to a single locality set that is three times the size of memory. They also access data from private locality sets each of which is three times the size of memory. The response time is measured as a function of the probability of accessing the shared locality set. When the probability of sharing is low, accesses to the shared locality set behave as cold accesses. The response times for all the policies increases initially before decreasing. As the probability of sharing increases, more accesses are directed towards the shared region. ClSv-Decl and Resv-Decl have a lower response time than ClSv-Clus, because, when the shared locality set is declustered, the memory at all the servers are used to cache the pages of the shared locality set. With ClSv-Clus, only the memory of the server where the shared locality set resides is used to cache its pages.

ClSv-Decl and Resv-Decl have a higher response than Optimal, because, both these policies duplicate the pages of the shared locality set in the memories of the five primary servers. Even though the memory at the five primary servers combined can hold the entire shared locality set, they do not cache the entire locality set due to duplication. The Optimal policy on the other hand, has only one copy of the pages of the shared locality set in memory and therefore caches the entire locality set in memory. The effect of duplication is studied in further detail in Chapter 5.

3.5.7 Summary of Results

To summarize, when there is a single large client and data is declustered, simple memory management policies like ClSv-Decl and Resv-Decl are sufficient to utilize the aggregate memory. By reserving memory, Resv-Decl avoids local disk I/O by making use of idle memory at remote servers, and therefore has a lower response time than ClSv-Decl. When data is clustered, only the memory at two servers, the owner and the primary server are used to cache pages, the memory at the rest of the servers are "idle", and this leads to very poor response time for ClSv-Clus. When data is clustered, there is a need for greater interaction between the servers to identify servers with "idle" memory and utilize them. The relative performance of the policies are unaffected by cold accesses, or small clients, or writes. Policies that decluster perform much better than the policies that cluster as they make better use of the aggregate memory to cache their pages.

When there is more than one large client, the response time of ClSv-Clus is several times that of ClSv-Decl and Resv-Decl. The miss ratio and response time of Resv-Decl is better than that of ClSv-Decl and it is closer to Optimal. No matter how well the memory is managed, it is not possible to achieve the response time of the Optimal policy. because the message latency assumed for the Optimal policy is zero (all of memory is local). When there are a large number of clients, for example ten in our workload, the response time of Resv-Decl and ClSv-Decl is worse than that of ClSv-Clus, even though they make better use of global memory. This is due to higher disk response time when data is declustered. When clients share access to a locality set ClSv-Decl and Resv-Decl do better than ClSv-Clus as they are able to make use of memory at all the servers to cache the pages of the private and shared locality set. However, their performance is worse than Optimal due to duplication.

In this study, the memory reserved for the Reserve algorithm is ad hoc. However, we demonstrate that reservation is important to utilize idle memory at remote servers. In the Chapter 4, we design and study a new replacement algorithm, Global, that reserves memory for local and client data dynamically.

3.6 Conclusions

In this study we demonstrate that data placement is an important factor in determining how idle memory is utilized in a symmetric multi-server OODBMS. Through simulations on a variety of workloads we show that when the locality sets are declustered, simple memory management algorithms like ClSv and Reserve are enough to efficiently utilize the aggregate system memory. In contrast, when the locality set is clustered, simple algorithms, like Base and ClSv, are not sufficient to effectively use the aggregate memory in the system; complex memory management algorithms that identify and utilize idle memory are necessary.

Chapter 4

Idle Memory: Implementation

4.1 Implementation Details and Optimizations

In this section, we present the implementation and optimization techniques that we incorporate into the memory management algorithms, when moving from the simulation to the implementation. These are necessary, because, some of the assumptions made in the simulation regarding: the availability of a global clock, the CPU cost, the message latencies, etc, may not be valid on the implementation platform.

4.1.1 Global Time

Distributed systems do not have a globally synchronized clock. However, to implement the page replacement algorithms, it is necessary for a page to have a time stamp, that indicates the time the page was last accessed. The time stamp on the page is used to order the page in the LRU list. When a page is sent from one server to another, the time on the page becomes invalid as the clocks across nodes are not synchronized. Fortunately, the important information that the time stamp conveys for page replacement is the elapsed time since the last access to the page, and not the absolute time. The elapsed time since the last access to the page is piggy-backed along with the page. Specifically, at the sending end, the difference between the current time and the time stamp on the page is sent with the page. At the receiving end, the time stamp on the page is updated by taking the difference of the current time and the elapsed time value that is shipped with the page. The time stamp on the page now has a local time that roughly corresponds to the time the page was last accessed. This enables us to maintain an approximate global time stamp on a page across server boundaries, without a synchronized global clock.

4.1.2 LRU List

In order to implement the page replacement algorithms, it is necessary to maintain multiple LRU lists (hated, duplicate, single, and reserved list). A page moves from one list to another, when its state changes, for example, when a page becomes a last copy in memory, a duplicate page becomes a single page, and it has to be moved from the duplicate list to the single list. In doing so, the LRU order must be maintained. Since the size of each list is proportional to the size of memory, traversing the list to insert a page entry is very CPU intensive. For example, when the size of memory is 8 MB (about 2,000 pages) it takes an average of about 700 μs for inserting into a list. To solve this problem, we associate an array index structure with each list. The index is based on the time stamp on the page, and it is used to quickly locate the place where a page is to be inserted into the list. Each entry in the array corresponds to the LRU time stamp of one hundred pages. To insert a page, instead of traversing the entire list, the array structure is traversed to locate the page with a higher time stamp than that of the page to be inserted. This is followed by a search of at most 100 entries on the list. Since the array structure has at most 20 entries, for 2000 pages, this reduces the insertion time into the list to 20 μs.

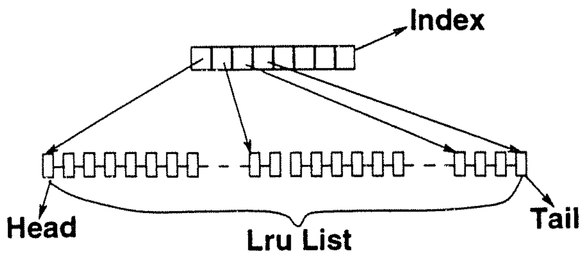

Figure 30: LRU List

4.1.3 Message Cost

In the simulator, the interconnect is assumed to be fast, with a message latency of 550 μs. In an implementation platform, the message costs may be expensive, and it may be necessary to reduce the number of messages for good performance. In this section, we describe some techniques that we implement to minimize the number of messages.

Message costs consists of two components, a fixed message overhead that is very high, and a per-byte overhead that is low. For example, Figure 31 shows the one way message latency for various message sizes using TCP/IP on the IBM SP/2. The one way message latency for an 8 KB page is 1.65 ms. This consists of a very high initial message overhead of 0.9 ms and a small per byte overhead. The message latency increase for every additional byte is small when compared with the fixed overhead .

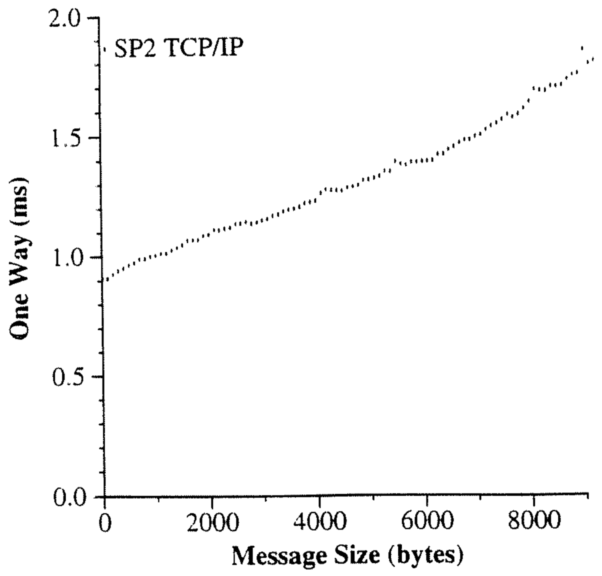

Figure 31: One Way Latency

Performance will improve, if the number of messages are reduced and the data transmitted with each message is increased. We take advantage of this in the implementation by piggy-backing information with each message wherever possible, instead of sending explicit messages. This includes messages for maintaining duplicate information, maintaining directory structures, gathering state information, and messages for sending pages back. These optimizations are described in detail here:

- **Maintaining Duplicate Information:** If each server knows the pages that are duplicates in its buffer pool, messages will be minimized by not shipping these pages back to the owner when they are discarded. Contrary to expectations, this information can be maintained with minimal message overhead. A low message overhead algorithm for maintaining this information is one of the contributions of this thesis.

The directory structure at each server has information regarding the location of each page that it owns in global memory. We use the updates to the directory to trigger state change messages to maintain duplicate information. The only times a server has to be informed about state changes to a page (duplicate to single, or single to duplicate) in its memory by the owner are: 1) when a page becomes a duplicate after being the only copy in global memory, and 2) when the page becomes the last copy in global memory after being a duplicate. There is no need for the owner to inform the servers of state change to a page if the number of copies of the page is greater than one. The state change information is conveyed to the server by the owner, in the following manner:

1. A page makes a transition from being the only copy in global memory to being a duplicate, when the owner requests the server with the only copy of the page in memory to duplicate and forward the page to another server. The server duplicating and forwarding the page moves the copy of the page in its memory to a duplicate list, and piggy-backs the information that the page is a duplicate along with the page transfer. The server receiving the page, places the page on the duplicate list.

2. When a page is discarded by a server, the owner is informed about it so that the directory at the owner is updated. This information is piggy-backed with other messages directed to the owner. The owner, when it receives this information, updates the directory and checks to see if this causes a page at

some other server to become last copy in global memory. If so, it has two options to choose from to inform the server with the last copy: 1) it can wait and piggy back this information along with another message, or 2) it can send a message immediately. If the owner piggy-backs this information, no additional message is necessary. However, this results in inaccurate duplicate state information at the server, and may cause wrong replacement decisions at that server. The period of time such inaccuracies last is generally at most only a few milliseconds, and consequently incorrect replacement decisions due to this are very rare. Therefore we piggy-back this information so that the common case is faster, instead of sending an explicit message.

- **Maintaining State Information:** To implement the Global algorithm, each server has to maintain a summary about pages in memory at every other server. The summary consists of the following information: number of hated pages, number of duplicate pages, number of single pages, and the average elapsed time of the tail of the LRU list of all the servers. We define the tail of the LRU list as the 100 least recently used pages on that server. The average elapsed time since last access to the tail of the LRU list is referred to as the tail temperature. The tail temperature is used to identify servers with the global LRU page. Maintaining exact summary information up to date at each server is very expensive. Instead we maintain approximate state information at each server. This is maintained by piggy-backing the summary information with every message.

- **Sending Pages Back:** In the simulation, when a server discards a page, a message is sent to its owner. This message serves dual purposes: 1) it enables the owner to update the directory, and 2) it allows the owner to request the page be shipped back to it if the page happens to be the last copy in memory, and the LRU page at the owner has a lower time stamp than the discarded page. If the owner does not need the page, the page is discarded from memory by the primary server. This handshake takes place in the critical path of the request. If messages are expensive, this handshake had to be eliminated in the implementation.

  To avoid message overheads in the implementation, updating the directory, and sending a page back when it is discarded, are handled separately. To maintain the directory, the page and the owner identifier of the page being discarded are cached until a message is sent to the owner. The page identifier is piggy backed with messages sent to the owner. The owner, upon receiving this information, updates its directory. As a result of this, the owner's directory has approximate information on the number of copies of a page in global memory. The inaccuracy of the directory structure may result in race conditions where the owner requests a page to be forwarded from a server after the server has replaced the page. When this happens, the owner may have to re-service the request. Our experiments show that the window of opportunity for such a race condition is fairly small, and that it happens very rarely.

  A page that is sent back to the owner is not guaranteed to be retained in the owner's memory for various reasons. If the server makes this evaluation before shipping the page back, these messages will be avoided. In order to reduce these messages, we use the following criteria to determine whether it is necessary to ship a discarded page back to the owner.

1. If the page to be discarded has a duplicate copy elsewhere in memory, then the page need not be shipped back to the owner. The server determines this by maintaining duplicate information as described earlier.

2. Using the state information maintained at each server, a server determines if the owner has duplicate or hated pages, or if the temperature on the discarded page is lower than the average elapsed on the tail of the LRU list at the owner. If the owner has no duplicate or hated pages, and if the elapsed time since last access to the discarded page is greater than the average elapsed time of the tail at the owner, the page is not sent to the owner.

If neither of these two conditions are true, the discarded page is shipped back to the owner. The chances of the page being retained in the owner's memory is very good.

4.2 Page Replacement

The ClSv algorithm and the Reserve algorithm are described in the previous chapter. These algorithms are implemented along with the optimization described in the previous section. In the implementation, the Reserve algorithm is modified so that it uses the information about duplicate pages to reduce duplication between the owner and the primary server. We also propose a new page replacement algorithm, Global, that comes close to mimicking global LRU, when the data is declustered. In this section, ClSv algorithm is briefly summarized, followed by the description of the Reserve and the Global algorithm.

4.2.1 Client-Server:ClSv

Hated pages are replaced first, followed by the LRU replacement of the rest of the pages in the buffer pool. A replaced page is shipped back to the owner, as explained in Section 4.1.3.

4.2.2 Reserve

The Reserve algorithm described in Chapter 3 is modified to make use of the approximate duplicate information maintained at each server to reduce duplication between the primary server and the owner. A portion of memory is reserved (60% for the experiments) for local data. The reserved portion consists of hated pages, and single pages, owned by the server.

In this replacement strategy, duplicate pages are replaced first, because, eliminating duplicate pages makes use of idle memory at the owner that is in the form of hated pages. In the absence of duplicate pages, if the number of local pages exceeds the portion reserved, the hated pages are replaced, followed by the LRU local page. Otherwise, the LRU page that is owned by a remote server is replaced.

4.2.3 Global

In Section 3.3, we showed that declustering combined with ClSv is insufficient to manage global memory efficiently, because, hot pages get discarded from memory at servers that are busy, while cold pages are cached in memory at servers that are idle. The Reserve policy attempts to solve this problem by allocating a fixed block of memory at each server for local data, so that an LRU page that is discarded from a server finds room in the owner's memory even though the owner may be busy serving other clients. The problem with Reserve is, if one of the server has a client with a locality set smaller than the server's memory, only a small portion of the small client's locality set is cached locally with Reserve, even though most of it could be cached locally with ClSv. The small client has to frequently access pages from remote memory. This severely hurts the performance of the small client. However, if the server with the large client discards pages owned by servers with idle memory and caches pages owned by servers that are busy, then the performance of the small client will remain unaffected and global memory will be fully utilized. This is the rationale behind the Global algorithm.

Global, approximates global LRU policy by caching pages owned by servers that are busy and discarding pages owned by servers with cold pages. This is because, replacing the page owned by the server that has cold pages, is equivalent to locating the server with the global LRU page and replacing that page. By replacing the cold pages, the page that eventually gets discarded from memory is the globally LRU page. When a server chooses a page for replacement, it chooses the page owned by the server with the globally LRU page, so that when this page is shipped back to the owner, the globally LRU page is replaced.

Global policy chooses pages in the following order for replacement:

- Hated pages
- Duplicate pages
- Single pages owned by servers that have hated or duplicate pages
- If there are no pages in any of the above categories, an idle page is chosen based on the following rules:
    1. Choose the LRU page L
    2. Choose the LRU page (X) that is owned by the server that has the global LRU page (O) (lowest tail temperature). This is determined from the state table. If there is no such page X in the server's memory choose the page owned by the server that has the second lowest tail temperature and so on.
    3. If the elapsed time of L is greater than that of O, then L is replaced, otherwise X is replaced. If X is discarded, it replaces O from the owner's memory

Global replaces hated and duplicate pages first. In the absence of these pages it discards pages owned by servers that have duplicate or hated pages. Hated and duplicate pages are given a high priority for elimination as cost of accessing these pages from remote memory is very low, and eliminating them increases the percentage of the database in global memory. By discarding the pages owned by servers that have duplicate and hated pages, Global, increases the database in memory and increases the chances for the pages that are discarded to be cached in the owner's memory.

In the absence of any pages in these categories, the server locates the page owned by the server with the globally LRU page. First, it locates the server with the globally LRU page by sorting the tail temperatures of all the servers in ascending order. It does this from the approximate global state information maintained in the state table at each server. If the tail temperature of the server replacing the page is the lowest, then the LRU page on this server is replaced. Otherwise, it looks for a page owned by the server with the global LRU page. It does this by going up the sorted list of LRU tail temperatures in ascending order and chooses the LRU page owned by the server with the lowest tail temperature. This page is discarded, and it replaces the LRU page on the server with the lowest tail temperature.

To implement this algorithm one LRU list for each category of pages is maintained. LRU list, for hated and duplicate pages plus a LRU list for pages owned by each of the servers, are maintained. Each server also maintains approximate state information to determine the server with the global LRU page.

Page Replacement	Data Placement	Policies
ClSv	Clustered	ClSv-Clus
ClSv	Declustered	ClSv-Decl
Reserve	Declustered	Resv-Decl
Global	Declustered	Global-Decl

Table 10: Policies

4.3 Policies

We implement and evaluate the policies that are combinations of data placement and memory management algorithms. Table 10 summarizes the policies that are evaluated.

4.4 Implementation Platform

We implemented the memory management algorithms in the SHORE database system, on a 16 node IBM SP/2, with the modification and optimizations described in this chapter. The IBM SP/2 is based on the superscalar IBM RS/6000 processors. Fast inter-node communication on the SP/2 is provided by a multistage packet switching network with a hardware capacity of 40 MB/sec duplex transfers between nodes. The switch supports direct DMA transfers between the processors memory and the switch, reducing the network latency.

One main concern for performance is the message latency of TCP/IP on the IBM SP/2. Due to compatibility problems between the threads used by SHORE and the Message Passing Interface (MPI) on the SP/2, we use TCP/IP for communication. The performance of TCP/IP on the SP/2 is much worse than that of MPI on the SP/2. The latency to send an 8 KB message from one node to another using TCP/IP is 1.65 ms,

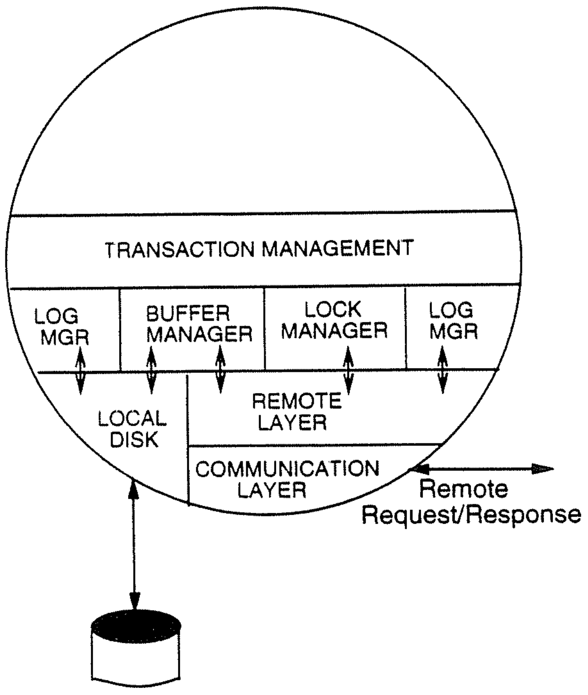

Figure 32: Server Architecture

and the throughput sending 8 KB messages using TCP/IP is 81.6 MB/sec. However, the message costs with TCP/IP on the IBM SP/2 is sufficiently low to make remote memory accesses (2.65 ms) cheaper than local disk I/O (9.9 ms). We also implement the optimization techniques described in this chapter to minimize the message overheads for maintaining state information.

The architecture of the SHORE storage manager is shown in Figure 32. The "remote" layer is responsible for coordinating with the other servers in the cluster. It manages connections, communication, reads pages from remote servers, and manages distributed transactions (including distributed locking, recovery, and two phase commit). The buffer manager directs remote read requests to other servers through the "remote layer". Remote lock requests and log pages to remote servers also goes through this layer.

The following changes were made to implement the memory management algorithms in SHORE, on the IBM SP/2.

- The buffer manager that sits on top of the remote layer and the disk layer was modified to implement the page replacement policies.
- Every message is piggy-backed with information, to update the directory structure, the state table, and maintain duplicate information.
- Forwarding pages and sending discarded pages back to the owner is implemented in the remote layer.

Most of the implementation and modifications to the existing code are fairly straightforward. We also implement the techniques and the optimizations described in Section 4.1.

4.5 Performance

We run experiments on the SHORE database system on a 16 node IBM SP/2. The hardware parameters are shown in Table 11. We used ten SP/2 nodes, one disk per node, and an 8 MB buffer pool, on the IBM SP/2. We use a buffer pool of 8 MB as the time it takes to warm up larger sized buffer pools is high. We also performed an experiment with larger buffer pool, and scaled up database and show that the size of the buffer pool does not impact performance, and it is the relative size of the database and memory that is important. Raw disks are used to bypass the operating system file cache.

Parameter	**Value**
Nodes	10
Memory (M)	8MB/Node
Disks	1 Disk/Node
Disk Latency	9.9 ms
Msg Latency	1.30 ms/page
Link Bandwidth	81.6 Mbits/sec

Table 11: Hardware Parameters

The size of each page is fixed at 4 KB. We use 4 KB pages as message costs are high with 8 KB page. We run experiments with the synthetic workloads similar to those used in the simulation study. Average request response time is used as the primary performance measure.

4.5.1 Workload

The synthetic workload is similar to the one used in the simulation study described in the previous chapter. The workload consists of two kinds of clients, a large client that accesses data from a large private locality set, and a small client that accesses data from a small private locality set. The size of the locality set of the large and small clients are parameters to the experiment. Each locality set has hot and cold pages. The probability of accessing pages in the hot set is given by P_{hot}. The pages within the hot and cold set are accessed with uniform probability. Table 12 illustrates some of the workload parameters.

Parameter	Value
Locality Set (L)	M to 20*M
Cold Region	4*L
Request Size	Page
P_{hot}	1.0 or 0.9
Execution Time (T)	400-800 secs
Warmup Period	Half of T

Table 12: Workload Parameters

4.5.2 Road Map

The first two experiments are repetitions of experiments 2 and 4 in the simulation study described in the previous. In the first experiment, we study the performance of one client that accesses data from a large private locality set that is uniformly hot. In the second experiment, we study the effect of increasing the number of large clients. Following these two experiments, we run experiments to further evaluate the policies that decluster the data. In the third experiment, we study the effect of decreasing idle memory by introducing small clients. This experiment is used to test the ability of the algorithms to identify and utilize servers with idle memory.. Following this, we run experiments with clients accessing data from locality sets with both hot and cold pages. This experiment studies if the algorithms make uniform use of memory at all the servers. In the last experiment we test the sensitivity of the response time to larger memory sizes.

4.5.3 Expt 1: Data Size

In this experiment, one client accesses data from a large private locality set. The size of the locality set is measured as a multiple of the memory size of one server. For example, when the size of the locality set is ten, it equals the size of aggregate memory, as there are ten servers in the system.

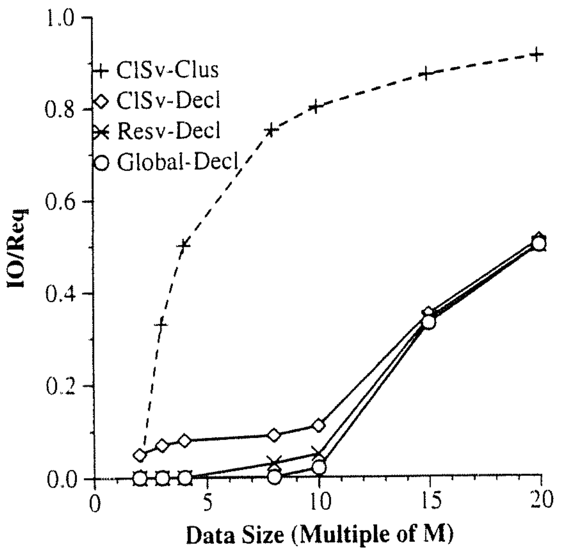

Figure 33: Data Size: I/O

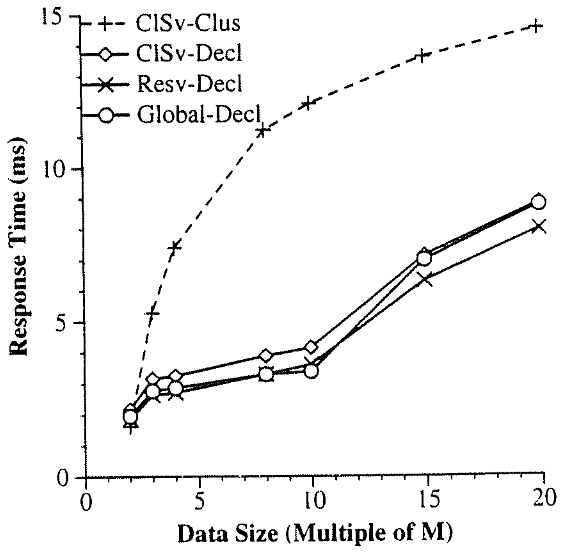

Figure 34: Data Size:Response Time

Figure 33 shows the number of disk I/Os per request as a function of the size of the locality set. It has two distinct sets of lines. The lines corresponding to ClSv-Clus shows that it has a high miss rate, while the other policies that use declustered data have much lower miss rates. The new algorithm, Global-Decl, has the lowest miss rate, and it is close to zero. The miss rate for Global-Decl is zero when the locality set is smaller than the size of aggregate memory. Global-Decl has a low miss rate, because, the primary server caches local pages in preference to pages owned by other servers. Global-Decl caches local pages that have to be read from disk if they are discarded, and discards pages owned by servers that have idle memory. Resv-Decl achieves the same effect by reserving memory for local pages.

Figure 34 shows the response time graph, this graph essentially reflects the miss

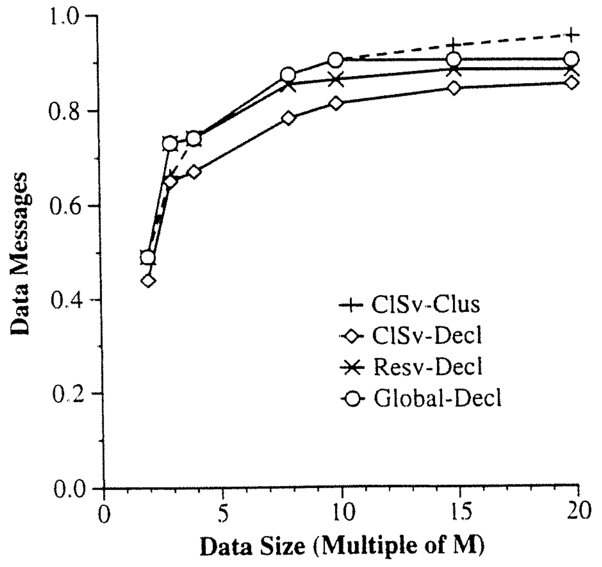

Figure 35: Data Size: Message Cost

ratio behavior. ClSv-Clus has a poor response time, while the policies that decluster have a very low response time. The response time gap between Global-Decl, Resv-Decl, and ClSv-Decl is small even though Global-Decl and Resv-Decl have a much better I/O performance than ClSv-Decl. This is due to the higher message overhead of Global-Decl and Resv-Decl over ClSv-Decl, as they fetch and discard pages from remote memory more often than ClSv-Decl. Figure 35 shows the data messages as a function of the size of the locality set. The message overhead of ClSv-Decl is low compared to those of Global-Decl and Resv-Decl as ClSv-Decl caches more remote pages and fewer local pages. The response time of Global-Decl for larger locality set sizes is higher than that of Resv-Decl, because, the message overhead of Global-Decl is higher than that of Resv-Decl, while the I/O difference between the two is negligible. This is due to the larger number of local pages cached by Global-Decl when compared to Resv-Decl for larger sized locality sets.

4.5.4 Expt 2: Multiple Large Clients

Increasing the number of clients reduces the memory that is available to cache the locality set of each client. It creates a load imbalance in memory usage, as servers with clients have hot pages, while servers without clients have idle memory. Memory mangement algorithms have to identify the servers with idle memory and move data from heavily loaded servers, to servers with idle memory.

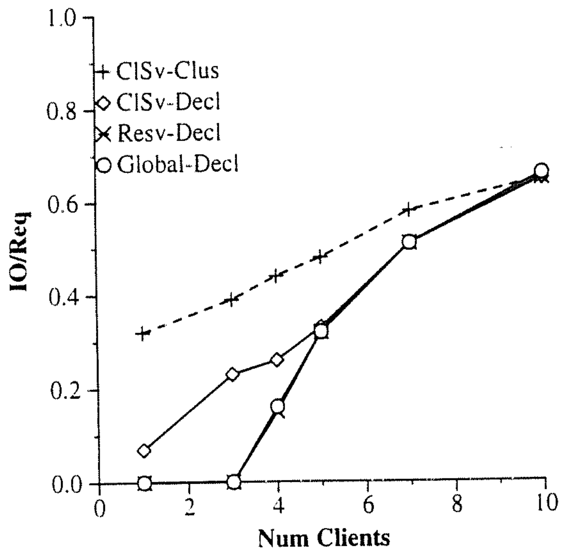

Figure 36: Many Clients: I/O

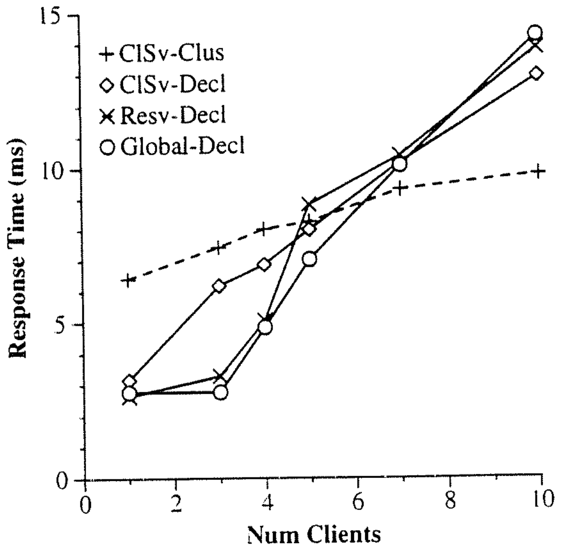

Figure 37: Many Clients:Response Time

In this experiment, each client accesses a private locality set that is three times the size of memory. The pages of the locality set in this experiment are uniformly hot. When there are more than three clients, the size of the combined locality sets of the clients exceeds the size of aggregate memory.

Figure 37 shows the response time as a function of the number of large clients. When there are few clients, the policies that decluster perform well compared to ClSv-Clus. Global-Decl has the lowest response time, because, with Global-Decl, the primary server of a large client caches local pages and pages of those servers with clients in preference to pages that are owned by servers with idle memory (server with duplicate pages and those with the global LRU page). Discarding a page owned by the server with idle memory maximizes the chance for the discarded page to be retained in the owner's memory. The Reserve policy, reserves memory at each server for local data, reducing the memory that is available to cache the client pages, and this increases the number of remote memory accesses. However, pages that are discarded by a server are cached at the owner, as memory is reserved for these pages at the owner. This reduces disk I/Os at the expense of increased message traffic to access remote pages.

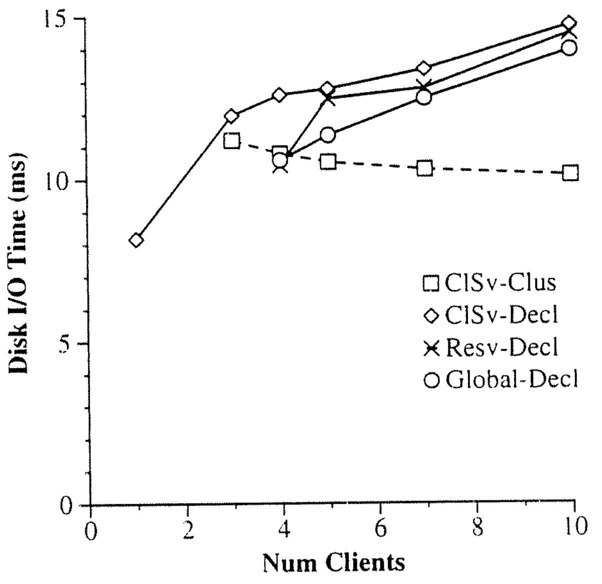

Figure 38: Many Clients: Disk I/O Time

When the number of clients are greater than 7, ClSv-Clus does better than Resv-Decl and Global-Decl. This is surprising, given the fact that ClSv-Clus has a higher miss ratio. Figure 36 shows the I/O graph. ClSv-Clus has a higher miss ratio than the policies that decluster. The reason for this contradictory performance results can be understood by studying the disk performance.

Figure 38 shows the average disk I/O times. When data is declustered, and the number of clients are more than 3. the average disk I/O is 50% worse when compared with ClSv-Clus. confirming the simulation result in Section 3.5.5. When data is declustered, requests are uniformly directed to all the servers, this results in the formation of temporary queues at disks causing poor disk I/O performance. However, when data is clustered, the degree to which disk requests get queued is lower, since each disk receives request from at most one or two clients.

4.5.5 Expt 3: Idle Memory

Having demonstrated the superiority of declustering over clustering, we concentrate on evaluating the three policies that decluster the data. In this experiment. we evaluate the performance of the three policies that decluster the data and study their ability to identify and utilize idle memory.

The workload consists of one large client that accesses data from a locality set that is three times the size of memory. and some number of small clients each of which accesses a locality set 1.2 times the size of a server's memory. Each server has at most one client associated with it. Increasing the number of small clients reduces the idle memory available to cache pages of the large client. Idle memory is represented as the percentage of aggregate memory that is unused after caching the locality set of all the clients. When

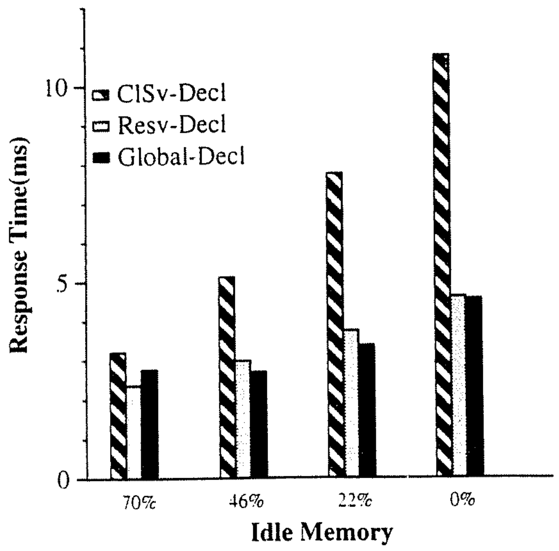

Figure 39: Large Client: Response Time

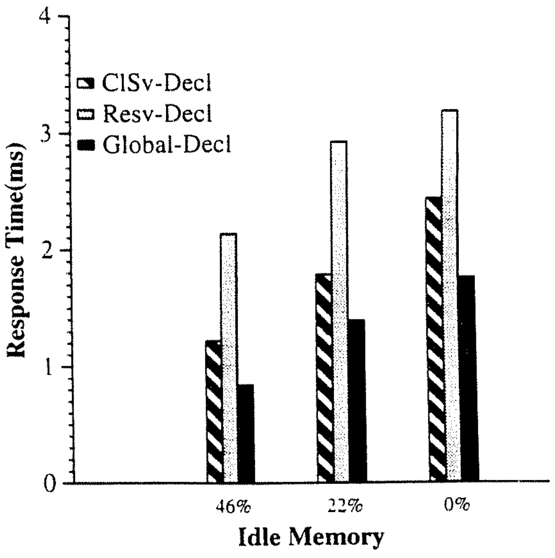

Figure 40: Small Client:Response Time

there are no small clients, the idle memory available is 70% of global memory, as the number of small clients increase to four, the idle memory drops down to 22% of global memory. When there are 6 small clients, there is no idle memory available as the locality sets of the small and large client fits entirely in memory.

Figure 39 shows the response time of the large client as a function of idle memory. Global-Decl has the best response time, followed closely by Resv-Decl, and ClSv-Decl has the worst response time. With ClSv-Decl, each server discards pages owned by busy and idle servers with equal probability. Therefore it does not fully exploit the memory present at idle servers. Global-Decl preferentially discards pages owned by servers that are idle, and caches the pages owned by those servers that are actively using their memory. Therefore it makes very good use of idle memory. Resv-Decl, by reserving 60% of the memory for local data, artificially creates space at the owner for pages that are discarded. This improves the performance of the large client, as the pages discarded by the primary server associated with the large client are cached in the owner's memory. Ideally, if the aggregate memory available is larger than the locality set of the large client, the response time should not increase as the idle memory decreases. Global shows this property as its response time only increases marginally when moving from 70% idle memory to 0% idle memory.

Table 13 shows the break up of the pages cached in the primary server's memory that serves the large client. Server one corresponds to the server hosting the large client, and servers two through five corresponds to those that are associated with the small client. The rest of the servers correspond to the servers that do not serve a client. For ClSv-Decl the pages owned by all the servers are cached with equal probability. With Resv-Decl, 60% of the primary server's memory consists of local pages, the rest of the page frames are divided among the pages owned by the servers that host a client. With Global-Decl, the primary server caches pages owned by the servers that serve a client and discards pages owned by servers with idle memory. Also notice that, Global, reserves more pages for local data, than for pages owned by servers with small clients. The local pages correspond to the pages accessed by the large client and the pages accessed by the small client. These pages when discarded from memory have to be accessed from disk the next time they are accessed, while the pages owned by servers with small clients when discarded may be cached in memory at those servers. Global dynamically allocates its memory between the local data and data owned by servers that are busy. This shows that Global caches pages owned by servers that are busy and discards pages owned by

Policy	**Server**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
ClSv-Decl	274	202	190	201	174	197	191	202	214	202
Resv-Decl	1228	200	196	201	196	4	9	3	7	7
Global-Decl	607	356	334	366	350	9	7	7	5	7

Table 13: Pages Cached at the Primary Server of the Large Client

servers that have idle memory. By doing this, it discards pages owned by servers with idle memory, and makes use of idle memory.

The response time of Resv-Decl is very good for the large client, but the response time is very poor for the small client. Figure 40 shows the response time graph of the small client as a function of the amount of idle memory. By reserving pages for local data, fewer pages of the locality sets of the small client are cached locally. This forces the small client to frequently access data from remote memory. Notice how poorly Resv-Decl performs in comparison to the other two policies. Small clients have the lowest response time with Global-Decl, because, the primary server of the small clients caches most of the locality set in its memory.

4.5.6 Expt 4: Hot-Cold

In this experiment, the locality set of the large client has hot and cold pages. The size of the hot region for the large client is set to three times the size of memory of one server, and for the small client it is set to 1.2 times the size of memory of one server. The size of the cold region of the locality set is set to four times the size of the hot region. The probability of accessing data from the hot region is fixed at 0.9. The accesses to pages

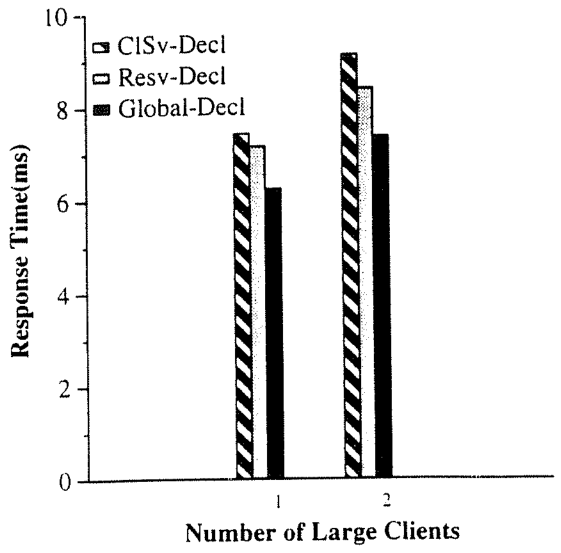

Figure 41: Large Client: Response Time

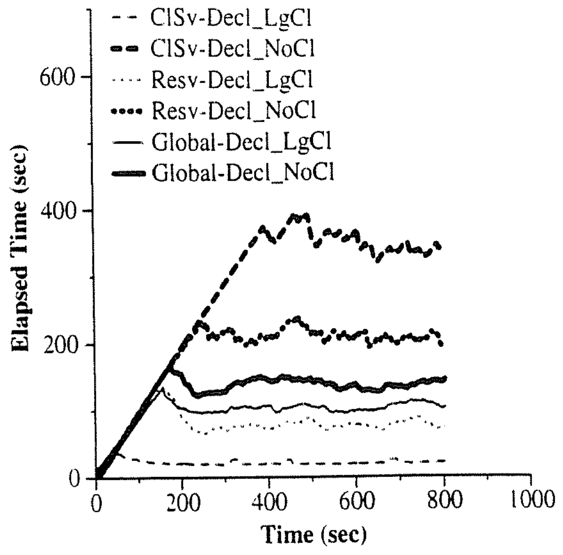

Figure 42: LRU Time

within the hot and cold region are uniform.

We repeated the previous experiment, with one large client and four small clients. Due to the low response times of small client when compared to the large client, the cold pages accessed by the small clients have a lower elapsed time than the hot pages of the large client. Therefore, when these cold pages are discarded, they dislodge the hot pages of the large client. This floods the memory at all the servers with the cold pages accessed by the small client, and displaces the hot pages accessed by the large client. Very few pages accessed by the large client are cached in memory. If LRU were employed in centralized memory, we would see the same effect.

To further study how the policies identify and utilize cold pages, we slow down the accesses of the small client by introducing a think time of 4 ms between each access. Therefore, the pages accessed by the small client are cold compared to the hot pages of the large client. We repeat the experiment for two configurations, the first with one large and four small clients, and the other with two large clients and three small clients.

Figure 41 shows the response time for the large client when there are 1 and 2 large clients. Notice that the response time difference between Global-Decl and ClSv-Decl is lower than in the previous experiment. This is because, the hot region of the locality set of the small-client is colder than the hot locality set of the large client, due to the think time between accesses. Therefore, all the policies including ClSv-Decl are able to make use of the memory of the small client to cache the pages of the large client. This was not possible in the previous experiment.

Global-Decl has the best response time, because, when there is one large client, the primary server caches local pages and discards pages owned by remote servers. When there are two large clients, the primary server caches local pages and pages owned by the server with the other large client. With ClSv-Decl the pages owned by all the servers are discarded with uniform probability. Therefore, when local pages, and pages owned by servers that have the other large client are discarded, they have to be read from disk the next time they are accessed. Resv-Decl has a response time that is better than ClSv-Decl, but worse than that of Global-Decl, because, Resv-Decl does not identify and utilize memory at servers with cold pages.

Figure 42 shows a plot of the average elapsed time of the tail of the LRU list as a function of time, when there are two large clients. The LRU tail temperatures are computed at two second intervals during the course of execution, and plotted against time. In this graph, each memory management policy has two lines, one corresponding to each of the following servers: 1) the server that is associated with the large client (LgCl). and 2) the server without any client (NoCl). For ClSv-Decl, the tail temperature for the server with a client is much higher than that for the server without a client. This shows that the server discards pages that are hot even though there are cold pages at other servers without clients. Therefore, ClSv-Decl does not make use of cold pages available at the other servers. Resv-Decl shows similar behavior, except that the gap between the elapsed time is smaller than that for ClSv-Decl. For Global-Decl, the average elapsed times of the tail of the LRU list at servers with and without the large client is close, indicating that the LRU tail temperatures are uniform at both these servers. This shows that the Global-Decl makes uniform use of memory at all the servers, and is a good indicator that Global-Decl closely mimics global LRU.

4.5.7 Memory Size

In this experiment we increase the size of the SHORE buffer pool to from 8 MB to 16 MB and 32 MB. A large client accesses a locality set that is three times the size of the buffer pool, and small clients access a locality set that is 1.2 times the buffer pool. The size of the locality set scales with the size of the buffer pool.

Figure 43 shows the response time for two buffer pool sizes. The ratio of the size of the locality set to the size of the buffer pool is held constant as the size of the buffer pool is increased. These results indicate that the response times of the three policies does not change significantly. This demonstrates that memory size has no significant impact

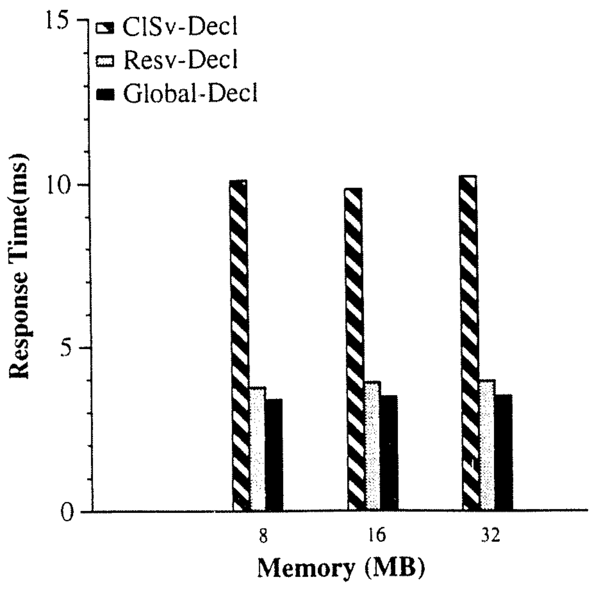

Figure 43: Memory Size

on the response time, and it is the ratio of the memory size to the database size that is important for performance.

4.5.8 Summary of Results

In the first experiment, one client accesses data from a large locality set, the size of which is varied. The policies that decluster data sets were shown to utilize aggregate memory better than ClSv-Clus. Global-Decl has the lowest miss ratio and best response time. In the second experiment, the number of clients in the system are increased to study how the policies utilize aggregate memory. Global-Decl and Resv-Decl have the best performance. The experiments following this are designed to study the robustness of the policies that decluster when the idle memory is reduced, by introducing small clients. Global-Decl is far superior to the other two policies in utilizing idle memory, as each server caches pages owned by servers that have idle memory, and discards pages owned by servers with idle memory. In the fourth experiment, hot and cold regions are introduced into the locality set. We show that the tail temperatures of the LRU list are nearly the same for Global-Decl, while there is a large difference in the LRU times at the servers for other policies. This demonstrates that Global-Decl utilizes memory at all the servers uniformly and closely mimics global LRU. In the fourth experiment, the size of buffer pool is varied and the size of the locality set is scaled appropriately. The response times for all the policies remains similar for scaled up database and memory sizes. This shows that the size of the buffer pool has very little impact on performance, and it is the ratio of the database size to the buffer pool size that is important.

4.6 Conclusions

The memory management algorithms were implemented in SHORE on the IBM SP/2. Implementing these algorithms in SHORE required only modest changes to the SHORE buffer management code. The experiments demonstrated the superiority of declustering over clustering in utilizing aggregate memory. Declustering, combined with simple memory management algorithms has the best performance. Among the policies that we consider, Global-Decl has the best performance, and it mimics global LRU closely by utilizing memory at all the servers.

Chapter 5

Duplicate Management

5.1 Motivation

Popular web sites already experience very heavy loads, and the number of users accessing them is growing exponentially. These loads present heavy demands on many parts of the computer system; with current technology, perhaps the dominant bottleneck is the CPU overhead of servicing TCP/IP connections. Fortunately, the networking and operating system communities are working on this bottleneck. However, lurking below this bottleneck is another: the I/O bottleneck. Simply put, as thousands of users request many large objects (e.g., images, or video), the system must make good use of the main memory buffer pool, or the web site will slow to a crawl waiting to service disk requests.

One promising solution to the CPU bottleneck is to replace a single processor server with a cluster of servers. The National Center for Super Computing Application (NCSA) Mosaic web site [KBM94] has taken exactly this approach. It has a cluster of identical servers connected by an Fiber Distribute Data Interface (FDDI) network to a centralized Andrew File System [HKS$^+$88] (AFS) server. Figure 44 illustrates this configuration. When you request a URL from `http://www.ncsa.uiuc.edu`, the request is actually routed to one of nine HP9000 series workstations. They use a Domain Name Server (DNS) to act as a virtual router or switch through which the requests travel. The DNS round-robins the requests to nine servers in the web server cluster in order to balance the load. This effectively brings the power of nine CPU's to service network connections. Our goal in is to develop buffer management techniques to bring the aggregate capacity of the memories of the machines in the server cluster to attack the I/O bottleneck.

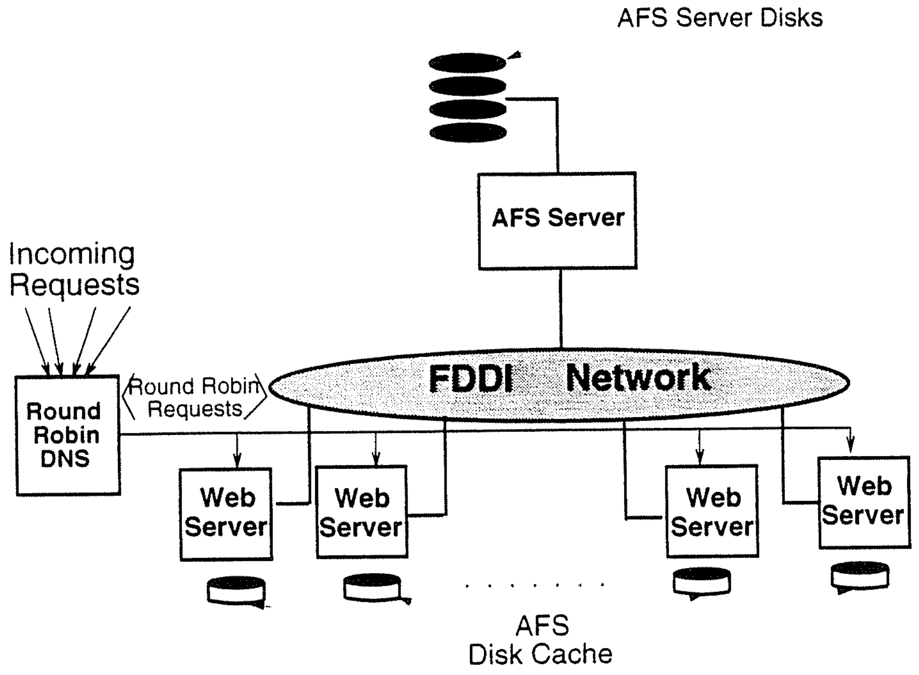

Figure 44: NCSA Server Configuration

Naive approaches to this buffer management problem are considerably suboptimal. For example, one approach is to partition the data over the servers, routing each incoming connection to the machine storing the data requested by the connection. This has the serious flaw of vulnerability to skew: if the data at one processor is significantly more popular than the other, that processor will soon be a bottleneck. Another naive approach that avoids the skew problem is to connect the machines in the server cluster to a centralized file server (this is the NCSA approach), and round-robin the connections among the servers. While this eliminates skew, the workload at all processors are statistically identical, the memories of the machines in the cluster will be filled with data duplicated throughout the cluster. In effect, this reduces the effective capacity of the memory in a cluster of k machines with memories of size M from $k \times M$ to something much closer to M. In this situation the machines would frequently go to the server's disks for data that could have otherwise been obtained quickly from another server's memory had there been no duplication.

Our approach to this problem is to suggest an architecture similar to the NCSA approach, but to store the data on disks attached to the servers in the cluster rather than in a centralized file server. We consider three memory management policies for this architecture. In the first, called "client-server" we simply apply standard client-server memory management algorithm at each node. That is, each processor in the cluster is a server for the data it stores locally; it is a client for the data stored at any other server. Each server applies a simple replacement policy like Least Recently Used (LRU) to the pages in its buffer pool. This approach gives rise to duplication among the buffer pools of the servers in the cluster. Accordingly, we also consider a buffer management policy that eliminates all duplicates: "duplicate elimination". This algorithm works by giving duplicate pages the highest priority for removal from the buffer pool. This algorithm requires that each server know which pages are duplicates and which are not: part of the contribution of this work is a low overhead algorithm to maintain this global information.

In general, neither client-server nor duplicate-elimination provide optimal response times to data requests. On the one hand, as we have discussed, duplicate pages in the buffer pools reduces the effective global memory size, giving rise to increased I/Os. On the other hand, eliminating duplicates means that a server may frequently have to go to another server for a hot page that could have been cached locally if duplicates were allowed, giving rise to increased intracluster network traffic. Thus the key challenge in designing a buffer management algorithm is to control data replication so as to achieve a good balance between the network traffic and disk I/O. Our third memory management algorithm, Hybrid, dynamically controls the amount of duplication in order to minimize the average response time.

5.2 Web Model

We base the architecture of the multi-node web server on the multi-server architecture described in Chapter 2. Figure 45 is an illustration of this architecture. Incoming client requests go through a router that round-robin's the requests among the servers. The intent of using round-robin is to balance the CPU load among the servers. We evaluate the algorithms on workloads that are read-only, since web servers primarily deal with read only data. However, this is likely to change soon, with the advent of online banking, reservation, home shopping etc. We can expect to see a mixture of read-write workloads. To permit read-write workloads, consistency, concurrency control, and recovery guarantees have to be given to clients. These guarantees can be easily added to this architecture as each server can be made responsible for providing concurrency control, coherency, and recovery for the data that they own. Call back locking and coherency strategy that ensures "read multiple, write one" memory coherence described in [WR91] can be adopted.

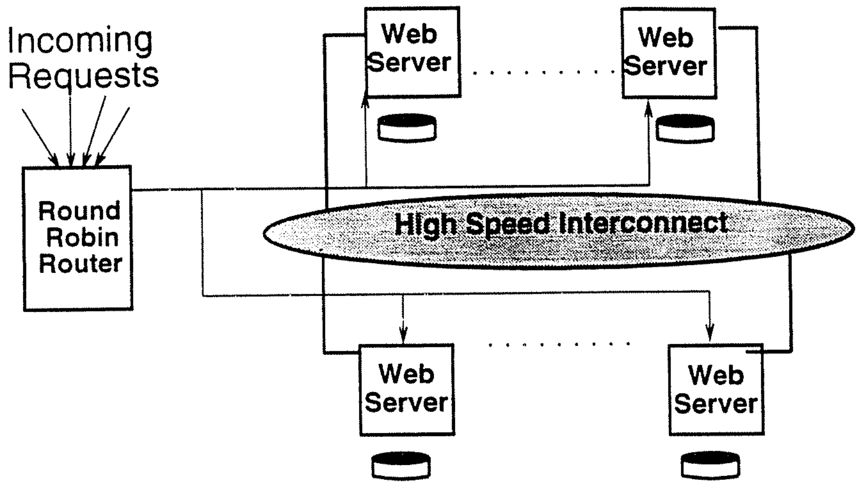

Figure 45: Web Server Architecture

5.2.1 Replacement Algorithms

All replacement algorithms described here maintain a FIFO list for hated pages. If a hated page is available, it is chosen for replacement. This detail is omitted from the description of the page replacement algorithms.

1. **Client-Server:ClSv**

   This is the direct application of the "client-server" algorithm that we use as a baseline for comparing other algorithms. Single and duplicate pages are maintained in one LRU list. When a page is needed, an LRU page is chosen for replacement. ClSv makes no distinction between single and duplicate pages, it uses LRU to choose a page that is least likely to be accessed in the future, completely ignoring the cost differential of accessing a single page from disk and a duplicate page from remote memory. The good thing about this algorithm is that, it is simple to implement, it increases the number of local hits as each server caches the hot pages in its local memory, and it requires minimal interaction among the servers. The main problem with this policy is that global memory could be severely underutilized due to duplication.

2. **Duplicate Elimination:Dup-Elim**

   Dup-Elim takes the other extreme approach. It only considers the cost differential between replacing a duplicate page and a single page. Since it is inexpensive to fetch a copy of the duplicate page from remote memory, compared to a disk I/O, all duplicate pages are eliminated before single pages. Dup-Elim tries to retain a large percentage of the database in global memory by eliminating duplicate pages first. Each server maintains two LRU lists, one for duplicate pages and the other for single pages. In the absence of hated pages the LRU duplicate page is replaced. If there are no duplicate pages in memory the LRU single page is replaced.

   The advantage of Dup-Elim is that it increases the percentage of the database in memory. However, a main drawback of Dup-Elim is that all duplicate pages are replaced before single pages, therefore hot pages that are duplicates may get replaced in place of cold single pages. This could result in frequent accesses to remote memory. This could prove to be costly and hurt performance.

In order to implement Dup-Elim each server has to know the list of pages that are duplicates in memory. A low message overhead algorithm to maintain this information is presented in Section 4.1.3. We implement the same algorithm in the simulator.

3. **Hybrid**

The page replacement algorithm in a single server system attempts to replace a page that is least likely to be accessed in the future. Since it is impossible to predict future accesses, heuristics like LRU are used to try to attain the goal. Reaching this goal in a single server system results in the best replacement policy, but, attaining this goal does not give us the best replacement policy in a multi-server system that has more than two levels in the memory hierarchy. In a single server, all pages replaced in memory incur a disk I/O when they are reaccessed, but, in a cluster of servers, the memory at each server has two kinds of pages: 1) duplicate pages when replaced are inexpensive to reaccess from remote memory and 2) single pages when replaced must be fetched from disk when they are reaccessed.

In order to minimize the performance effect of replacing a page, we use the following heuristic to identify the page for replacement. We estimate the performance impact of replacing a page in memory on the next page reference, and choose the page that has the lowest such impact. We use a heuristic algorithm that considers both the likelihood of the page being accessed next and the cost of reaccessing the page if it is discarded. In order to do this, we compute a metric called the "expected cost" of the next page reference if the page is discarded. We define the expected cost as the product of the latency to fetch page p ($C(p)$) back into memory and a weighting factor ($W(p)$) that gives a measure of the likelihood of the page being accessed next in memory. The page replaced from memory is the one with the lowest expected access cost $E(p)$.

$E(p)$ = Expected cost of accessing page p

$W(p)$ = A weight based on the likelyhood of the page being accessed

$C(p)$ = Cost to re-access page p

$E(p) = W(p) \times C(p)$

The following heuristic is used to compute the weight factor of each page in memory. We use the LRU order as an indicator of the likelyhood of accessing a page p next and set $W(p)$ to be proportional to the inverse of the elapsed time since the time of last access to page p ($\frac{1}{T(p)}$). We refer to the inverse of the elapsed time since last access as the temperature of the page p. We also make the assumption that the duplicate page when replaced remains in global memory when reaccessed and a single page when replaced has to be fetched from disk when it is reaccessed. This is a reasonable assumption, because, on the one hand a copy of the duplicate page will continue to remain in global memory even if it becomes a single page, because a single page that is hot is replaced only in LRU order. On the other hand, a single page has to be fetched from disk the next time it is reaccessed.

$T(p)$ = Elapsed time since last access to page p

$\frac{1}{T(p)}$ = Temperature of page p

$W(p) \propto \frac{1}{T(p)}$

The Hybrid page replacement policy works as follows: Two LRU lists, one for duplicate pages and the other for single pages are maintained. The LRU list for duplicate pages is maintained using a low message overhead algorithm described in Section 2. At the time of page replacement, only two pages have to be considered, they are the LRU duplicate page ($p_{duplicate}$) and the LRU single page (p_{single}). The ratio of the expected cost for discarding these two pages is computed. If this ratio is less than or equal to 1, then the expected cost of $p_{duplicate}$ is lower than that of p_{single} and therefore the duplicate page is replaced, otherwise p_{single} is replaced.

$$\frac{E(p_{duplicate})}{E(p_{single})} \begin{array}{ll} \leq 1 & \text{replace } p_{duplicate} \\ > 1 & \text{replace } p_{single} \end{array}$$

$$\frac{E(p_{duplicate})}{E(p_{single})} = \frac{\frac{1}{T(p_{duplicate})} \times C(p_{duplicate})}{\frac{1}{T(p_{single})} \times C(p_{single})}$$

The elapsed time since last access $T(p)$ can be easily calculated if a time stamp is maintained along with the buffer pool entry for the page. Maintaining the time stamp is inexpensive as the only overhead associated with it is to update the time stamp every time the page is accessed and this is an inexpensive operation. The average cost to access a page from remote memory ($C(p_{duplicate})$) and disk

$(C(p_{single}))$ can be easily maintained at run time by measuring the average time it takes to do a disk I/O and average time to access a page from remote memory for fixed time intervals.

5.3 Simulation Model

The simulator is written using the CSIM/C++ [Sch90] process oriented simulation package. The hardware platform simulated is a cluster of workstations with a server on each node. The number of physical nodes is controlled by the parameter N. For all experiments, except for the scale up, the number of nodes is fixed at 8. Each processing node has: a 40 MIP CPU, two disks, and 8 MB of memory. The page size is fixed at 8 KB. We use scaled down memory and database size to make simulation times feasible. Further, it is the relative size of the database to memory that is important in the performance measurements. The physical parameter are illustrated 14.

The disk modeled is a Fujitsu Model M2266 [FA90] (1 GB, 5.25") disk drive. The node and disk parameters are shown in Table 6. The interconnect modeled is a network with point to point bidirectional FIFO links with a link bandwidth of 15 MB/sec. Each node has a finite number of message buffers to send and receive messages. Each message incurs a fixed protocol overhead, a per byte instructional overhead, and a network delay. The message latency to send a 8 KB page from one node to another is set to 1.0 ms. The message latency is a parameter that we varied in the simulations.

Parameter	**Value**
Cpu MIPS	40
Service Discipline	Proc Sharing
Page Size	8 KB
Request Size	One File
Link Bandwidth	15 MB/sec
Message Cost	1.0 ms/8 KB
Message Buffers	160 KB/Node
Memory Size/Node	8 MB

Table 14: Physical Parameters

Table 7 also summarizes the CPU costs and execution parameters used in the simulator. The simulator uses CPU instructions for various segments of the buffer management code, message overheads, disk I/Os, table lookups, locking/unlocking, and for processing pages. In all cases, the simulations were run for a period of time specified by *Sim Time*. To avoid transient startup effects, each run included an initial startup time specified by *Cold Time* to warm up the buffer pool before collecting statistics. To ensure the statistical validity of our results, we verify that the confidence intervals for the response times were well within 2 percent of the mean in all cases.

5.3.1 Workload

To simulate workloads experienced by web servers, we closely study the logs maintained at several web sites. We find that the access frequencies to files at these web sites to closely resemble a Zipfian distribution [Zip49]. If the number of accesses to the files T, the number of files is F, and the skew is z. The frequency for accessing a file "i" generated by a zipfian distribution is given by the following formula:

$$t_i = T\frac{1/i^z}{\sum_{i=1}^{F} 1/i^z} \text{ for all } 1 \leq i \geq F$$

We demonstrate this with the logs maintained for the web server at NCSA for Oct 11, 1995 (`http://www.ncsa.uiuc.edu`) and the local web server for the CS Department web server at University of Wisconsin (UW-CS) (`http://www.cs.wisc.edu`). We sort the files by descending order of the access frequencies and normalize the access frequencies with the total number of accesses. We plot the normalized access frequencies to the top 50 files along with a zipfian distribution of a skew that is close to the observed curve. Figure 46 shows the plot for the files at NCSA and a curve for the zipfian distribution of skew 0.55. Notice how closely the zipfian curve matches the observed access frequencies. Figure 47 shows the plot for the files at UW-CS along with a zipfian distribution with a skew of 0.6. Once again the zipfian curve closely matches the observed access frequency. An hour to hour log of the access frequencies also shows similar behavior. Study of the logs for other days and at other sites reveal a similar co-relation between the observed access frequencies and the zipfian distribution.

Since the observed access frequencies closely matches the zipfian distribution, we use the zipfian distribution to control access frequencies to the files in the simulations. We control the skew of the zipfian distribution with the zipf parameter. The zipf parameter help us to perform a controlled study of the effect of duplication on performance. The zipf parameter is varied from 0 to 2 to cover a wide range of skews. When zipf is 0,

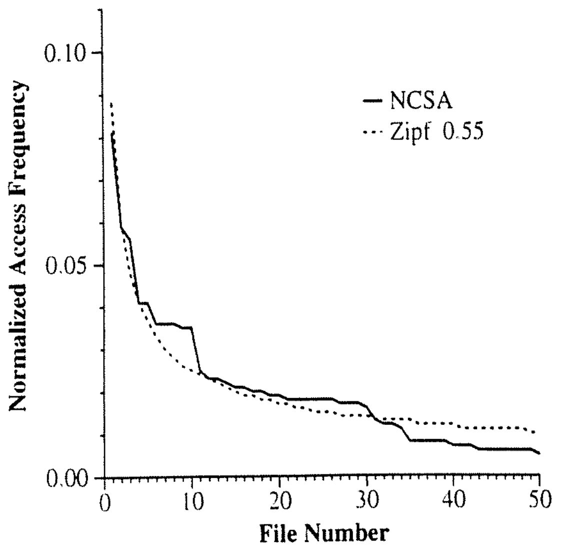

Figure 46: NCSA vs Zipf 0.55

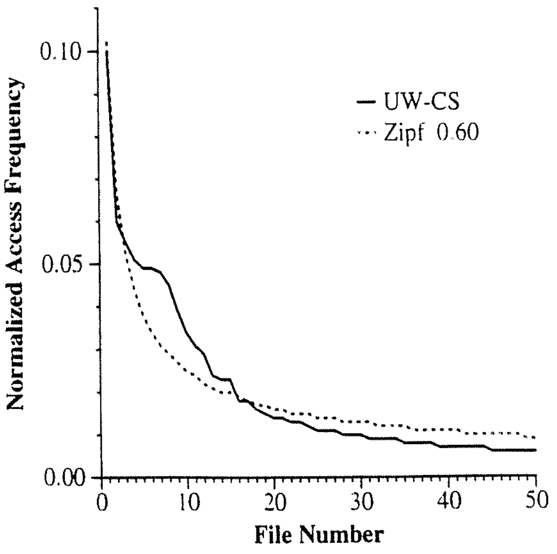

Figure 47: UW-CS vs Zipf 0.60

the access frequencies to files are uniform, and when the zipf parameter is 2 the access frequencies to files are highly skewed, i.e a few files are very hot and the rest are cold. The zipf parameter is also referred to as the skew.

Typical database sizes at web sites are large, of the order of several gigabytes. A close study of the web traces reveals that a large number of requests are directed to a small portion of the database. For example: the web site as NCSA has a database that is several gigabytes, but the portion of the database accessed during the course of an entire day is only about 265 MB. Therefore, for the purpose of this study we only consider the active portion of the database, the accesses to which are significant, since the outcome of the experiments are not dependent on the accesses to the less frequently accessed portion of the database. Moreover, our intent is to capture the portion of the database that is of interest to the clients rather than to model the entire database.

We simulate a database system consisting of files of the same fixed size. The number of files and the size of the files are parameters to the experiments. For most experiments, the size of each file is fixed at 32 pages and number of files at 300. We use fixed file sizes because experiments with different file sizes and randomly distributed file sizes described in Section 5.4.4 shows that the algorithms are not dependent on the file sizes.

The aggregate file size (active database size) is 9,600 pages (78.7 MB). This is 1.2 times the size of global memory for 8 nodes. We reiterate that we view this as the size of the active portion of the database and not the entire database. To study the sensitivity of the results to the active database size, we also perform experiments for a larger active database size that is 2 times the size of global memory and for a smaller active database size that is 0.8 times the size of global memory. We did not simulate active database sizes of say 5 times the size of global memory, because, if a large portion of the accesses are directed to such a database then the system is grossly under-configured in terms of memory; only 20% of the hot active portion of the database fits in memory in the best case. In this case, it would not matter how one manages global memory as the system would in any case be completely disk bound.

The pages of the files are round-robin declustered among all the servers. The study in Venkataraman et al. [VLN95] shows that declustering is better than clustering for good global memory utilization in a cluster of servers. If the concern is about a single point of failure, then chained declustering described by Hsiao and DeWitt [HD90], or Redundant Array of Inexpensive Disks (RAID) [PCGK89, CLVW94], or shared SCSI chains can be used; Chained declustering allows the servers to continue running in the event of a node

Parameter	Value
Number of Nodes	8
Aggregate memory	64 MB
Message Cost	1 ms
Number of Files	300
File Size	32 pages/file
Aggregate File Size	76.8 MB
Zipfian Skew	0 to 2
MPL	4

Table 15: Workload Parameters

failure and it distributes the disk load evenly among remaining servers. Data placement is a factor only if there is idle global memory available to be utilized. Since all the servers are heavily loaded, data placement does not affect the use of memory. Also, the algorithms to control replication are independent of the data placement strategy used.

A round-robin policy is used to distribute the clients among the servers. The maximum number of requests at a server is controlled by the Multi Programming Level (MPL). For all experiments, the MPL level at each server is fixed at 4. A MPL of 4 would mean that the total number of clients in the system is 32 at any given time. We operate the system at the maximum MPL to study the system when it is heavily loaded, as we expected this to be the case with web servers. We confine ourselves to studying read-only workloads since the workloads experienced by web servers are primarily read-only. The workload parameters varied in the experiments are shown in the Table 15.

5.4 Performance Evaluation

RoadMap

We use response time as the primary performance measure to evaluate the algorithms. In the first experiment, before evaluating the memory management algorithms, we demonstrate the effect of duplication on response time by artificially controlling the number of duplicate pages in global memory. Following this, we thoroughly evaluate all the three algorithms and study how I/Os, remote memory accesses, and message overheads contribute toward the response time by varying the zipfian skew. The ratio of the I/O and message latency determines the relative importance of a single and a duplicate page. We evaluate the impact of this ratio by varying the message latency in the third experiment. In the fourth experiment, we examine the effect of the ratio of the database size to global memory size, and we also study the effect of different file sizes. In the next experiment we demonstrate the scalability of the Hybrid algorithm by varying the number of nodes. In the last experiment, we demonstrate the robustness of the Hybrid algorithm for non zipfian workloads.

5.4.1 Duplication vs. Response Time

Before proceeding to evaluate the algorithms, we perform a preliminary experiment to study the effect of duplication on response time. To do this, we use a replacement algorithm to artificially control the maximum number of duplicate pages in memory at each server. We are not proposing this as a new replacement algorithm, but, we use this algorithm to perform a controlled experiment to study the effects of duplication on response time. A threshold parameter is used by the algorithm to control the number of duplicate pages, and it works as follows:

Each server maintains two LRU lists: one for duplicate pages and the other for single pages. The replacement algorithm replaces the duplicate page if the fraction of the duplicate pages in memory exceeds the threshold; otherwise, it replaces the LRU page among all the pages (including duplicates). A threshold of 0 corresponds to Dup-Elim as all duplicate pages are eliminated before single pages and a threshold of 1 corresponds to ClSv, as all pages are replaced in LRU order. One can view the Hybrid algorithm as the one trying to dynamically find the optimum point on this curve.

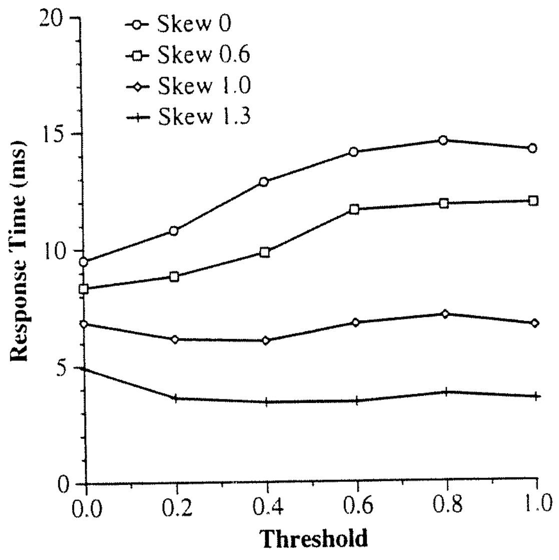

Figure 48: Threshold, Msg Lat=1.0 ms

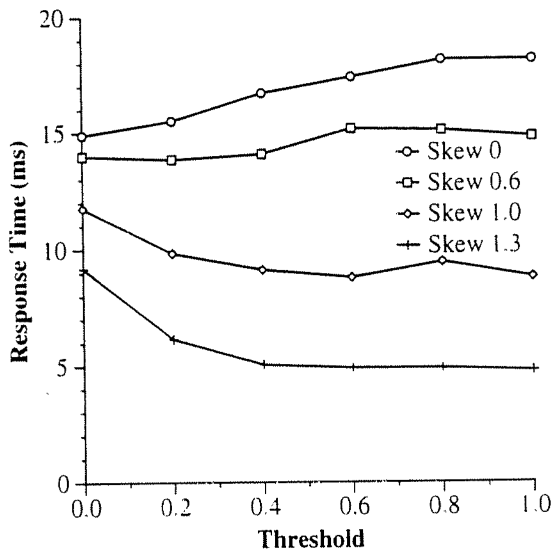

Figure 49: Threshold. Msg Lat=2.0 ms

For this experiment, the number of files is fixed at 300 and the size of each file is fixed at 32 pages. As a consequence of which the aggregate database size is 9,600 pages, and this is 1.2 times the size of aggregate memory. The response time is plotted as a function of threshold.

Figure 48 shows the response time as a function of the threshold when the message latency is 1 ms. In this figure, at low skews (0 and 0.6), the response time increases significantly as the threshold is increased. This is because, at low skew, all pages are accessed with uniform probability and caching more pages in global memory reduces I/O and therefore lowers response time. Increasing the threshold increases the number of duplicate pages in memory thereby increasing the number of disk I/Os, as a consequence of which the response time increases sharply with threshold.

For high skews (1.0 and 1.3) the response time is a minimum when the threshold is around 0.4. This is because, at high skew, some files are hot and others are cold, so duplicating and retaining the hot pages locally decreases the number of remote memory accesses but does not increase the number of disk I/O as the hot pages are accessed often enough to be duplicated in local memory at each server. Beyond a certain threshold the response time increases slightly because all hot pages are duplicated and cached in local memory and increasing the threshold does not save remote memory accesses but increases disk I/O, as pages of files that are cold also get duplicated.

Figure 49 shows the response time as a function of the threshold when the message latency is 2.0 ms. The response time for skews of 0 and 0.6 increases by about 15-20% when the threshold increases. This increase is not as significant when compared to the increase when the message latency is 1.0 ms because the higher message cost value lowers the benefits produced by fewer disk I/Os. When the skew is 1.3, the response time drops by 50% as the threshold increases from 0 to 0.6 as increasing the number of duplicate pages in memory reduces the number of remote memory accesses to frequently accessed files. Since messages are expensive, this leads to a significant decrease in the response time.

Neither a threshold of 0 (ClSv) nor a threshold of 1.0 (Dup-Elim) provides the best response time for all skew levels. This experiment demonstrates that neither works well in all cases. The aim of the Hybrid policy is to find the optimum point on this curve by dynamically controlling the number of duplicate pages in memory.

5.4.2 Vary Zipfian Skew

In this experiment we study the performance of the ClSv, Dup-Elim, and Hybrid algorithms, by varying the skew of the zipfian distribution to control the the access frequency. The number of files and the file sizes are fixed at 300 and 32 pages/file respectively. and the latency for data messages is fixed at 1.0 ms.

Figure 50 shows the average number of disk I/Os per request as a function of skew. Dup-Elim, by getting rid of duplicates quickly retains a greater portion of the database in global memory and therefore has the lowest I/O cost. ClSv has the highest I/O cost as it allows pages to be duplicated in memory among the servers. Hybrid, has an intermediate I/O cost, that leans towards Dup-Elim at low skew because the expected cost for discarding a duplicate page is much higher when accesses are uniform. The I/O performance can be better understood by looking at Figure 51 which shows the percentage

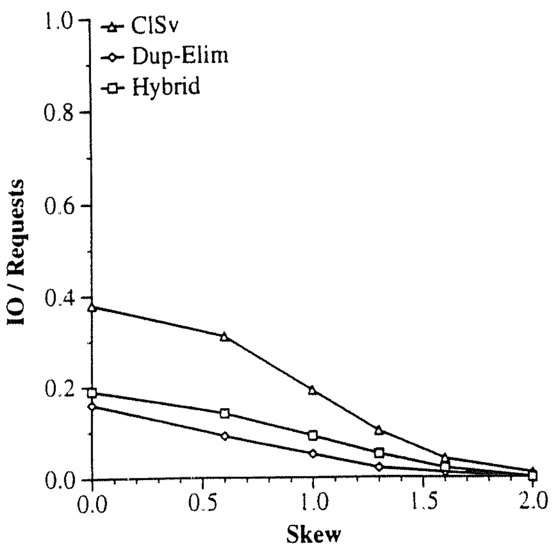

Figure 50: Skew : I/O

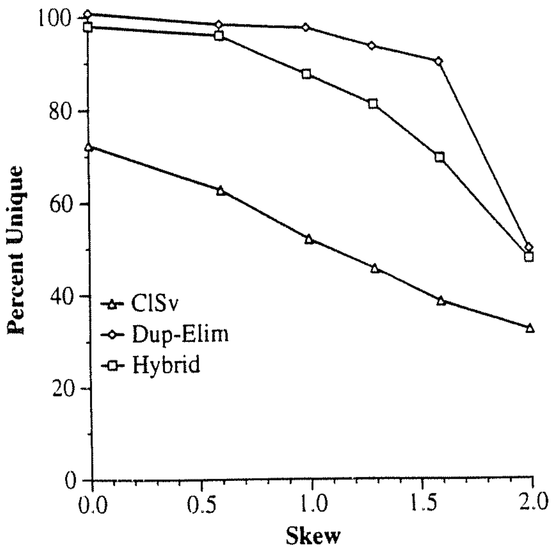

Figure 51: Skew : % Unique Pages

of global memory occupied by unique pages. Dup-Elim has the highest percentage of the database in global memory, followed by Hybrid and ClSv. Note that the number of unique pages in global memory for Dup-Elim is slightly less than 100% even though it quickly eliminates all duplicate pages. This is because, some duplicate pages are very hot compared to others and they remain at the head of the LRU list while other duplicate pages are being replaced. The number of unique pages decreases as the skew increases because the portion of the database to which most accesses are directed decreases. At high skews between 1.6 and 2.0 the number of unique pages for Dup-Elim is close to 50%, since a very small portion of the database is active. The number of unique pages for Hybrid is in-between the two algorithms, as it permits the duplication of a limited number of hot pages.

The I/O gap between Dup-Elim and ClSv is the highest, when the skew is low and the gap is close to 0 for skews of 1.6 and 2.0. At low skews, keeping a large portion of the database in memory is important as all files are accessed with uniform probability. At skews of 1.3 and above only few files are hot and ClSv does a good job of duplicating and retaining the pages of the hot files in local memory at each server.

Figure 52 shows the average number of data messages per request. Message overhead for all the policies is about the same when the skew is 0 as all pages are accessed with the same probability. On a miss, message is needed to either access the page from remote disk in the case of ClSv or to fetch it from remote memory in the case of Dup-Elim. For skews greater than 0 the message overhead for Dup-Elim is 60%-100% higher than that of ClSv, since Dup-Elim in its eagerness to eliminate duplicates ends up accessing a lot of hot pages from remote memory. Hybrid has an intermediate message overhead since it only permits the hot pages to be duplicated, but does not duplicate as many pages as ClSv. The pages duplicated by Hybrid are those that have a higher expected access cost than the LRU single page.

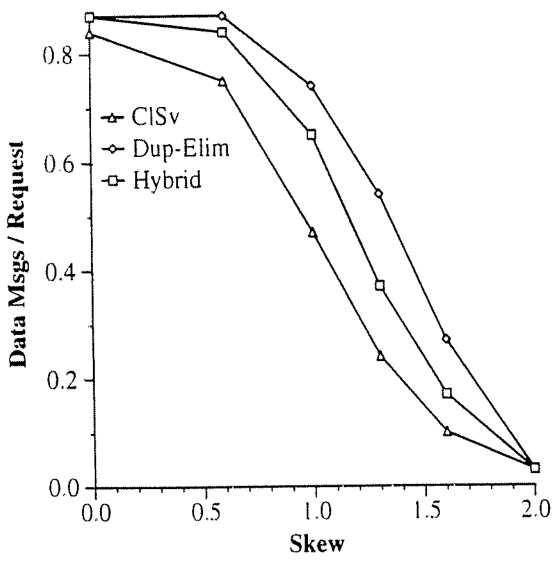

Figure 52: Skew : Num Data Msgs

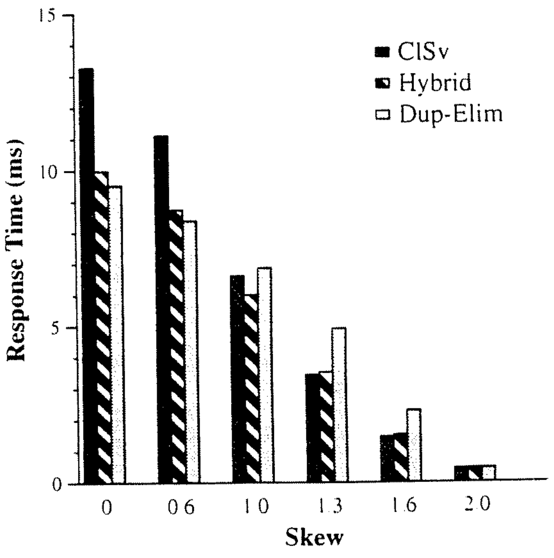

Figure 53: Skew : Response Time

Figure 53 shows the response time as a function of skew. At skews of 0 and 0.6, ClSv does 30% worse than Dup-Elim due to the poor global memory utilization. At skews greater than 1.0, ClSv does 40% better than Dup-Elim as the I/O cost difference between the two algorithms is negligible and the message costs for Dup-Elim is much higher than that for ClSv. Hybrid has a response time that is close to the minimum at both low and high skews as it strikes a good balance between duplicating hot pages and discarding duplicate pages. At low skew Hybrid eliminates duplicates like Dup-Elim and at high skew it duplicates the pages that are very hot like ClSv.

5.4.3 Message Latency

The ratio of the message to I/O latency determines the relative importance of a duplicate page and a single page. In this experiment we study the algorithms for message latencies of 0.5 ms and 2.0 ms. The number of files is fixed at 300 and the size of each file is fixed at 32 and the response time is measured as a function of the skew.

Figure 54 shows the response time per request when the message latency is 0.5 ms. With message costs of 0.5 ms, the cost of accessing data from remote memory is very cheap. Dup-Elim does very well and the response time is half that of ClSv for skews less than 1.0. For skews between 1.0 and 1.3 the response times of Dup-Elim and ClSv are comparable even though Dup-Elim incurs a higher message overhead than ClSv at high skew. The small I/O gain is enough to offset message overhead due to the lower message latency. The response time for Hybrid is close to the minimum response time for all

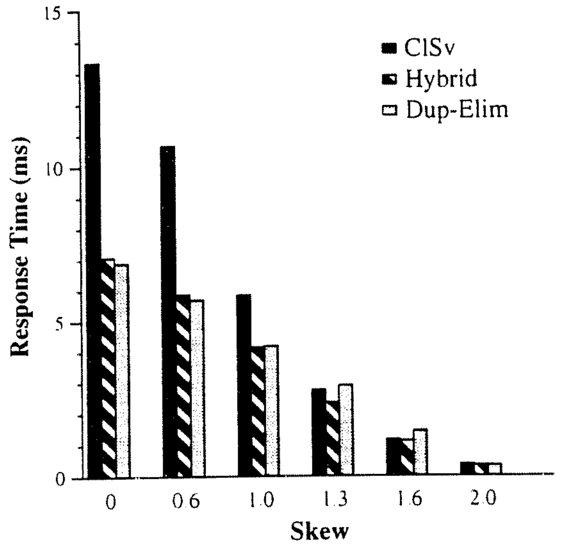

Figure 54: Msg Latency 0.5 ms

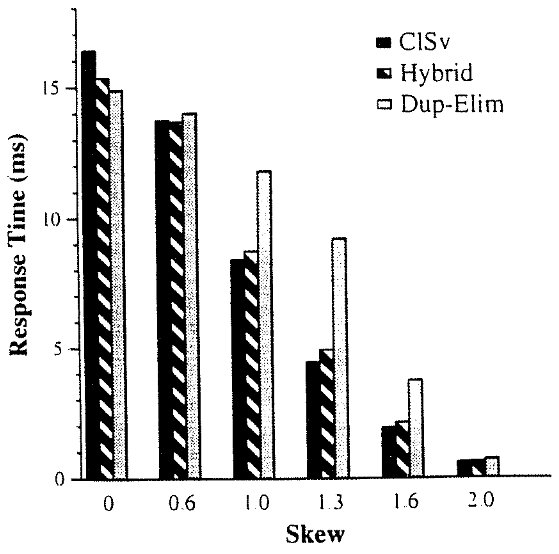

Figure 55: Msg Latency 2.0 ms

skews. Its response time is close to that of Dup-Elim at low skew and almost equal to ClSv at high skew. At intermediate skew it has a response time that is lower than both Dup-Elim and ClSv.

Figure 55 illustrates the response time when the message latency is 2.0 ms. At low skews the response times of ClSv and Dup-Elim are comparable, even though Dup-Elim has half the I/O cost of ClSv. The reason for this is the higher message latencies and the higher number of remote memory accesses by Dup-Elim over ClSv negates Dup-Elim's I/O gains. At higher skews, (for example, a skew of 1.3), the response time of ClSv is half that of Dup-Elim because Dup-Elim has a higher message cost than ClSv while the I/O costs for the two policies are comparable. The message latency of 2.0 ms widens the gap between the response time of Dup-Elim and ClSv. Hybrid once again has a response time that is close to the minimum, which in this case is equal to that of ClSv at high skew and Dup-Elim at low skew. The response time of Hybrid is marginally higher.

about 2% higher than that of ClSv at high skew. This is due to the message overhead necessary to maintain duplicate information by Hybrid.

5.4.4 Active Database Size

So far we have considered an active database size of 1.2 times the size of global memory. In this experiment we evaluate the algorithms for two cases: one where the active database size is twice the size of global memory and the other where it is 0.8 times the size of global memory.

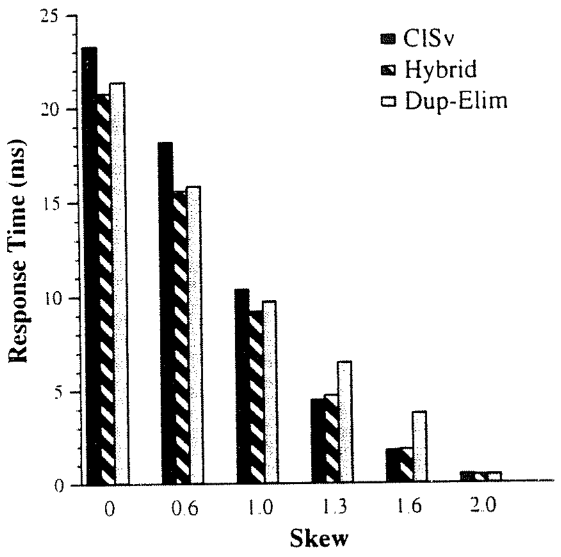

Figure 56: DB Size =2.0 * Total Mem

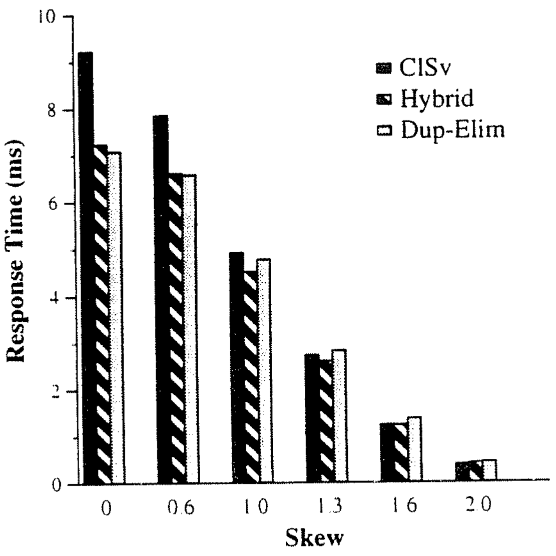

Figure 57: DB Size =0.8 * Total Mem

Figure 56 shows the response time when the database size is twice the size of global memory. The number of files is 600, and the size of each file was fixed at 32 pages. The response time gains of Dup-Elim over ClSv at low skew is very small, only about 10%. This is because, the amount of memory that Dup-Elim frees by eliminating duplicates is a much smaller fraction of the database size. For skews above 1.0, the response time of Dup-Elim is 40% worse than ClSv because the I/O costs for the two algorithms are comparable, and the number of data messages for Dup-Elim is much higher than that for ClSv.

Figure 57 shows the response time when the database size is 0.8 times the global memory size. The number of files is 200, and the size of each files is 32 pages. Dup-Elim has a 40% better response time than ClSv at low skew because Dup-Elim is able to fit the entire database in global memory and incurs no I/O costs while ClSv fits only a portion of the database in global memory due to duplication and incurs I/O costs as a consequence. Fitting a larger percentage of the database in memory improves performance significantly because all files are accessed with uniform probability. At high skew, the performance of all algorithms are comparable because Dup-Elim, like ClSv, is able to duplicate all the hot pages in local memory at each server. Hybrid allows pages to be duplicated as there are very few pages that are hot.

For both database sizes Hybrid has a response time that is close to the minimum at both low and high skews. At low skew the response time of Hybrid is close to that of Dup-Elim and at high skew response time of Hybrid is close to ClSv.

File Size

We perform this experiment to study the effect of file size on response time. The database size was fixed at 9600 pages. We used file sizes of 16, 32 and 64 pages and found that the response times to be about the same for all three file sizes. We do not show the graph for these experiments as they look identical to the one in Figure 53.

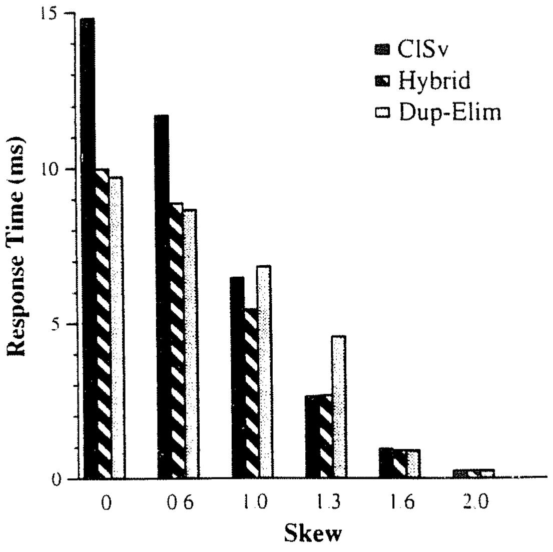

Figure 58: File Size

Another experiment that we performed was to have multiple file sizes in the database. We used file sizes of 16, 32, 64 and 128, but fixed the size of the database at 9600 pages. The pages were divided equally among the different file sizes. There were 150 files with a file size of 16, 75 files with a file size of 32 and so on. To relate the files with the zipfian distribution we numbered the files in the decreasing order of frequency and generated the file numbers using a random number generator. We ran our simulations several times with different seeds for the random number generator. We found the response time in each case to be well within the confidence interval. Figure 58 shows the average response time as a function of skew. Each point in the graph is an average over several runs of the simulator. The response time for this experiment closely matches the response time shown in Figure 53, where the file sizes are uniform and fixed at 32 pages/file. This demonstrates that the file size has no significant effect on performance.

5.4.5 Scale-Up

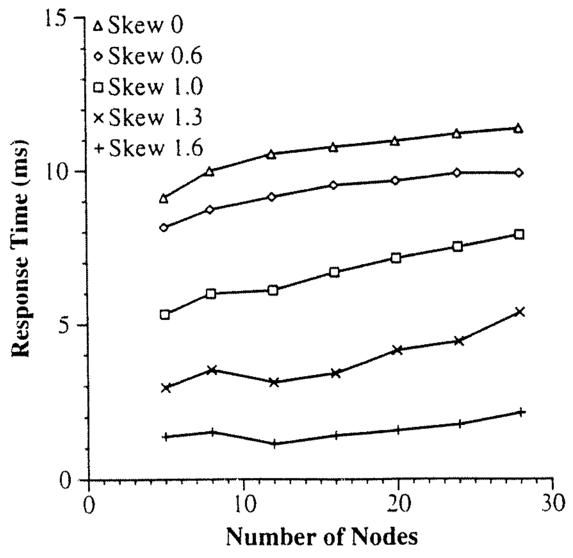

Figure 59: Scale-Up

We studied the scale-up characteristic of the Hybrid algorithm by varying the number of nodes. An algorithm with a good scale-up characteristic should have a flat response time curve when the database and workload are scaled with the number of nodes. A flat scale-up curve is an indication that the users do not see degraded response times for their request as the number of nodes in the system goes up. The number of nodes was varied from 5 to 28. In each case the database size was fixed at 1.2 times the size of global memory. Briefly, as the number of nodes is increased, the size of the database is also increased so that it is always 1.2 times the size of global memory. The number of clients in the system was also increased proportionately so that the MPL at each node remained at 4. The message latency for a data message was fixed at 1.0 ms.

Figure 59 shows the response time as a function of the number of nodes for the Hybrid algorithm for workloads with skews ranging from 0 to 1.6. The response time for Hybrid is relatively flat for all skew levels indicating a good scale-up characteristic of Hybrid. However, there is a slight increase in response time when the number of nodes is increased from 5 to 28. With 5 nodes, 20% of the accesses are for the data on the local disk and the rest are to data on remote disks as the files are declustered among the servers. With 28 nodes, only about 3.5% of the accesses are from the local disk. and the rest are from remote disks, so the response time increases as the result of the larger percentage of remote accesses.

We also checked the response time of Hybrid with that of the other two algorithms and found that Hybrid has the best response time for all the node configurations and skews.

5.4.6 Non Zipfian Workload

So far we have confined ourselves to examining access patterns that are zipfian. In this experiment we evaluate the algorithms for a non zipfian workload typical of those for object database systems. The workload has a small hot data set and a large set of cold files. The hot set has 5 files, the cold set has 295 files, and the size of each file as before is fixed at 32. A client request consists of an access to a random file in the hot set followed by an access to a random file in the cold set. The clients are round-robined among the servers.

Figure 60 is a histogram that shows the response time per request for three message latencies. At low message latencies, saving I/O is more important than avoiding network

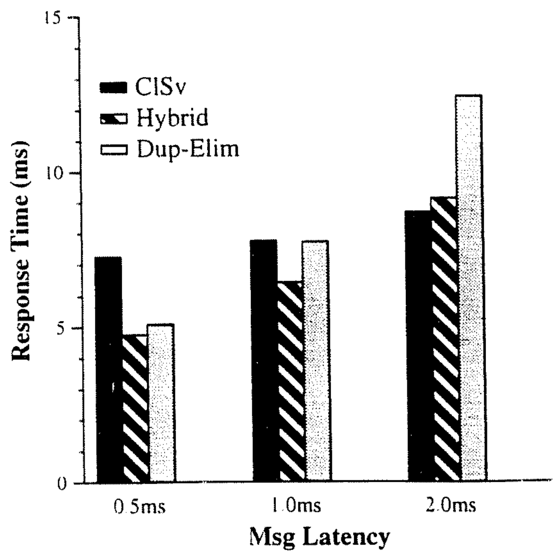

Figure 60: Non Zipfian Workload

overhead; therefore, ClSv does 40% worse than Dup-Elim as it eliminates duplicates to save on I/O. When the message latency is 2.0 ms, the high message costs incurred by Dup-Elim to access pages from remote memory is evident and Dup-Elim does 50% worse than ClSv. At 1.0 ms the response time for both ClSv and Dup-Elim are comparable, but Hybrid does the best and has a response time that is 20% lower. Hybrid has the best response time for all messages latencies. This experiment is a simple demonstration of the applicability of the Hybrid algorithm to other workloads and further evaluation and experimentation for non zipfian workloads is necessary.

5.4.7 Summary of Experimental Results

Our experimental results show that, in a cluster of servers, a good balance must be struck between remote memory accesses and disk I/Os, in order to efficiently manage global memory. Neither an algorithm that does nothing about duplicates (ClSv) and incurs a lot of disk I/Os due to poor global memory utilization, nor the one that tries to eliminate all duplicates (Dup-Elim) and incurs message costs to access data from remote memory has the best performance. Hybrid that uses a heuristic to replace the pages with the lowest expected access cost. It strikes a good balance between network overhead and I/O cost and has a response time that is close to the minimum for a wide range of workloads. Our experiments with workloads with a wide range of skews indicate that Dup-Elim is better than ClSv when the access probabilities to the files are nearly uniform and ClSv is better than Dup-Elim when the accesses to files are highly skewed. Only the Hybrid algorithm does well at low and high skews as it replaces the pages based on their temperature and the cost of accessing them. Evaluation of the Hybrid algorithm for a range of parameters involving message latencies, database sizes, and file sizes shows that it does well under a variety of conditions. Scale-up experiments demonstrates that Hybrid scales up very well to large node configurations. The experiment with a non zipfian workload demonstrates in a small way that the Hybrid algorithm works well for non-zipfian workloads.

5.5 Conclusions

Clusters of servers have become necessary in light of the growing popularity of the WWW. Direct application of client-server memory management algorithm like ClSv reduces effective global memory utilization in the cluster due to duplication of data in memory and, as a result of this incurs high I/O cost and low network overhead. At the other extreme, an algorithm that attempts to eliminate all duplicates, Dup-Elim, has low I/O cost and high network overhead. While Dup-Elim has the best response time in some cases and ClSv in others, neither of these two approaches perform well for the range of workloads we studied. We have shown that the Hybrid algorithm correctly balances the I/O and the network costs and performs well for a wide range of workloads and node configurations.

Chapter 6

Conclusions

6.1 Thesis Summary

In a client server database system the single server is both a cpu and an I/O bottleneck. An obvious solution that is increasingly becoming popular is to replace the bottleneck with multiple servers connected by a high speed network. A driving force behind this trend is the availability of low cost, high speed networks, that enables a collection of database servers to function as a single database "super-server". With present day network technology it is much faster to access data from remote memory than to read data from the local disk. This trend is not likely to change in the near future. The availability of low cost, high speed networks, and large aggregate memory makes it important to utilize global memory efficiently for high performance. We identify two important factors, 1)utilizing global idle memory, and 2) managing duplicates, to minimize response time in Chapter 1. We design and study algorithms that address these two issues. The algorithms are designed so that they are simple to implement but at the same time efficiently utilize aggregate memory.

On the issue of utilizing idle memory in the cluster, we demonstrate the significant impact of data placement, namely, clustering and declustering, on how global memory is utilized. We design three page replacement algorithms, ClSv, Reserve, and Global, that are simple modifications to the client-server buffer management code. We evaluate the performance of the memory management policies that are combinations of the data placement, and page replacement algorithms through a simulation study using CSIM, and an implementation on the IBM SP/2. Using synthetic workloads characteristic of those experienced by object databases we demonstrate that when the data is declustered, simple memory management policies are sufficient to utilize the aggregate memory capacity of the server cluster. We show that Global when combined with declustering gives the best performance.

To control duplication of data in memory we present a new algorithm, Hybrid, that dynamically controls the amount of duplication. We show that on workloads characteristic of those experienced by Web servers, the Hybrid algorithm correctly trades off intracluster network traffic and disk I/O to minimize the average response time.

6.2 Future Work

There are many technical hurdles to be overcome in the transition from a single server to a multi-server database system. This thesis addresses one of these issues; distributed memory management. In this section, we discuss future work in the area of distributed memory management, and other areas of research in multi-server database systems.

6.2.1 Refinements to Memory Mangement

In this section we describe some of the refinements that may be incorporated into the policies to improve performance.

- **Integrated Algorithm:** Global-Decl is identified as the best algorithm for utilizing idle memory and Hybrid controls duplicates and has the best response time. Since both these problems can exist simultaneously, combining these two policies is necessary to manage memory.

  Global-Decl discards a duplicate page first and then discards single pages owned by servers that have duplicate pages, followed by single pages that are owned by servers that have the global LRU page. It is shown in Chapter 5 that discarding duplicate pages first hurts performance when duplicate pages are hot. Hybrid solves this problem by discarding the page with the lowest expected cost, be it a single or a duplicate page. The Global-Decl policy can be modified to consider the expected cost instead of the LRU time on the page. Implementing this algorithm is fairly straightforward. Additional work has to be done to evaluate this algorithm.

- **Non Homogeneous Memory:** In the experiments the size of memory at each server is identical. If the memory size at the servers is not the same, uniformly declustering the data across all the servers would not make good use of memory at all the servers. The memory at the server with more memory would be poorly utilized. One solution to solve this problem would be to decluster data in the locality set in proportion to the size of the memory at each server. As a result, a proportionate portion of the accesses would be directed towards that server with more memory resulting in better use of memory at that server. We did not study this in the thesis.

- Declustering Skew: A problem related to the issue of non-homogeneous memory is that of declustering skew. If the data is declustered so that the data is skewed, then global memory would not be fully utilized.

6.2.2 Load balancing

Balancing the CPU load in a multi-server database system is very important for performance. It is also tied closely with how global memory is utilized. For CPU intensive workloads, the CPU usage may not be balanced among the servers even though memory use may be balanced across all the servers. Balancing the CPU load while utilizing global memory is an important area of future work.

6.2.3 Parallelism

Introducing parallelism into object database workloads, for example ParSets [DNSV94], distributes the use of memory among the servers. However, skew in the execution times at different servers could cause memory and CPU load imbalances. Studying the effect of parallelism on resource utilization, and balancing the use of CPU, and memory resources is very important for good performance in such a system.

6.2.4 High availability

Another important issue that must be addressed in a multi-server is that of resilience to failure. Since the multi-server database system consists of multiple nodes, the mean time to failure is fairly high; however, there is hardware redundancy, and fault isolation inherent in the system that could be used to keep the system running when there is a failure. Software must be designed so that there is a smooth transition from a normal state to a failure state when one or more nodes fail. Performance impact of this transition should be minimal and the load of the failed node(s) should be shared by the nodes that are active.

Multi-server database systems have become popular recently due to their scalability and low cost. Several research issues related to multi-servers are very poorly understood. In this thesis we have implemented and studied practical algorithms for managing memory in a multi-server database system. There is a lot of scope for future research in this area and other related areas some of which are highlighted in this chapter.

Bibliography

[AMM$^+$95] T. Agerwala, J. Martin, H. Mirza, D. Sadler, D. Dias, and M. Snir. SP2 System Architecture. *IBM Systems Journal*, 34(2), Jan 1995.

[BAC$^+$90] H. Boral, W. Alexander, W. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[BCF$^+$95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet – A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb 1995.

[Be89] B. Bret and etal. The GemStone Data Management System. In *Object Oriented Concpts, Databases and Applications*, Brisbane, Australia, 1989.

[BG88] D. Bitton and J. Gray. Disk Shadowing. In *Proc. of the 14th VLDB Conf.* Los Angeles, 1988.

[BLA$^+$94] A. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. Felten. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[CABK88] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. In *SIGMOD*, pages 99–108, Chicago, IL, June 1988. Database Machine, Parallelism, Data Placement.

[CBZ91] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th International Symposium on Operating System Principles*, Pacific Grove, CA, Oct 1991.

[CDF$^+$94] M. Carey, D. DeWitt, M. Franklin, N. Hall, McAuliffe, M., J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, J. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1994.

[CDG$^+$90] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vendenber. The EXODUS Extensible DBMS Project: An Overview. Readings in Object Oriented Databases, S. Zdonik and D. Maier, 1990. eds., Morgan-Kaufman.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proceedings of the ACM SIGMOD Internationational Conference on Management of Data*, pages 12–21, June 1993.

[CFLS91] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, June 1991.

[CFZ94] M. Carey, M. Franklin, and M. Zahariodakis. Fine Grained Sharing in a Page Server OODBMS. In *Proceedings of the ACM SIGMOD International-ational Conference on Management of Data*, June 1994.

[CG90] D. Comer and J. Griffoen. A New Design for Distributed Systems.: The Remote Memory Model. In *Proceedings of Summer USENIX Conference*, pages 127–135, June 1990.

[CLVW94] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. TickerTAIP: A Parallel RAID Array. *ACM Transactions on Computer Systems*, August 1994.

[Deu91] O. Deux. The O2 System. *Communications of the ACM*, 34(10), Oct 1991.

[DFMV90] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A Study of Three Alternative Workstation Server Architectures for Object Oriented Database SYstems. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Aug 1990.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[DNSV94] D. DeWitt, J. Naughton, J. Shafer, and S. Venkataraman. ParSets for Parallelizing OODBMS Traversals: A Performance Evaluation. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, Austin, Tx, Sept 1994.

[DWAP94] 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 Conference on Operating Systems Design and Implementation*, Nov 1994.

[DY91] A. Dan and P. Yu. Analytical Modelling of a Hierarchical Buffer for a Data Sharing Environment. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991.

[DY92] A. Dan and P. Yu. Performance Analysis of Coherency Control Policies through Lock Retention. In *Proceedings of the International Conference on Management of Data*, June 1992.

[DY93] A. Dan and P. Yu. Performance Analysis of Buffer Coherency Policies in a Multisystem Data Sharing Environment. *IEEE Transactions on Parallel and Distributed Systems*, 4(3), March 1993.

[FA90] Inc Fujistsu America. M2265 Techniccal Manual. Technical Report 41FH5048E-01, Fujistsu America Technical Assistance Center, SanJose, CA, 1990.

[fADF90] The Committee for Advanced DBMS Functions. Third Generation Database System Manifesto. *SIGMOD Record*, 19(3), Sept 1990.

[FBR93] S. Frank, H. Burkhardt, and J. Rothnie. The KSR1:Bridging the Gap Between Shared Memory and MPPs. In *Proceedings of the IEEE International Conference COMPCON*, pages 285–294, SanFrancisco, CA, February 1993.

[FC92] M. Franklin and M. Carey. Client-Server Caching Revisited. In *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, Aug 1992. Morgan Kaufmann.

[FCL92] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada, August 1992.

[FCL93] M. Franklin, M. Carey, and M. Livny. Local Disk Caching in Client-Server Database Systems. In *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[FMP$^+$95] M. Feeley, W. Morgan, F. Pighi, A. Karlin, H. Levy, and C. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[FZ91] E. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, University of Washington, March 1991.

[FZT+92] M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, San Diego, June 1992.

[Gha90] S. Ghandeharzadeh. *Physical Database Design in Multiprocessor Systems.* PhD thesis, Department of Computer Science, University of Wisconsin-Madison, 1990.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface . MIT Press, 1994.

[HD90] H. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machine. In *Proceedings of the 6th International Conference on Data Engineering*, Feb 1990.

[HF86] R. Hagmann and D. Ferrari. Performance Analysis of Several Back-End Database Architectures. *ACM Transactions on Database Systems*, 11(1), March 1986.

[HKS+88] J. Howard, M. Kazar, M. Sherri, D. Nichols, M. Satynarayana, Sidebothamand R., and J West. Sclae and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), Feb 1988.

[HZ87] M. Hornick and S. Zdonik. A Shared Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1), Jan 1987.

[ILP93] L. Iftode, K. Li, and K. Petersen. Memory Servers for Multicomputers. In *Proceedings of the IEEE Spring COMPCON '93*, pages 534–547, February 1993.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine Grained Mobility in the Emerald System. *ACM Transaction on Computer Systems*, 6(1), February 1988.

[KBM94] E. Katz, M. Butler, and R. McGrath. A Scalable HTTP Server: The NCSA Prototype. *Computer Networks and ISDN Systems*, 27:155–164, September 1994.

[KCWW92] S. Khoshafian, A. Chan, A. Wong, and H. Wong. A Guide to Developing Client-Server SQL Applications. Morgan Kaufmann, 1992.

[LE90] P. LaRowe and C. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. Technical Report CS-1990-10, Duke University, April 1990.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, September 1986.

[LLG$^+$92] D. Lenoski, J. Laudon, K. Gharachorloo, D. Weber, A. Gupta, J. Hennessy, and M. Horowitz. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3), March 1992.

[LLM88] M. Litzkow, M. Livny, and M. Mukta. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System . *Communications of the ACM*, 34(10), October 1991.

[LWY92] A. Leff, J. Wolf, and P. Yu. LRU-based Replication Strategies in a LAN Remote Caching Architecture. In *Proceedings of 17th Annual Conference on Local Computer Networks*, Minneapolis, MN, Sept 1992.

[LWY93] A. Leff, J. Wolf, and P. Yu. Replication Algorithms in a Remote Caching Architecture. *IEEE Transactions on Parallel and Distributed Information Systems*, 4, August 1993.

[Meh94] M. Mehta. *Resource Allocation in Parallel Shared-Nothing Database Systems.* PhD thesis, University of Wisconsin-Madison, 1994.

[MHL$^+$92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), March 1992.

[MN91] C. Mohan and I. Narang. Recovery and Coherency Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment. In *Proceedings of the International Conference on Very Large Data Bases*, Barcelona, September 1991.

[MN94] C. Mohan and I. Narang. ARIES/CSA:A Method for Database Recovery in Client-Server Architecture. In *Proceedings of the ACM SIGMOD Internationational Conference on Management of Data*, June 1994.

[MS93] J. Melton and A. Simon. Understanding the New SQL: A Complete Guide. Morgan-Kaufmann, 1993. A soft introduction to SQL2.

[PCGK89] D. Patterson, P. Chen, G. Gibson, and R. Katz. Introduction to Redundant Array of Inexpensive Disks (RAID). In *Proceedings of IEEE Compcom*, Spring 1989.

[PLC95] S. Pakin, M. Lauria, and A. Chen. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing*, 1995.

[PLKC90] C. Pu, A. Leff, F. Korz, and S. Chen. Redundancy Management in a Symmetric Distributed Main-Memory Database. Technical Report CUCS-014-90, Columbia University, 1990.

[Rah92] E. Rahm. Performance Evaluation of Extended Storage Architecture for Transaction Processing. In *Proceedings of the International Conference on Management of Data*, June 1992.

[Rah93] E. Rahm. Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Databbase Sharing Systems. *ACM Transactions on Database Systems*, 18(2), June 1993.

[RC89] J. Richardson and M. Carey. Persistence in the E Language: Issues and Implementation. *Software-Practice and Experience*, December 1989.

[RLD94] S. Reinhardt, J. Larus, and Wood. D. Tempest and Typhoon:User Level Shared Memory. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, Washington, DC, April 1994.

[Sch90] H. Schwetman. CSIM Users' Guide. MCC Tech Report ACT-126-90, Microelectronics and Computer Technology Corp., March 1990.

[SD91] B. Schilit and D. Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report CUCS-004-91, Columbia University, 1991.

[Str91] B Stroustrup. The C++ Programming Language. Addison-Weseley, 1991. ISBN0-201-53992-6.

[Tec91] Versant Object Technology. VERSANT System Reference manual, Release 1.6, 1991.

[TL93] C. Thekkath and M. Levy. Limits to Low-Latency Communication on High-Speed Networks. In *ACM Transactions on Computer Systems*, pages 179–203, San Diego, May 1993.

[VLN95] S. Venkataraman, M. Livny, and J. Naughton. Impact of Data Placement on Memory Management for Multi-Server OODBMS. In *Proceedings of the 11th IEEE International Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[WD94] S. White and D. DeWitt. QuickStore: A High Performance Mapped Object Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1994.

[WD95] S. White and D. DeWitt. Implementing Crash Recovery in QuickStore: A Performance Study. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1995.

[WN90] W. Wilkinson and M. Neimat. Maintaining Consistency of Client Cached Data. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[WR91] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, June 1991.

[Zip49] G. Zipf. Human Behavior and the Principle of Least Effort. Addison-Wesley, 1949.