1162

CACHING AND MEMORY MANAGEMENT IN CLIENT-SERVER DATABASE SYSTEMS

By

Michael Jay Franklin

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1993

1162

Abstract

The widespread adoption of *client-server* architectures has made distributed computing the conventional mode of operation for many application domains. At the same time, new classes of applications have placed additional demands on database systems, resulting in an emerging generation of object-based database management systems. The combination of these factors gives rise to significant challenges and performance opportunities in the design of modern database systems. This thesis proposes and investigates techniques to provide high performance and scalability for these new database systems, while maintaining the transaction semantics, reliability, and availability associated with more traditional database architectures. The common theme of the techniques developed in the thesis is the utilization of client resources through caching-based data replication.

The initial chapters motivate and describe the architectural alternatives for client-server database systems, present the arguments for using caching as the basis for constructing *page server* database systems, and provide an overview of other environments in which caching-related issues arise. The bulk of the thesis is then focused on the development and simulation-based performance analysis of algorithms for data caching and memory management. A taxonomy of transactional cache consistency algorithms is developed, which includes the algorithms proposed in this thesis as well as other algorithms that have appeared in the literature. A performance study of seven proposed algorithms is then presented. The study shows that significant performance and scalability gains can be obtained through client caching, and quantifies the performance impact of the tradeoffs that are identified in the taxonomy.

The remainder of the thesis extends the caching-based techniques to further improve system performance and scalability. Two major extensions are proposed and investigated. The first extension is the development of algorithms to efficiently manage the "global memory hierarchy" that results from allowing client page requests to be satisfied from the caches of other clients, thus avoiding disk accesses at the server. The second extension involves algorithms for using local client disks to augment client memory caches. Both extensions are shown to be simple and effective ways to reduce dependence on server disk and cpu resources.

Acknowledgements

I have been very fortunate to have the opportunity to work with Prof. Mike Carey. Mike has been a truly great Ph.D. advisor. He always made time to talk despite his constantly heavy workload, and he never failed to quickly figure out things that I had struggled with for days or weeks. He stuck with me through the many ups and downs of my graduate school career, and "encouraged" me at the proper times. I have also had the privilege of working closely with two other professors. Miron Livny has been my co-advisor for much of the research that is presented in the thesis. Miron never failed to come up with an angle on the work that neither I nor anyone else had thought of. Miron spent many hours of his time working with me and made numerous valuable contributions to my work. Furthermore, the research in this thesis could not have been done without the tools that he provided. The third member of my advising troika has been David DeWitt. David takes a lot of pride in the department and the database group, with good reason — he is responsible for making things happen around here. And although he doesn't like the word to get around, he is also a really nice guy.

Many other people have been responsible for making the department a great place to be. I learned much from working with the members of the EXODUS and SHORE projects. In particular, I had a lot of fun working with Mike Zwilling, who was more than patient as I fumbled around the EXODUS code. Beau Shekita was a contributor to the early work on caching, which led to this thesis. My two most recent officemates, Praveen Seshadri and Stacia Wyman, deserve much credit for keeping me from getting too much work done and for putting up with my sometimes(?) grumpy behavior. Susan Dinan and Sheryl Pomraning worked hard to shield the rest of us from the often perplexing machinations of the travel pre-audit office and other university bureaucracies. The Computer Systems Lab also deserves recognition for providing a top-rate computing environment. I learned much from my interactions with the other students in the database group, including: Joe Albert, Paul Bober, Kurt Brown, Shaul Dar, Joey Hellerstein, Hui-I Hsiao, Shahram G++, Hwee-Hwa Pang, Manish Mehta, Donovan Schnieder, Praveen Seshadri, Divesh Shrivastava, V. Srinivasan, S. Sudarshan, Prakash Sundaresan, C.K. Tan, Manolis Tsangaris, Seth White, and Markos Zaharioudakis, among others.

I also owe a great debt to many people from my pre-Wisconsin days. In particular, I would like to thank Phil Bernstein for all of his help through the years. Much of my understanding of computer systems stems from two excellent courses that Phil taught at the Wang Institute. Since that time, he has provided invaluable help and advice. Phil deserves much of the credit for getting me on the path that has led to this degree. Another debt that I have been looking forward to acknowledging is to Haran Boral and all the members of the Bubba project at MCC. Haran took a chance on hiring me seven years ago, and gave me the opportunity to work in what was most likely the premier database research group of the time. That experience was one of the main reasons that I came to graduate school; I realized that I was not likely to find a job as interesting, challenging, and fun as the one that I had there. More recently, a number of people have taken the time to give me help and advice, particularly as I was entering the strange world of the Ph.D. job market. These people include Phil Bernstein, David DeWitt, Jim Gray, Dave Lomet, Jeff Naughton, Todd Proebsting, Mike Stonebraker, Prasoon Tiwari, and Gerhard Weikum. I am especially indebted to Gerhard, as he was willing to put his reputation on the line by being one of my references. Prof. Raphael Lazimy deserves thanks for serving as the external member of my thesis committee. Sharon Malek patiently clarified the complex relationship between response time and throughput for me. Brian Hart has provided much discussion and encouragement for my work and for the decision to go back to school in the first place. Munir Cochinwala deserves credit for daring me into doing several dumb things such as getting a Ph.D. and running a marathon.

I would also like to thank the many people who have made Madison a fun place to be. Kurt Brown and Scooter deserve special recognition for letting me share their comfortable house during the past year. Madison would have been a cold and lonely (and hungry) place, however, if it hadn't been for Janice Ng. Her smile and laugh could melt Lake Mendota in January — she is responsible for keeping me sane these past couple years. I would thank her for putting up with my long hours except that she works harder than anyone I know.

Finally, I would most like to thank my parents, Marilyn and Martin, my sister Sherryl, and my brother David. Their love and support has kept me going and given me confidence. They have encouraged me in whatever I have chosen to do and have always been there for me when I needed them. This thesis is dedicated to them.

Contents

Chapter 1

Introduction

In recent years, powerful technological and market forces have combined to affect a major shift in the nature of computing and data management. These forces have had a profound effect on the requirements for Database Management Systems (DBMSs) and hence, on the way such systems are designed and built. The widespread adoption of *client-server* architectures has made distributed computing the conventional mode of operation for many application domains. At the same time, new classes of applications and new programming paradigms have placed additional demands on database systems, resulting in an emerging generation of object-based database systems. The combination of these factors gives rise to significant challenges and performance opportunities in the design of modern DBMSs. In this thesis, I propose and investigate a range of techniques to provide high performance and scalability for these new database systems while maintaining the transaction semantics, reliability, and availability associated with more traditional centralized and distributed DBMSs. The common theme of the techniques developed here is the utilization of client resources through caching-based data replication.

1.1 Background and Motivation

1.1.1 Client-Server Computing

In the late 1970's and early 1980's the advent of cheap, high-performance microprocessors, memories, and disks enabled the placement of substantial computing resources on individual desktops. The proliferation of personal computers and workstations, however, resulted in a polarization of data management in large organizations. Desktop computers were often considered to be outside the realm of the traditional Management Information Systems (MIS) organization and thus, the use of desktop computers was limited to the management of data that

was primarily of use to individuals. In contrast, data that was intended to be used at the corporate or departmental levels remained in the large mainframes and minicomputers managed by the MIS staff. This polarization resulted in gross inefficiencies in data management, and inhibited access to potentially valuable data.

As desktop machines have become more powerful, with price/performance characteristics that far surpass those of the larger systems, a natural pressure has arisen towards utilizing desktop machines for corporate and departmental functions in order to reduce reliance on the more expensive systems. Furthermore, the increased power of desktop machines has made them applicable to new application areas such as Computer Aided Design (CAD) and Office Automation, which require coordination among workgroups. Concurrently with these developments, advances in networking hardware and software have made it possible to efficiently connect the desktop systems of large organizations. The result of these forces has been the emergence of client-server architectures.

A client-server system is a distributed software architecture that executes on a network of interconnected desktop machines and shared server machines. Client processes typically execute on the desktop machines where they provide responsive interaction with individual users. Server processes are shared resources that provide services at the request of multiple clients. The partitioning of responsibility that is inherent in the client-server structure accommodates the needs of users while providing support for the management, coordination, and control of the shared resources of the organization. As a result, the client-server model provides an effective way to structure distributed systems.

1.1.2 Object-Oriented Database Systems

At the same time that distributed computing was evolving due to the client-server model, the database community was also undergoing a major shift in direction. This shift has lead to the construction of "object-based" database systems. The 1980's saw the maturation of relational database systems and their commercial acceptance for use in business applications. The advantages of relational systems include data independence (i.e., the ability to access the database without knowing its actual storage structure) and a declarative (and ultimately, standardized) data manipulation language (i.e., SQL). Despite the success of relational systems for business data processing, however, the limitations of relational systems have hindered their acceptance in other application areas.

For large classes of applications ranging from CAD and Software Development Environments to Geographical Information Systems (GIS), relational database systems provide neither the modeling power nor the responsiveness to be of use.[1] The restricted type system of relational databases (flat tuples and attributes) is not

[1] The DBMS requirements of such applications are surveyed in [Catt91b].

rich enough to easily represent the complex structures that these applications require and results in an "impedance mismatch" between the type system of the DBMS and that of the programming language used to manipulate the data once extracted from the DBMS [Zdon90]. Also, the mode of interaction provided by the relational DBMSs (i.e, queries and cursors) is neither responsive nor flexible enough to support the more interactive, navigational style of access of these advanced applications.

These problems have provided the impetus for the development of a new generation of DBMSs to efficiently support advanced applications. These new database systems are referred to as *Object-Oriented Database Management Systems (OODBMSs)*. OODBMSs provide a closer integration between the database and the programming language, allowing complex objects to be represented directly in the database and accessed using navigation (i.e. "pointer chasing") within the programming language.

1.1.3 New Database System Architectures

The confluence of these two trends, client-server computing and object-based database systems, has produced a new class of database systems. These systems use the client-server computing model to provide both responsiveness to users and support for complex, shared data in a distributed environment. Client processes, which execute close to the users, allow for fine-grained interaction with the database system as required by the advanced applications that utilize object-based DBMSs. Server processes manage the shared database and provide transaction management support to allow the coordination of database access by multiple users distributed across the network. Examples of this new class of database systems include research systems such as ORION [Kim90] and Client-Server EXODUS [Exod93, Fran92c], as well as products such as O2 [Deux91], Objectivity [Obje91], ObjectStore [Lamb91], Ontos [Onto92], and Versant [Vers91].

While there has been significant ongoing work in the development of these new database systems, many of the basic architectural and performance tradeoffs involved in their design are not yet well understood. The designers of the various systems have adopted a wide range of strategies in areas such as process structure, client-server interaction, concurrency control, transaction management, and memory management. This thesis investigates several fundamental aspects of this emerging generation of database systems. The strategies that are proposed and analyzed strive to provide clients with high performance data access while working within the constraints that are endemic to the client-server environment.

1.2 Contributions of the Thesis

This work in this thesis is based on two main premises:

1. The key to performance and scalability in client-server database systems is the exploitation of the plentiful and relatively inexpensive resources provided by client machines.

2. The use of *client caching* is the most effective way to exploit client resources without sacrificing availability in a client-server DBMS.

The following chapter describes client-server database systems in more detail, providing arguments for the above two premises. The bulk of the research in the thesis is then focused on the development and performance analysis of algorithms for data caching and memory management. All of the algorithms proposed seek to offload shared resources without sacrificing correct transaction semantics or reliability. Three main research topics are addressed: 1) consistency maintenance for data cached in client memories, 2) the use of remote client memories to reduce server disk demands, and 3) the issues raised by the use of client disks to extend client memory caches. For each of these topics, several algorithms are proposed and the performance tradeoffs among them are examined. The performance analyses employ a detailed simulation environment that has been constructed based on insights provided by implementation experience with the client-server version of the EXODUS storage manager.

The major contributions of the thesis are the following:

- **Cache consistency maintenance:** Allowing clients to cache portions of the database in their local memory is an effective way to reduce dependence on the server. However, if clients are allowed to retain their caches across transaction boundaries, cache consistency protocols are required to ensure that all clients see a consistent (serializable) view of the database. This thesis provides a taxonomy of such protocols, presents a detailed analysis of the performance tradeoffs inherent in their design, and develops several new protocols. The main results include: 1) algorithms that avoid access to stale data by not allowing such data to reside in client caches perform better than algorithms that detect the presence of stale cached data, 2) invalidating cached copies is shown to be more efficient and robust under many types of data sharing than propagating changes to remote sites, and 3) a dynamic algorithm is proposed and shown to provide the benefits of invalidation while performing well under workloads where propagation is advantageous.

- **Global memory management:** By exploiting copy information that is maintained by cache consistency algorithms, clients can be provided with access to a *global* memory hierarchy. Three global memory

management techniques are proposed and studied in this thesis. All of the techniques are designed to work within the context of the client-server DBMS caching algorithms developed in the first part of the thesis. The three techniques are: 1) clients are allowed to utilize the entire aggregate memory of the system by obtaining pages from other clients, 2) page replacement policies at the server are modified to reduce replication between the buffer pool contents of the server and its clients, and 3) a simple protocol between clients and servers is used to "extend" the client caches by migrating pages from clients to the server's memory. The techniques are examined in various combinations and are shown to provide substantial performance advantages under many workloads.

- **Client disk caching:** Additional performance and scalability gains can be obtained by employing client disks for caching. The use of local disks qualitatively changes the utility of caching at client workstations. Their lower storage cost as compared to client main memory allows a greater volume of data to be cached, possibly enabling a client to maintain local copies of all data items of interest to a given application. The non-volatility of disk storage facilitates long-term data caching *spanning numerous database sessions*. Due to performance and robustness considerations, however, disk caching is not a straightforward extension of memory caching. The thesis investigates the following issues relating to client disk caching: 1) the tradeoffs inherent in keeping a large volume of disk-resident data consistent, 2) techniques for bringing a disk-resident cache up-to-date after an extended off-line period, and 3) methods to reduce the work performed by the server to ensure transaction durability. Client disk caching is shown to be an effective means of offloading shared resources, particularly for high-locality workloads. Client disk caching can also serve as the basis for exploiting idle workstations and for supporting disconnected operation.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows:

- The next chapter describes client-server DBMSs in more detail. First, alternative architectures are described and evaluated. The choice of the *page server* architecture as the basis for the thesis is motivated. A reference page server architecture is then presented at a high level. Next, the critical performance and reliability characteristics of page server DBMSs are described. These characteristics play a crucial role in the design of the algorithms that are studied in the following chapters. Finally, other contexts in which similar problems arise (e.g., distributed file systems, shared-memory multiprocessors, etc.) are discussed briefly, and the differences between those environments and a page server DBMS are outlined.

- Chapter 3 provides a description of a detailed simulation model of a page server DBMS which has been used to obtain the performance results of the thesis. Nearly two dozen caching and memory management algorithms have been implemented within the model. A description of the workload model is provided along with two example workloads. An overview of the methodology employed in using the model for experimentation is also presented.

- Chapter 4 provides an introduction to cache consistency issues in page server DBMSs. A taxonomy of cache consistency maintenance algorithms is presented. The algorithms that are subsequently studied in Chapter 5 are described and a brief survey of other algorithms from the literature is provided.

- Chapters 5, 6, and 7 contain the main research contributions of the thesis, in the areas of client cache consistency, global memory management, and client disk caching, respectively. An overview of and comparison with closely related work is presented at the end of each chapter.

- Finally, Chapter 8 summarizes the conclusions of the thesis and outlines avenues for future work.

Chapter 2

Client-Server Database Systems

2.1 Architectural Alternatives

Client-server DBMS architectures can be categorized according to the unit of interaction among client and server processes. In general, clients can send data requests to the server as queries or as requests for specific data items. Systems of the former type are referred to as *query-shipping* systems and those of the latter type are referred to as *data-shipping*. These two alternatives are shown in Figure 2.1. In query-shipping systems, clients send a query to the server; the server then processes the query and sends the results back to the client. Queries may be sent as plain text (e.g., SQL), in a compiled representation, or as calls to precompiled queries that are stored at the server. In contrast, data-shipping systems perform the bulk of the work of query processing at the clients, and as a result, much more DBMS functionality is placed at the clients (see Figure 2.1). Data-shipping systems can be further categorized as *page servers*, which interact using physical units of data (e.g., individual pages or groups of pages such as segments), and *object servers*, which interact using logical units of data (e.g., tuples or objects).[1] In a page server system, the client sends requests for particular database pages to the server. The server returns each requested page (and possibly others) to the client. The client is responsible for mapping between objects and pages. In an object server system, the client requests specific objects from the server; the server is responsible for mapping between objects and pages. In all of these architectures, servers are ultimately responsible for providing database access, concurrency control, and recovery support. The arguments in favor of each of the three types of systems, query-shipping, page server, and object server, are presented in the following sections.

[1] Objects are logical in that they may occupy only a portion of a page of storage or may span multiple pages.

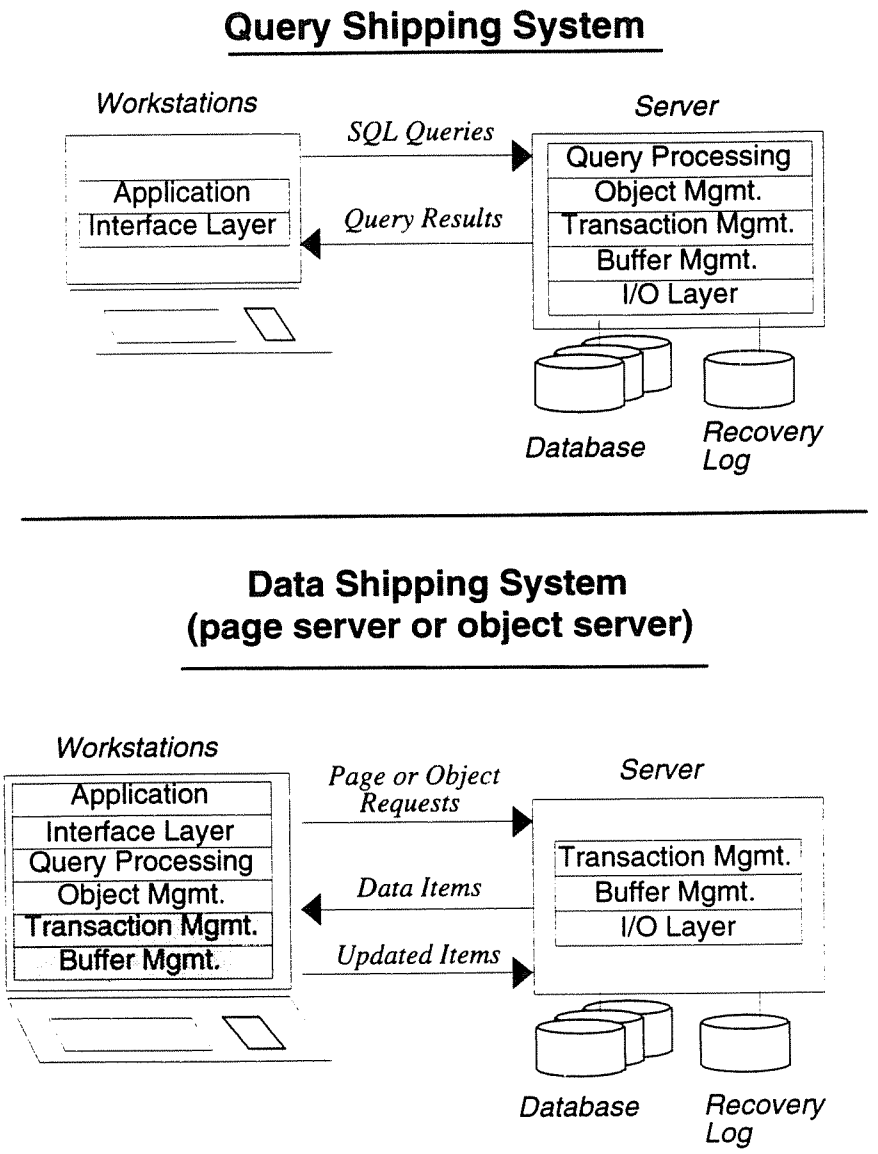


Figure 2.1: Architectural Alternatives

2.1.1 Query-Shipping Architectures

Most commercial relational database systems have adopted query-shipping architectures [Khos92]. The case for query-shipping architectures is summarized in [Hagm86, Comm90]. The advantages of query-shipping relative to data-shipping are the following:

1. Communication costs and client buffer space requirements are reduced since only the data items that satisfy a given query are transferred from the server to clients [Hagm86]. This is especially important in a wide area network [Ston90].

2. Query-shipping provides a relatively easy migration path from an existing single-site system to the client-server environment since the database engine (which resides solely on the server) can have a process structure similar to that of a single-site database system. Therefore, standard solutions for issues such as concurrency control and recovery can be used.

3. Interaction at the query level facilitates interaction among heterogeneous database systems using a standardized query language such as SQL.

2.1.2 Data-Shipping Architectures

In contrast to relational DBMSs, virtually all commercial OODBMS products and recent research prototypes have adopted the data-shipping approach. The advantages of these architectures for OODBMSs are mainly performance-related. Workloads that demonstrate the performance advantages of data-shipping architectures are the 001 (or "Sun") engineering database benchmark [Catt91a] and the more recent 007 benchmark [Care93a]. The advantages of data-shipping with respect to query-shipping are the following:

1. Data-shipping approaches offload functionality from the server to the clients. This may prove crucial for performance, as the majority of processing power and memory in a workstation-server environment is likely to be at the clients.

2. Scalability is improved because usable resources are added to the system as more clients are added. This allows the system to grow incrementally.

3. Responsiveness is improved by moving data closer to the application and allowing the programmatic interfaces of OODBMSs to directly access that data in the application's address space. The importance of

the programmatic (vs. query-based) interface to database systems is discussed in [Atki89].[2]

2.1.3 Page-Server vs. Object-Server Architectures

There are also tradeoffs between the two data-shipping approaches: object server and page server. The performance tradeoffs are investigated in [DeWi90a] and are somewhat workload-dependent. Object servers have the following advantages:

1. Client buffer space can be used efficiently, as only requested objects are brought into the client's buffer pool.

2. Communication cost can be reduced by sending objects between clients and servers rather than entire pages if the locality of reference of objects is poor with respect to the contents of pages.

3. Object-level locking is easily implemented at the server since it knows which clients are accessing which objects.

Page servers have the following advantages:

1. Effective clustering of objects into pages can result in fewer messages between clients and servers, especially for scan queries. Furthermore, the cost in terms of CPU cycles for sending a page is not much higher than that for sending an object (e.g., if pages are 4k bytes and objects are 200 bytes).

2. The page server design places somewhat more functionality on the clients, which is where the majority of the processing power in a workstation-server environment is expected to be.

3. The design of the overall system is simpler. For example, page-oriented recovery techniques can be efficiently supported in a page server environment [Fran92c]. Also, in a page server system, there is no need to support two query processors (one for objects, and one for pages) as is done in some object server systems, such as ORION-1SX [Kim90].

2.1.4 Discussion

Given the above arguments, it appears that current trends in software and hardware will work to the advantage of data-shipping architectures over query-shipping. In terms of software, an important trend is the support of

[2] It should also be noted that some researchers dispute the need for navigational interfaces [Comm90]. If navigational interfaces do not achieve widespread use, then obviously, the relative disadvantages of query-shipping for such interfaces become moot.

programmatic and navigational interfaces by object-oriented database systems. These systems allow applications to access data in their virtual address space either directly (e.g., [Cope90, Shek90]) or through "pointer swizzling" (e.g. [Lamb91, Whit92]). Therefore, it is necessary to bring the data to the machine on which the application is running (i.e., the client workstation). Another important software trend is the ongoing improvement in workstation operating systems. Support for features such as lightweight threads, RPC, and flexible buffer management will allow sophisticated database function to be more easily and efficiently implemented on workstations. Relevant hardware trends include significant gains in workstation processor speeds, memory capacity and price/performance, which benefit data-shipping architectures because they better utilize the capabilities of the workstations. Also, improvements in network bandwidth (e.g., FDDI) will reduce the likelihood of the communication wire becoming a bottleneck, thereby reducing the penalty for sending larger messages between clients and servers.

These same trends also favor page server approaches over object server approaches. The only advantage of object servers that is not reduced by the hardware trends above is the ability to do fine-grained locking. While it is beyond the scope of this thesis, algorithms that balance concurrency and overhead by dynamically adapting the granularity of locking can be implemented in the page server environment and can perform better than pure fine-grained schemes [Care93b]. Furthermore, the comparative simplicity of the page server approach makes it a more reasonable choice for implementation. For these reasons, the work reported in this thesis assumes a page server architecture as the base system. It should be noted, however, that the evolution of OODBMSs will likely result in systems that combine navigational access with more declarative, set-oriented access. The extension of the techniques of this thesis to support such hybrid systems is an important direction for future work, and is discussed further in Chapter 8.

2.2 Page Server DBMS Reference Architecture

This section presents a reference architecture for a page server DBMS. The reference architecture, which is shown in Figure 2.2, serves as a foundation for the remainder of the dissertation. In a page server DBMS, applications run at the client workstations. The DBMS consists of two types of processes that are distributed throughout the network. First, each client workstation runs a Client DBMS process. This Client DBMS process is responsible for providing access to the database for the applications running at its local workstation. For protection reasons, the applications run in a separate address space from their local Client DBMS process, though some shared-memory may be used for efficiency (e.g., for communication). Applications send requests

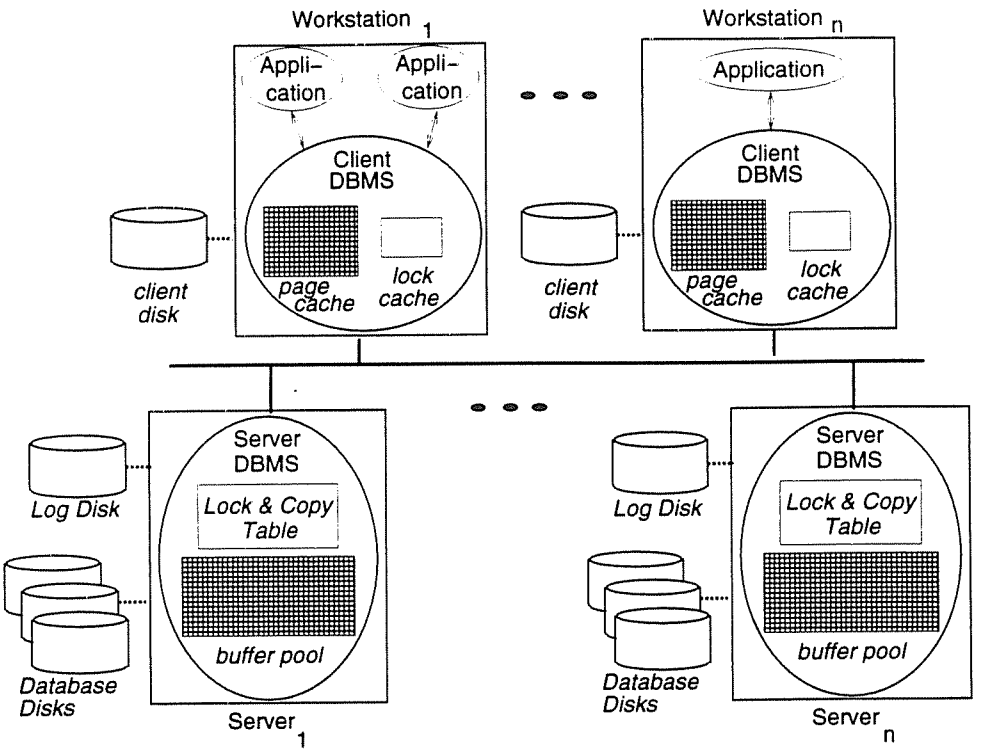


Figure 2.2: Reference Architecture for a Page Server DBMS

to their local Client DBMS process, which executes the request, in turn sending requests for transaction support and for specific data items to Server DBMS processes. Server DBMS processes are the actual owners of data: they are ultimately responsible for preserving the integrity of the data and for enforcing transaction semantics. The Server DBMS processes manage the stable storage on which the permanent version of the database and the log reside. They also provide locking and copy management functions for the data that they own, as is described in later chapters. In this reference architecture, the database is assumed to be statically partitioned across the servers. While it is possible to replicate data among the servers in order to improve availability and/or performance, server replication is beyond the scope of this thesis.

The database itself is divided into fixed length *pages*, which are relatively small (on the order of four or eight Kbytes). The Client DBMS process at a client is responsible for translating local application requests into requests for specific database pages and for bringing those pages into memory at the client. As a result, all of the pages required by an application are ultimately brought from the server(s) to the clients. As described in the following section, pages may be cached at the clients in order to reduce transaction path length and server load. A Server DBMS process is responsible for providing the most recent committed values for the data pages that it

owns in response to client requests; of course, due to concurrency conflicts, it may not be possible for the server to provide the page immediately.

An important aspect of page server systems is that all of the DBMS processes (both Client and Server) have memory that they use for buffering database pages. Therefore, the database system as a whole is responsible for managing buffers across the entire collection of machines. This allows the database system to make allocation and replacement decisions based on global constraints and to coordinate buffer management at the various sites accordingly. Similarly, the other resources of the system, such as CPUs and disks, can also be exploited by the DBMS. The efficient utilization of these distributed resources (CPUs, memories, and disks) in a page server context is the primary focus of this dissertation. The next section discusses the overall strategy for utilizing these resources, as well as the properties of page server DBMSs that constrain the design space of techniques to achieve this utilization.

2.3 Utilizing Client Resources

2.3.1 Improving Performance and Scalability

One implication of the reference architecture presented above is that the majority of CPU power and memory in the system will be located at the clients. While a server will typically have more CPU power and main memory than any single workstation, the aggregate CPU power and main memory capacities of all the clients in the system will be much greater than that of the server(s). As a result, offloading functionality from the servers to the clients can provide both performance and scalability benefits. Performance benefits are realized as latency and throughput improvements. By reducing the need to communicate with the server, the overall pathlength of transactions can be shortened, thereby improving latency. In addition, for a heterogeneous system, clients can cache data in their native format, thus avoiding the conversion costs that would be required when importing data items from a remote site. System throughput can be improved through the offloading of shared resources such as server machines and the network, which are potential bottlenecks.

The offloading of shared resources also improves the scalability of the system. As clients are added to the system, they provide additional load which could cause throughput bottlenecks. However, if the resources of these additional clients are exploited, then the demand on the shared resources can be minimized, thereby allowing more clients (and users) to be accommodated. Despite progress in the development and commercialization of more scalable server machines, the addition of client workstations remains the most cost-effective way to grow a shared computing system. This is largely because server machines have yet to attain the commodity-level pricing

of desktop machines. Due to the fierce market competition and the development of low-overhead distribution channels for desktop machines (such as retail stores and mail order), this disparity is unlikely to change in the foreseeable future.

Another important benefit of utilizing client resources is predictability. Hennesy and Patterson state, "The attraction of a personal computer is that you don't have to share it with anyone. This means response time is predictable, unlike timesharing systems" [Henn90]. Unfortunately, this is only true to the extent that access to shared resources can be avoided. For example, if data must be obtained from a remote server, then the time required to respond to a user request will vary depending on the load factors of the server and the network. If the data can be obtained locally at the workstation, however, then its access time is independent of the activity of other users in the network.

2.3.2 Correctness and Availability Considerations

While the potential gains to be realized by exploiting client resources are great, they are ultimately limited by correctness and availability constraints. In terms of correctness, workstation-server database systems must be capable of providing the same level of transaction support as more traditional database architectures. Transactions have both concurrency and failure correctness constraints. This level of transaction support is known informally as the ACID properties [Gray93]:

Atomicity: Either all of the operations of a transaction complete successfully, or none of them do.

Consistency: Transactions preserve any integrity constraints defined on the data.

Isolation: A transaction sees a state of the database that could occur if that transaction were the only one running on the system — despite concurrent execution.

Durability: The effects of committed transactions survive failures.

The notion of isolation is traditionally referred to as *serializability*. Serialability is the property that a (possibly interleaved) execution of transactions has the same effect on the database, and produces the same output as some serial (i.e., non-interleaved) ordering of the transactions [Bern87]. Due to the requirement for serializability, the items in the database become shared resources that are subject to contention in much the same way as shared *physical* resources such as the server and network. In order to provide serializable transaction execution, data access conflicts must be properly handled. Data conflicts are typically resolved using blocking (for pessimistic

schemes) or transaction restarts (for optimistic methods), which can both limit the performance of the system [Agra87].

Much research has been performed on the relaxation of serializability, particularly for distributed databases (e.g., [Davi85, Alon90, Pu91b]). This thesis, however, adopts the more stringent correctness criterion. This requirement is made for two reasons. First, relaxed consistency models are not applicable to all database applications, and thus, the inability to support the ACID properties would limit the acceptance of page server DBMSs. Second, once the mechanisms for providing the ACID properties are in place, they can be relaxed in order to provide additional performance for applications that have less stringent requirements. For example, the notion of "degree 2 isolation" allows some transactions to read data in an unserializable manner, without impacting the correctness of other transactions [Gray93].

The amount of data contention present in the system is largely workload dependent, and thus, techniques that exploit client resources will be more effective in application environments that do not have very high levels of data contention or in which contention can be minimized through the use of techniques such as versioning or check-in/check-out access paradigms. Fortunately, as is shown in the following chapters, support for strict transaction semantics can be made quite efficient in the page server environment.

In terms of availability, workstation-server environments have inherent asymmetries that impact the extent to which responsibility for data and transaction management can be placed at the client workstations. These asymmetries include both physical and environmental ones. The physical differences result from economies of scale that dictate the placement of resources at client or server machines. For example, it is likely to be more cost effective to place duplexed log disk storage or non-volatile memory at server machines than to place such functionality at each client machine. Therefore, it may not be possible (or efficient) to associate a separate local log with each client. The environmental differences result from the fact that client machines are typically placed in close proximity to users, while server machines are kept under the tighter control of an operations staff (e.g., in separate machine rooms rather than on individual desks). Often, individual users are given responsibility for the configuration and management of their workstations. Furthermore, as portable client machines become increasingly popular, it cannot be assumed that a client is always connected to the network. For these reasons, the server machines are inherently more available and reliable than client machines. As a result, care must be taken so that the failure (e.g., crash or disconnection) of one client workstation does not impact the availability of data for applications running at other client workstations. This constraint limits the extent to which clients can be allowed to manage data independently (i.e., without interaction with servers).

2.3.3 The Case for Client Caching

The previous subsections have argued the following two points:

1. There are substantial gains to be obtained through the utilization of client resources.
2. Methods that attempt to realize these gains must do so within the correctness and availability constraints of the workstation-server DBMS environment.

In this section the case is made that the combination of these two factors leads to the adoption of *client caching* as a fundamental technique in the design of a page server DBMS. Caching is the storing of copies of data items close to where they are needed in order to reduce the expense (in terms of time and/or dollars) of accessing them. Moreover, it is a technique that pervades all levels of computer system design. Examples are plentiful: the use of registers and processor caches to avoid memory accesses, the "memoization" of function results to avoid recomputation [Fiel88], the use of buffering in file systems and database systems to avoid disk accesses, etc.

For a page server system, client caching refers to the ability of clients to retain copies of database pages locally once they have been obtained from the server. In this context, client caching can be thought of in the following manner:

Client Caching = Dynamic Replication + Second-Class Ownership

Dynamic replication means that page copies are created and destroyed based on the runtime demands of clients. When a client needs to access a page, a copy of that page is placed in the client's cache if one does not already exist. Page copies are removed from a client's cache in order to make room for more recently requested ones or, under some caching algorithms, because they become invalid. Therefore, the amount of replication that exists in the system at any given time is driven by the recent activity of the clients. This is in contrast to *static* replication in which the replication of data is determined as part of the physical database design process. Changing the replication scheme under static replication requires a reorganization of the database. Static replication is typically associated with more traditional distributed database architectures such as Distributed Ingres [Ston79] (for example, see [Ceri84] chapter 4).

It is well known that replication can reduce data availability in the presence of updates and failures in a distributed environment [Davi85]. *Second-class ownership* allows consistency to be preserved without sacrificing availability. Second-class ownership refers to the fact that in client caching, the cached copies of pages are not considered to be the equals of the actual data pages, which are kept at the server. Specifically, the database process at the server always has locally, any and all data that is necessary for ensuring transaction durability

(e.g., data pages, logs, dirty page information, etc.), so client cached pages can be destroyed at any time without causing the loss of committed updates. This notion is crucial to data availability in a page server system, as it allows the server to consider a client to be "crashed" at any time, simply by unilaterally aborting any transactions currently active at that client. As a result, the server is never held hostage by an uncooperative or crashed client. The term "second-class ownership" is borrowed from a similar concept called "second-class replication" used in the CODA distributed file system [Kist91]. The two notions are similar in that a distinction is made between the "worth" of different types of copies. They differ, however, in that the second-class replicas of CODA are used to increase availability by allowing users to access *inconsistent* data, whereas in a page-server DBMS second-class copies are used to enhance the availability of *consistent* data.

As a consequence of the combination of dynamic replication and second-class ownership, client caching is a compromise between the utilization of client resources and the correctness and availability concerns raised in the previous subsection. Client resources are exploited by maintaining the cache, thereby reducing the need to obtain data from servers. Data that is in a client's local cache can typically be accessed faster than data that is at a server. Caching, however, does not transfer ownership of pages to clients. Servers remain the only true owners of data, and servers therefore are ultimately responsible for ensuring the correctness of transaction execution. Similarly, data availability also remains the responsibility of the servers rather than that of the (less reliable) clients. The sense in which caching is a compromise is that having servers maintain data ownership implies that they must remain involved in the execution of transactions. Therefore, some client autonomy is sacrificed. The key to making this compromise pay off is to minimize the extent of the server's participation in transaction execution. Minimizing this overhead is the goal of much of the research that appears in the following chapters of this thesis.

2.4 Related Systems

As stated in the previous section, caching is a fundamental technique that is used to enhance performance in many areas of computer systems design. As a result, work related to caching in a page server DBMS has been performed in many other contexts. This section address caching issues in these contexts, including: 1) other database system architectures, 2) distributed file systems, 3) multiprocessor architectures, and 4) distributed shared memory. Replicated data management is not addressed in its broadest sense (this is a research area in its own right), but rather, the focus here is on environments in which *dynamic* replication arises due to caching. This section is not meant to be an exhaustive survey of the work in these areas; rather, examples are used to examine

the general similarities and differences between those environments and page server DBMSs. Discussion of work directly related to the specific issues addressed in subsequent chapters of the thesis appears in the chapters themselves.

2.4.1 Dimensions for Comparison

While there are many similarities among the environments in which caching is used, each of the environments has its own unique set of tradeoffs. These tradeoffs result in different strategies and mechanisms for implementing caching, for managing cache contents, and for maintaining cache consistency. To compare the environments and to better understand the differences in their basic tradeoffs, it is helpful to examine them along four dimensions:

1. **Correctness criteria:** What guarantees are made to users about the consistency of replicated copies of data? What guarantees are made about the durability of updates? What are the failure semantics?

2. **Granularity:** What is (are) the unit(s) of data for which replication is managed (i.e., on which correctness guarantees are made)?

3. **Cost Tradeoffs:** What are the relative costs of obtaining copies of data items, maintaining consistency, etc.? These costs are highly architecture dependent. For example, the latency of point to point communication, the availability of broadcast or multicast communication, the communication bandwidth, and the relative speeds of storage devices, etc., all greatly impact the way that caching might best be done.

4. **Workload Characteristics:** What are characteristics of the workloads for which the system is designed or optimized? Some of the important issues here are the volume of data accessed, the degree of read-only and read/write sharing, the locality of access, and patterns in the flow of information.

With respect to these dimensions, page server DBMSs have the following properties:

Correctness criteria: As discussed in Section 2.3.2, a page server DBMS must support the ACID transaction properties.

Granularity: Fixed-length pages (on the order of 4 or 8 Kbytes) are the unit of granularity on which consistency is maintained.

Cost Tradeoffs: In a page server DBMS, assuming current technology, communication between the clients and servers is relatively expensive (on the order of one to several milliseconds, depending on the message size). In terms of the cost of data access, the penalty for a cache miss at a client is quite high; it includes

the cost of a round-trip message with the server, plus up to two disk accesses at the server, which each costs on the order of 10 to 25 milliseconds.[3]

Workload Characteristics: Aspects of workloads for page server DBMSs are not yet well understood. However, support for substantial sharing of data is a fundamental requirement. This includes simultaneous sharing as well as "sequential" sharing, in which data accessed by a transaction is also accessed by subsequent transactions at different sites. The data conflict rate is likely to be lower than for transaction processing workloads (e.g., the TPC benchmarks [Gray91]), however, data conflicts must be dealt with efficiently. Another important aspect of page server DBMS workloads is that some are likely to be highly interactive and thus, less tolerant of transaction aborts than more traditional DBMS workloads.

2.4.2 Other DBMS Architectures

Issues related to client caching also arise in database systems other than page server DBMSs, such as shared disk (or data sharing) systems, certain types of hybrid client-server DBMSs, distributed databases, and shared nothing parallel database systems. As would be expected, these are the closest systems to page server DBMSs in many ways, as they provide the same correctness guarantees.

Shared disk systems consist of a number of closely coupled processors, each with its own private memory, that share access to a farm of disks. Shared disk systems have been the subject of much study recently, including [Bhid88, Dan90a, Lome90, Moha91, Rahm91] and many others. Shared disk systems encounter a form of dynamic replication, as page copies that are read in from disk can reside in the memories of multiple nodes. They differ from page server DBMSs in several ways, however. First, they are typically designed to support transaction processing, so their workloads are expected to have less locality of access at individual nodes and a higher conflict rate. Secondly, the costs of communication among nodes is lower than would be expected in the local area network of a page server DBMS. Thirdly, the structure of a shared disk system is peer-to-peer rather than the client-server structure of a page server system. This third difference can be diminished somewhat through the use of shared "extended storage" [Rahm92] and global lock managers (as in [Moha91]). The impact of these similarities and differences on caching and memory management algorithms are discussed in the related work sections of subsequent chapters.

Caching issues also arise in certain types of hybrid client-server database systems. In these systems, all updates are performed at the server (as with query-shipping), but query results are cached at clients so

[3] A disk read is required if the requested page is not in the server's memory. A disk write may be required in addition if a dirty page must be replaced from the server's memory to make a buffer slot available for the requested page.

that subsequent queries can be answered locally (as with data-shipping). The ADMS± system developed at Maryland [Rous86] uses relational view-maintenance techniques to determine which updates need to be sent from the server to a client before executing a query at that client. Techniques to manage cached copies in a very different context, information retrieval (where updates are performed only at servers), are discussed in [Alon90]. These latter techniques are based on the relaxation of consistency constraints, however.

As stated earlier, replication has long been a subject of study for distributed database systems (e.g. Distributed Ingres [Ston79], SDD-1 [Roth80], R* [Will81], etc.). Replication for these systems is typically done statically as part of the physical database design, and the unit of replication is fairly coarse. More dynamic replication schemes for this environment have been proposed recently [Wolf92]. These latter algorithms evolve the replication scheme over time, however, rather than changing it on demand, as in a page server DBMS. Replication is also used in "shared nothing" parallel database systems in which the multiple processors of the system each have their own memory and disks (e.g., Bubba [Bora90] and Gamma [DeWi90b]). This replication is also determined statically and is primarily used for reconstructing the data owned by failed nodes. Replication schemes for shared nothing database systems are discussed in [Cope89], [Hsia90], and [Bhid92].

2.4.3 Distributed File Systems

Client caching has been used in distributed file systems since some of the earliest work in the area (e.g., DFS [Stur80]). Many distributed file systems that support some form of client caching have been proposed and built. These include Andrew [Howa88] and its follow on project CODA [Kist91], the Cedar File System [Giff88], Sprite [Nels88], the V system [Gray89], and Ficus [Pope90], to name just a few; a survey can be found in [Levy90]. As with page server DBMSs, these systems use client caching to improve performance and scalability. However, they support much less stringent notions of correctness in terms of both concurrency and failure semantics. For example, while most of the systems mentioned provide support for handling write-write conflicts, they do not provide support for detecting read-write conflicts among multi-operation units such as transactions. Thus, anomalies can arise even if individual read-write conflicts are ordered correctly (e.g., as in Sprite). In terms of failure semantics, file systems typically allow windows where crashes can result in lost updates, and most do not provide support (other than manual dumps) for recovery from media failures.

Another significant difference between distributed file systems and page server DBMSs lies in the assumptions they make about workload characteristics. An important assumption in most file systems is that users will manage the coordination of complex interactions on top of the file system (thus, transactions are typically not supported). As an example, coordination tools such as source change control systems are implemented by building a locking

mechanism on top of the file system. This assumption is borne out by the wide acceptance of distributed file systems such as NFS, which provide even lower levels of consistency than the systems described above. The workload characteristics of file systems have been studied at Berkeley (in [Oust85], and a recent follow-on study [Bake91]). A number of observations in those studies differ significantly from the characteristics for which page server DBMSs are designed, including:

- Random access to file contents is rare (they are usually read sequentially, and in their entirety).
- Concurrent read-write and write-write conflicts are rare.
- Fine-grained sharing of files is also rare.
- Many files are destroyed or rewritten soon after they are created.

Other recent studies indicate that the characteristics of commercial and production environments may be quite different [Rama92, Sand92]. However, the studies of research and academic environments have had a large impact on the design of distributed file systems.

Despite the differences in correctness guarantees and workload characteristics between distributed file systems and page server DBMSs, there is much similarity in their cost tradeoffs for caching. For example, the cost of obtaining a page of data from the server, the costs of communicating with the server for consistency purposes, and the costs of servicing cache misses are all very similar between the two types of systems. As a result, techniques used in distributed file systems can often be modified to work for a page server DBMS. For example, as is discussed in Chapter 4, one of the more promising schemes for maintaining page server DBMS cache consistency is a transactional adaptation of techniques developed for the Andrew and Sprite file systems.

2.4.4 Shared-Memory Multiprocessors

Another important area in which caching plays a large role is the design of shared-memory multiprocessor computers. In order to reduce traffic on the interconnection network and to reduce accesses to shared memory, processors typically have private local caches. Cache consistency in multiprocessors has been the subject of extensive research over the last ten years.[4] While there are obvious similarities between caching in multiprocessors and in page server DBMSs, the two domains differ significantly along all four of the dimensions outlined in Section 2.4.1. The most pervasive differences are along the cost tradeoff dimension. A large family of

[4] See [Arch86] for a performance analysis of many of the earlier schemes. A survey including more recent schemes appears in [Sten90].

multiprocessor caching algorithms depend on the use of a shared bus. These protocols are called "snooping" protocols [Good83] because all of the cache controllers must monitor the consistency traffic on the bus (i.e., they snoop) in order to maintain cache consistency. Snooping protocols make sense for small multiprocessors because the bus provides a cheap broadcast medium. Another protocol family, the directory-based schemes (see [Agar88]), store information on the location of cached data copies so that consistency messages can be directed to the sites that need to receive them. Directory-based schemes need not rely on a cheap broadcast facility, so they are closer in spirit to caching schemes for page server DBMSs[5]. Even in directory-based schemes, however, the overhead involved in communication for consistency messages is much lower than that for sending messages over a local area network in a page server DBMS.

There are also differences in the granularity of consistency maintenance. Cache lines are typically on the order of 10's of bytes, while database pages are typically on the order of 1000's of bytes. This difference in granularity impacts the amount of overhead that is acceptable when establishing the consistency of a granule on an initial access to that granule (e.g., obtaining a read lock on a database page). In terms of correctness, there are again substantial differences. Even *sequential consistency*, which is often considered to be too expensive a correctness model to enforce for a multiprocessor [Adve90], does not provide a degree of concurrency and/or failure atomicity that would be suitable for a DBMS. Finally, it is difficult to compare the assumptions made about workload characteristics in multiprocessors and DBMSs because there is such a wide gap in the granularities at which they are thought about (i.e., memory references vs. transactions). There are, however, some general similarities in terms of the expectation that substantial sharing does occur, and that locality of accesses will justify the effort involved in caching.

Despite the differences outlined above, both multiprocessors and page server DBMSs rely heavily on caching, and as a result, the two environments share many basic concepts. These include: tradeoffs between propagating changes to other copies and destroying those copies (i.e., write-update vs. write-invalidate), concern with the "false sharing" that can arise because the unit of caching is not exactly matched with the unit of access, and the desire to reduce the overhead required to maintain cache consistency.

2.4.5 Distributed Shared Memory

The final class of systems addressed here are called Distributed Shared Memory (DSM) systems [Li89, Nitz91, Judg92]. DSM (also referred to as Shared Virtual Memory [Li89]) provides the abstraction of a shared *virtual*

[5] It should be noted that some directory schemes store only limited directory information and rely on broadcast as a fallback mechanism [Agar88].

memory address space that spans the nodes of a distributed system. References to virtual address locations that reside on a remote machine are transparently handled by the system using a page fault mechanism. DSM implementations allow multiple readers to access the same page, so DSM systems must deal with the problems raised by dynamic replication. Unlike multiprocessor caching, which relies heavily on hardware support, DSMs are typically implemented in software with only minimal hardware assists. Because pages are the unit of consistency, DSMs are similar to page servers with respect to granularity. In terms of cost tradeoffs, DSMs are again closer to page servers than shared-memory multiprocessors, because messages are required for consistency (although if the DSM is implemented on a multiprocessor, these may be cheaper than in the local area network of a page server DBMS). One cost-related difference is that in some DSM algorithms, the ownership of a page can migrate (unlike in a page server DBMS, where the server is the owner of all pages). Migration can avoid bottlenecks due to a single owner site, but may result in a need for additional messages in order to find the owner of a page. In terms of workload characteristics, it is again difficult to compare the two environments because DSM is a general purpose mechanism (in fact, a database system can be implemented on top of a DSM, as described in [Hsu88]). The main differences are in terms of correctness criteria. DSM does not provide transaction support for concurrency or for failure. Consistency is provided in terms of individual reads and writes — so, higher-level consistency must be implemented on top of the DSM. Failures must be handled by techniques such as checkpointing which are not typically addressed by DSM systems.

Chapter 3

Modeling a Page Server DBMS

The performance analyses that are presented in the following chapters were obtained using a flexible and detailed simulation model The model consists of approximately 10,000 lines of code, written in DeNet [Livn90], a discrete event simulation language based on Modula 2. The simulation experiments themselves were performed using the Condor batch processing system [Litz88], which provided an abundance of CPU cycles through the exploitation of idle workstations. The simulation model was originally derived from the distributed database system model described in [Care91b], but it has been largely rewritten to reflect the page server architecture.

This chapter describes how the model captures the execution model, database, physical resources, and workloads of a page server DBMS. The general structure of the model is shown in Figure 3.1. Two types of sites are modeled: clients and servers. The model allows for an arbitrary number of each type of site; however, all of the experiments performed for this thesis model varying numbers of clients (in the range of 1 to 50) connected to a single server. All concurrency control and consistency maintenance is done at a page granularity.[1] Certain aspects of the system, such as concurrency control, consistency management, and buffer management, are modeled in full detail; other aspects, such as the database and its workload, are modeled more abstractly.

3.1 Client-Server Execution Model

The details of the actions that take place when transactions are executed differ widely depending on the cache consistency and concurrency control algorithms being used. This section describes the properties that are common to all of the algorithms, as well as the generic functions of the simulator. The algorithms themselves are described in more detail in Sections 4.3, 6.2, and 7.2. In the simulator, as shown in Figure 3.1, each client comprises five

[1] The model has since been extended to support finer-grained concurrency control, as described in [Care93b].

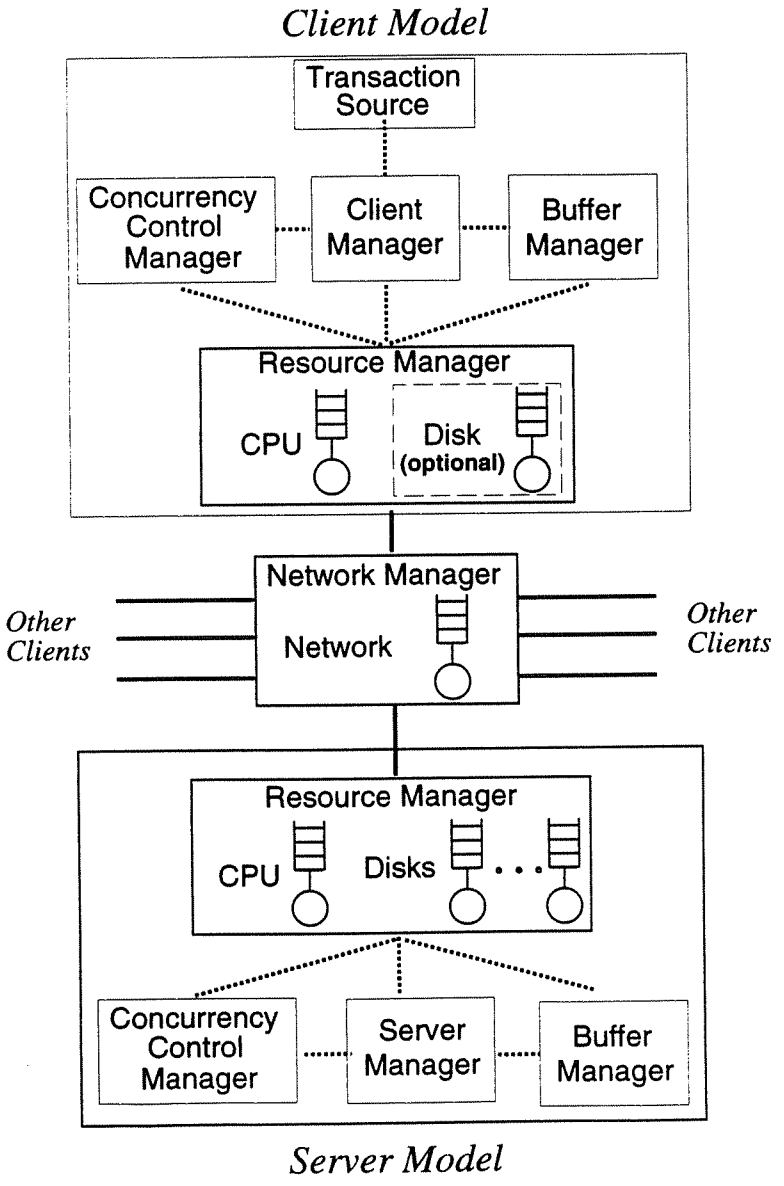


Figure 3.1: Simulation Model

modules. These are: a *Transaction Source*, which generates the workload as specified in Section 3.3; a *Client Manager*, which executes the transaction page reference strings generated by the *Transaction Source*, processes page access requests from local transactions, and responds to messages sent from the server; a *CC Manager*, which is in charge of concurrency control (i.e., locking) on the client; a *Buffer Manager*, which manages the client buffer pool and the disk-resident client caches used in Chapter 7; and a *Resource Manager*, which models the other physical resources of the client workstation. The server is modeled similarly to the clients, but with the following differences: the *CC Manager* has the ability to store information about the location of page copies in the system, and it also manages locks; there is a *Server Manager* component that coordinates the operation of the server (analogous to the client's *Client Manager*); and there is no Transaction Source module (since all transactions originate at client workstations).

Client transactions execute on the workstations that submit them. Transactions are represented as page reference (read or write) strings and are submitted to the client one-at-a-time. Upon completion of one transaction, the client waits for a random think time (with mean *ThinkTime*) and then begins a new transaction. When a transaction aborts, it is immediately resubmitted for execution with the same page reference string. When a transaction references a page, the *Client Manager* must first obtain a valid copy of the page in its local buffer pool. Once a local copy of the page exists, the transaction processes the page and then decides whether or not to update it. In the event of an update, the client obtains write permission for the page and performs further CPU processing on the page. The details of how to obtain valid page copies as well as how to obtain read and write permissions on the pages are highly algorithm dependent and are described along with the algorithms in the chapters that follow. Finally, at commit time, the *Client Manager* sends a commit request together with copies of any updated pages to the server. The commit processing that is initiated by the server is also algorithm dependent (as described in Section 4.3), and may include sending requests to remote clients to obtain locks and possibly updating data at the remote sites. For some algorithms, the commit process involves a two-phase commit protocol among the originating client, some or all of the remote clients, and the server. This process, if required, is managed by the server. When the commit processing for the transaction is completed, the server sends a message informing the client that the commit was successful.

3.2 Database and Physical Resource Model

The model parameters that specify the database and physical resources of the system are listed in Table 3.1. The database is modeled as a collection of *DatabaseSize* pages of *PageSize* bytes each. The number of client

workstations present for a given configuration is specified by the *NumClients* parameter. The client and server CPU speeds are specified in MIPS (*ClientCPU* and *ServerCPU*). The service discipline of the client and server CPUs is first-come, first-served (FIFO) for system services such as message and I/O handling. Such system processing preempts other CPU activity, for which a processor-sharing discipline is used. The sizes of the buffer pools on the clients and on the server (*ClientBufSize* and *ServerBufSize*) are specified as a percentage of the database size. The client and server buffer pools are both managed using an LRU replacement policy as a default, but this policy is modified for some of the memory management algorithms (see Chapter 6). Dirty pages are not given preferential treatment by the replacement algorithm, and they are written to disk when they are selected for replacement. Note that on clients, dirty pages exist only during the course of a transaction. Dirty pages are held on the client until commit time, at which point they are copied back to the server; once the transaction commits, the updated pages are marked as clean on the client.[2]

Parameter	Meaning
PageSize	Size of a page in bytes
DatabaseSize	Size of database in pages
NumClients	Number of client workstations
ClientCPU	Instruction rate of client CPU
ServerCPU	Instruction rate of server CPU
ClientBufSize	Per-client buffer size (% of database)
ServerBufSize	Server buffer size (% of database)
ServerDisks	Number of disks at server
ClientDisks	Number of disks per client
CliDiskCacheSize	Capacity of a client disk cache (% of database)
MinDiskTime	Minimum disk access time
MaxDiskTime	Maximum disk access time
DiskOverheadInst	CPU overhead for performing disk I/O
NetworkBandwidth	Network bandwidth
FixedMsgInst	Fixed number of instructions per message send or receive
PerByteMsgInst	Additional instructions per message byte send or receive
ControlMsgSize	Size of a control message (in bytes)
LockInst	Number of instructions per lock/unlock pair
RegisterCopyInst	Number of instructions to register/unregister a copy
DeadlockInterval	Global deadlock detection frequency (seconds)

Table 3.1: Resource and Overhead Parameters

The parameter *ServerDisks* specifies the number of database disks attached to the server. The model assumes that the database is uniformly partitioned across all server disks. The disk used to service a given I/O request at the server is thus chosen at random (uniformly) from among the server disks. The parameter *ClientDisks*

[2]The performance and complexity implications of relaxing this policy are considered in Section 7.4.1.

specifies the number of disks (if any) attached to each client. Chapters 4 and 6 use models in which the clients are diskless, while Chapter 7 investigates the use of client disks (one per client) for extending a client's memory cache. The model currently allows at most one disk to be attached to each client. In the cases where a client disk is used as an extension of the memory cache, the amount of space allocated for caching on disk is specified as a percentage of the database size using the parameter *CliDiskCacheSize*. The disks are modeled as having an access time that is uniformly distributed over the range from *MinDiskTime* to *MaxDiskTime*. The service discipline for each of the disks is FIFO, and a CPU charge of *DiskOverheadInst* instructions is incurred for each I/O request.

It should be noted that the model does not explicitly include a logging component, as logging is not expected to impact the relative performance of the algorithms being studied. There is no need to access the log in the experiments reported here for the following two reasons: First, the experiments involve parameter settings where all of the pages dirtied by a transaction can fit in the client's buffer pool. Because of this, together with the fact that updated pages are copied back to the server at commit-time, transaction aborts can be performed without needing to access an undo log by simply purging the pages dirtied by an aborting transaction from the client's buffer pool. Secondly, crashes are not modeled, as recovery time is not a focus of the performance study. Recovery concerns are reflected in the design of all of the algorithms studied, however. For example, this is the reason for copying back dirty pages to the server at commit time. All of the algorithms studied in this thesis are designed to work in conjunction with a Write Ahead Logging protocol such as the ARIES-based recovery system used in the page server EXODUS storage manager [Fran92c].

A very simple network model is used in the simulator's *Network Manager* component; the network is modeled as a FIFO server with a service rate of *NetworkBandwidth*. Thus, the model does not attempt to capture the details of the operation of a specific type of network (e.g., Ethernet, token ring, etc.). Rather, the approach taken in the model is to separate the CPU costs of messages from their on-the-wire costs, and to allow the on-the-wire costs of messages to be adjusted using the bandwidth parameter. A simple model is sufficient for the experimental purposes in this thesis because the target environment is a local area network, where the actual time-on-the-wire is negligible. The important performance factor, with regards to the network, is its capacity relative to the speeds of the other resources in the system and the demands placed on it by the workload. Using the bandwidth parameter, the network capacity can be varied relative to the other factors. In most cases, the main cost issue for messages is the CPU time for sending and receiving them. The CPU cost for managing the protocol for a message send or receive is modeled as *FixedMsgInst* instructions per message plus *PerByteMsgInst* instructions per message byte.

Finally, the physical resource model allows the specification of several other resource-related parameters. The size of a control message (such as a lock request or a commit protocol packet) is given by the parameter *ControlMsgSize*; messages that contain one or more data pages are sized based on Table 3.2's *PageSize* parameter. Other cost parameters of the model include *LockInst*, the cost involved in a lock/unlock pair on the client or server, and *RegisterCopyInst*, the cost (on the server) to register and unregister (i.e., to track the existence of) a newly cached page copy or to look up the copy sites for a given page. Finally, for algorithms that require global deadlock detection (as do some of the algorithms proposed in Chapter 4), the parameter *DeadlockInterval* determines how often global deadlock detection is initiated.

3.3 Workload Models

Table 3.2 presents the parameters used to model the system workloads. The system workload is generated by the client workstations present in the system. Each client workstation generates a single stream of transactions, where the arrival of a new transaction is separated from the completion of the previous transaction by an exponentially distributed think time with a mean of *ThinkTime*. The database is modeled as a collection of fixed-size pages. No distinction is made between data and index pages. A client transaction reads between *0.5* $\times$ *TransSize* and *1.5* $\times$ *TransSize* distinct pages from the database. It spends an average of *PerPageInst* CPU instructions processing each page that it reads and this amount is doubled for pages that it writes; the actual per-page CPU requirements are drawn from an exponential distribution.

Parameter	Meaning
ThinkTime	Mean think time between client transactions
TransSize	Mean number of pages accessed per transaction
PerPageInst	Mean no. of CPU instructions per page read (doubled on write)
HotBounds	Page bounds of *hot* range
ColdBounds	Page bounds of *cold* range
HotAccessProb	Probability of accessing a page in the *hot* range
HotWriteProb	Probability of writing to a page in the *hot* range
ColdWriteProb	Probability of writing to a page in the *cold* range

Table 3.2: Per Client Workload Parameters

An important feature of the model is its method for defining the page access patterns of workloads, which allows different types of locality at clients and data-sharing among clients to be easily specified. The workload is specified on a per client basis. For each client, two (possibly overlapping) regions of the database can be specified. These ranges are specified by the *HotBounds* and *ColdBounds* parameters. The parameter *HotAccessProb*

specifies the probability that a given page access will be to a page in the client's "hot" region, with the remainder of accesses being to pages in its "cold" region. Within each region, pages are chosen without replacement using a uniform distribution. The *HotWriteProb* and *ColdWriteProb* parameters specify the region-specific probabilities of writing a page that has been accessed. This workload model is fairly simple, but it allows the specification of workloads with a wide range of data access, sharing, and conflict characteristics. Such workloads can be used to examine the impact of these different factors on the performance of caching algorithms. The use of the workload model is demonstrated by the two example workloads shown in Table 3.3.

Parameter	HOTCOLD	FEED
TransSize	20 pages	5 pages
HotBounds	p to p + 49 p = 50(n-1) + 1	1 to 50
ColdBounds	remainder of database	remainder of database
HotAccessProb	0.8	0.8
HotWriteProb	0.2	1.0 (if (n) = *1), 0.0 (otherwise)*
ColdWriteProb	0.2	0.0

Table 3.3: Example Workload Parameter Settings for Client n

The HOTCOLD workload has high locality at each client and moderate read/write sharing among the clients. In HOTCOLD, each client has a 50 page range from which 80% of its accesses are drawn. The remaining 20% of the accesses are distributed uniformly among the remaining pages in the database. As is shown in Figure 3.2, the hot ranges of the clients do not overlap. Therefore, there is substantial affinity of pages to clients. However, the hot range pages of each client are also in the cold ranges of all of the other clients, so there is sharing of data among the clients. Because all pages are accessed with a 20% write probability, some of this sharing is read/write sharing, which will generate data conflicts. Thus, HOTCOLD can be used to examine the performance of caching algorithms in terms of locality and conflict resolution.

The other example workload, called FEED (shown in Figure 3.3), is designed to model an environment in which there is directional "information flow" from one client to the others. For example, a stock quotation system where updates are performed at a main site while the rest of the sites are consumers of the data would exhibit such directional sharing. As shown in Table 3.3, a 50 page region is updated by the first client and read by the remaining clients, while the rest of the database is shared in a read-only mode. In addition to information flow, this workload exhibits a high degree of locality and substantial read-write contention on the hot pages.

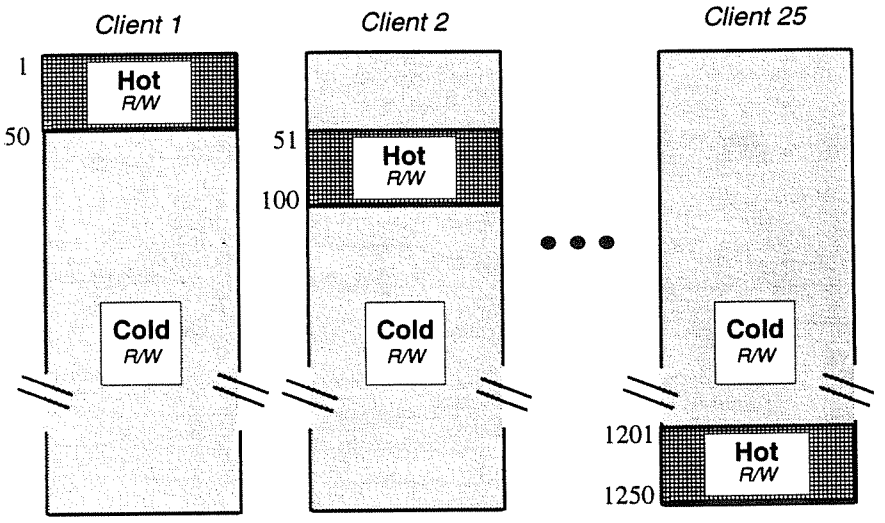


Figure 3.2: HOTCOLD Workload

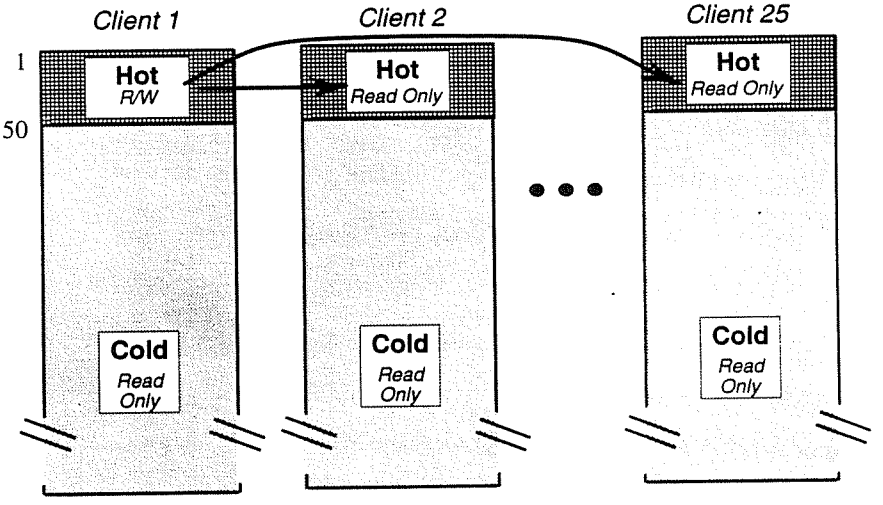


Figure 3.3: FEED Workload

3.4 Experimental Methodology

The simulation of a complex, distributed system such as a page server DBMS is a large undertaking, with many degrees of freedom. Simulation is a powerful technique; using simulation, one can obtain a deep understanding of system performance on configurations that do not yet exist or that are nearly impossible to manage in a controlled fashion (e.g., 50 dedicated workstations executing a carefully controlled workload). Moreover, a simulation can provide detailed statistics without impacting the performance measurements of the simulated system, and can therefore be used to gather information that would be impossible to obtain from an actual system (e.g., an instantaneous analysis of distributed buffer contents to determine types of sharing). Finally, simulations can be used to isolate specific aspects of system performance in order to gain insight that could not be obtained from an actual implementation. Of course, the disadvantage of simulation is that simplifications must be made in order to make the construction and execution of the models tractable. These simplifications impact the way in which the results of the simulations should be interpreted and utilized.

As should be clear from the preceding subsections, there are a vast number of knobs that can be turned in the page server DBMS model. Indeed, many of those knobs have been turned during the experimentation process. These knobs include various physical aspects of the system (e.g., CPU, disk, and network speeds), DBMS implementation issues (e.g., page size, deadlock detection interval, etc.), workload characteristics (locality, transaction length, etc.), and the algorithms under study. The physical parameter settings used throughout the thesis have been chosen to represent current and/or near-term workstation-server environments. Experiments have been performed with different CPU speeds, network bandwidths, etc. in order to run the systems in different performance regions (e.g., CPU limited vs. disk limited). The DBMS implementation parameter settings have been chosen based on knowledge of and experience with systems such as EXODUS and on the results of previous performance studies (e.g., [Care91b]).

Much of the effort of producing the experimental designs used in the performance studies that follow has gone into the development of a suite of workloads. At present, there is no widely agreed upon characterization of OODBMS workloads. Therefore, the approach taken in this thesis was to develop symbolic workloads, each emphasizing different application characteristics. The workloads used here vary the locality of access at clients, the level of data sharing and conflict among clients, and the properties of information flow among clients. None of the workloads studied have been drawn from actual OODBMS applications, as traces of production applications at the level of page accesses were not available. However, many of the abstract workloads have been developed with input gathered from several developers of commercial OODBMSs. Some of the workloads have

been designed to model the characteristics of certain application domains (e.g., a stock quotation environment, as in the FEED workload described in Section 3.3), while others are intentionally synthetic (e.g., a workload in which database pages are accessed with uniform probability) in order to aid in exploring the performance of the various algorithms under extreme conditions.

The choice of parameter settings is only one of the areas in which care must be taken in order to produce a meaningful performance study. In some ways, a simulator is harder to debug than an actual implementation because algorithmic bugs that would cause a crash in a real system may only affect the performance results produced by the simulation. As a result, much care has been taken in debugging the implementation of the simulator. Programming techniques, such as the use of assertions, have been employed to help ensure that the algorithms are operating properly. Also, the simulator has been designed to periodically pause during a simulation run and perform audits of the consistency of its internal data structures. Execution traces of small configurations were also examined to see that execution proceeded as expected. Of course, it is inevitable that some bugs escape such measures, and thus, a substantial effort has also been put into verifying the conclusions that have been drawn from the simulation results. First, confidence intervals are computed for all response time results in order to minimize the impact of transient factors such as buffer warm up. More importantly, once initial conclusions are drawn, evidence must be found to support them. The process of gathering evidence involves the examination of other statistical measurements from the simulation, and often, the installation of new probes and the re-running of experiments. This last step is crucial, as it is surprisingly easy to develop plausible (and interesting) explanations for phenomena, which after further investigation are turn out to be flawed.

Performance analysis of any reasonably complex system, real or simulated, is still an art because (among other things): 1) only a small region of the performance space can be explored, 2) it is often difficult to determine the cause of the observed performance behavior, and 3) it is difficult to generalize the results of specific tests for use in predicting performance under other conditions. The results that are presented in the following chapters are a small portion of the results that have been obtained. Because of the simplifications made in the simulation, these results should be interpreted *qualitatively*. In other words, the results are not likely to be accurate predictions of the actual throughput that would be obtained under a given configuration. Rather, the value of the experimental work is the comparison of the relative performance of the algorithms under a range of different workloads and regions of system operation. The simulations help uncover fundamental tradeoffs among the techniques studied, such as the effects of sharing on the contents of buffers or the relative communication requirements of different algorithms. In some cases, the results establish the dominance of certain techniques for a wide range of situations. In other cases, unfortunately, the relative merits of various approaches are dependent on

workload or configuration aspects. In these cases, the performance results are used to demonstrate regions in which the tradeoffs occur. This tradeoff analysis can be used to determine the best approach given the specific characteristics of a particular application environment, or it can be used (as is shown in Chapter 4) as the basis for choosing heuristics for the design of adaptive algorithms.

Chapter 4

Client Cache Consistency

4.1 The Consistency Maintenance Problem

As discussed in the preceding chapters, client data caching is a fundamental technique for obtaining high performance and scalability in a page server DBMS. Caching allows the DBMS to use the memory of client workstations in order to reduce the number of data requests that must be sent to the server. The effectiveness of caching depends on the assumption that there is significant *locality of access* in the system workload. Locality can be considered along two dimensions (e.g., [Henn90]):

- *Temporal Locality* - References to items are clustered in time. If an item is accessed, it is likely to be accessed again in the near future.
- *Spatial Locality* - References to items are clustered in space. If an item is accessed, it is likely that items that are physically near it (e.g., on the same data page) will be referenced in the near future.

Caching in a page server DBMS exploits locality along both of these dimensions. Temporal locality is exploited by keeping recently referenced items at clients on the assumption that they are likely to be needed at that client again. Caching at a page granularity exploits spatial locality on the assumption that if an object is accessed at a client, then it is likely that other objects on that same page will also be referenced at the client.

Because data items are allowed to be cached at multiple sites, a mechanism for insuring that all sites see a consistent view of the database must be used. This is referred to as the *cache consistency maintenance* problem. As stated in Section 2.3.2, the model of correctness provided by a page server DBMS is based on the ACID transaction properties, which require *serializable* execution of transactions. The extension of the notion of serializability to replicated data is called *one-copy serializability* [Bern87]. In one-copy serializability,

the presence of replicated data, if any, is transparent to the users of the system. Thus, a one-copy serializable execution of transactions on a replicated database is equivalent to some *serial* execution of those transactions on a non-replicated database.

Protocols to maintain cache consistency in a page server DBMS are necessarily related to the concurrency control and transaction management schemes used in the system. Data pages can be cached both within a single transaction (*intra-transaction caching*) and across transaction boundaries (*inter-transaction caching*). Intra-transaction caching is easily implemented; it requires only that a client can manage its own buffer pool (i.e., local cache) and can keep track of the locks that it has obtained. Due to this simplicity, all of the algorithms investigated in this thesis perform intra-transaction caching. On the other hand, inter-transaction caching requires additional mechanisms to ensure the consistency of cached data pages, possibly including full-function lock management on clients and protocols for distributed lock management. This chapter concentrates on the performance tradeoffs of inter-transaction caching.[1]

As discussed in Section 2.3.3, inter-transaction caching has the potential to provide substantial performance and scalability benefits. However, while the advantages of client data caching may be large, there are of course, potential pitfalls as well. In particular, it is unavoidable that cache consistency mechanisms will require overhead in terms of communication and processing (and also perhaps, I/O, as is shown in Chapter 5) , possibly even for clients at which the database is not currently in use. Furthermore, some consistency maintenance algorithms postpone the discovery of data conflicts, which can result in increased transaction abort rates.

The design space of cache consistency maintenance protocols is examined in the next section, which presents a taxonomy such protocols. This taxonomy can be used to help understand the design alternatives and performance tradeoffs for transactional cache consistency maintenance algorithms. Following the taxonomy, a description of the cache consistency algorithms that are analyzed in this thesis is presented, along with an overview of the algorithms that have been proposed elsewhere in the literature. The performance of the algorithms is then analyzed in detail in Chapter 5.

4.2 A Taxonomy of Consistency Protocols

As stated in the previous section, cache consistency protocols for page server DBMSs combine aspects of distributed concurrency control, transaction management, and replicated data management. As a result, there is a wide range of options for the design of such protocols. This section provides a taxonomy of cache consistency

[1] For the remainder of the thesis, the term "caching" will be used to refer to inter-transaction caching at client workstations, unless explicitly stated otherwise.

protocols that encompasses the algorithms that are studied in this thesis [Care91a, Fran92a] as well as other related algorithms proposed in the literature [Wilk90, Wang91, Lisk92]. The taxonomy is restricted to algorithms that are applicable to page server DBMSs (although some of the algorithms were originally proposed for object servers) and that provide one-copy serializability. Algorithms that provide lower levels of consistency are not considered here.

There are many possible ways to slice the design space for cache consistency protocols. One possibility is to organize it along the functional lines of concurrency control and replicated data management, as is done for algorithms in the shared disks environment in [Rahm91]. This approach is problematic, however, because the two concepts are so closely intertwined. Dividing a taxonomy at the highest level along these lines can result in substantial duplication of mechanism within the taxonomy, hurting its descriptive effectiveness. Serializability theory for non-replicated data can be extended to handle replicated data (i.e., one-copy serializability) without the creation of a parallel set of mechanisms [Bern87]. Therefore, it is natural that cache consistency maintenance protocols can be designed by extending concurrency control concepts rather than by duplicating them. Another problem that arises when concurrency control is considered to be a separate structural component of the taxonomy is that certain protocols become difficult to classify (e.g., as "locking" or "optimistic"). For example, eight of the thirteen protocols that are covered in this section use combinations of both locking and optimism that interact in complex ways.

In the taxonomy presented here, cache consistency protocols are considered as integrated entities in their own right. The design space is divided along the functional lines of the integrated algorithms. As a result, protocols that use combinations of locking and optimism (e.g., the O2PL algorithms studied in this thesis) can be accommodated in the taxonomy. Also, protocols that use locking and those that use optimistic techniques are sometimes co-located in the same sub-tree of the taxonomy, The taxonomy is shown in Figures 4.1 and 4.2. At the top level of the taxonomy, algorithms are classified according to the type of *replica update protocol* that they employ. This is a fundamental consideration, as it determines whether or not the clients can trust the validity of their cache contents. Because this is a major distinction among protocols, the properties upon which the lower levels of the taxonomy are based differ depending on the replica update protocol used (as can be seen by comparing Figures 4.1 and 4.2). The next section discusses the two main replica update options that serve as the basis for the taxonomy. The lower levels of the taxonomy for each of the replica update options are then described in the sections that follow.

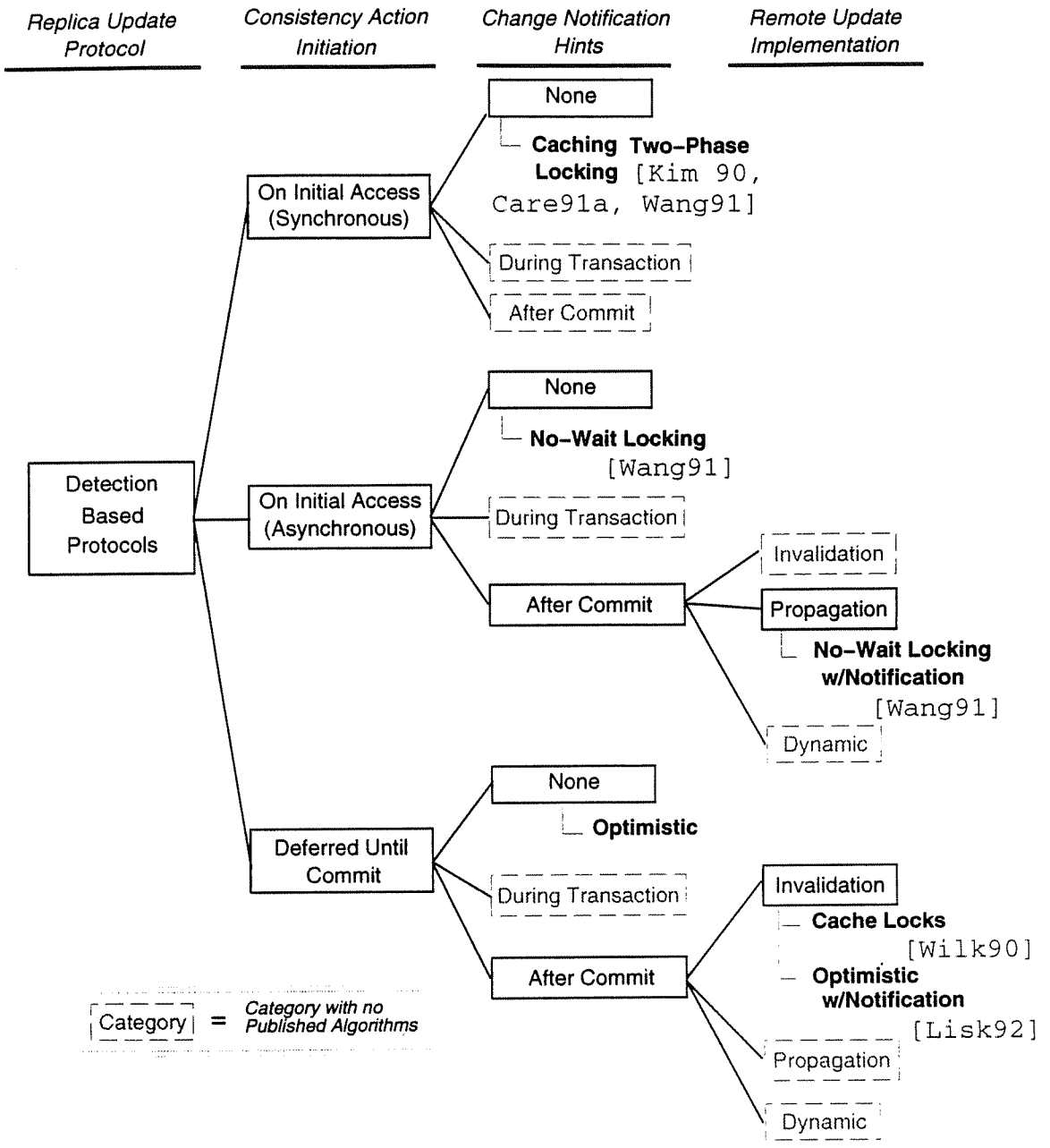


Figure 4.1: Taxonomy of Detection-Based Protocols

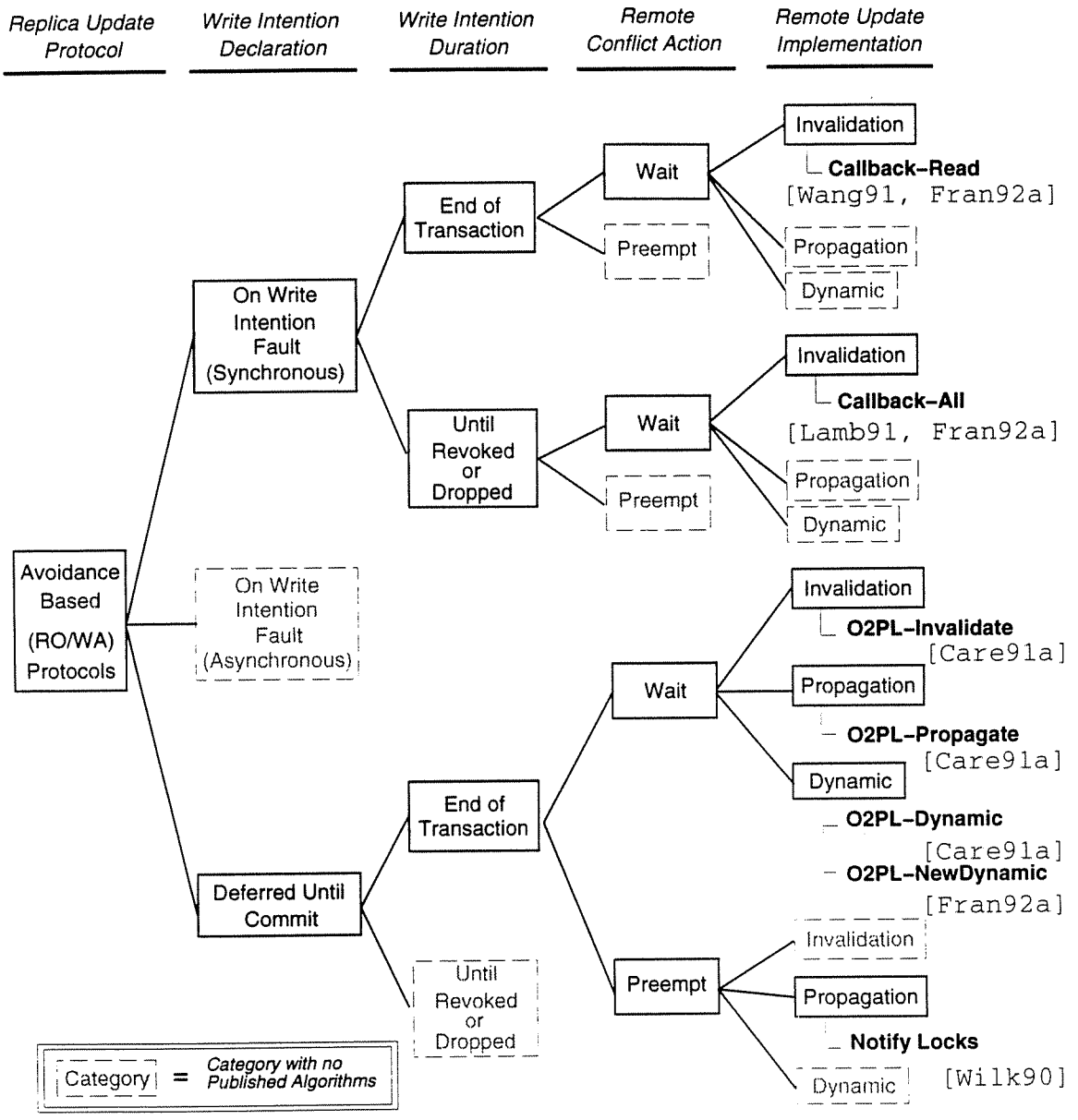


Figure 4.2: Taxonomy of Avoidance-Based Protocols

4.2.1 Replica Update Protocol

Because cached data is replicated data, it follows that traditional methods for managing updates to replicated data can be used or extended to manage cached copies. Cache consistency maintenance protocols can thus be classified based on the approach that they use to determine the replicas to which consistency actions must be applied as the result of a particular update. The protocols that have been proposed for page server DBMSs can be partitioned into two classes based on this criterion: *detection-based* and *avoidance-based.*

From a qualitative point of view, the difference between the detection-based and avoidance-based approaches lies in how access to *stale* (i.e., out-of-date) data is prevented. Because a page server DBMS preserves transaction semantics, the notion of data being stale is tied to the execution of transactions. Specifically, a data item is considered to be stale if its value is older than the item's latest *committed* value.[2] Consider a system in which data items are tagged with sequence numbers, and where the sequence number of an item is incremented when a transaction that has updated the item commits (commit is an atomic operation, so sequence numbers are monotonically increasing). A copy of a data item is then considered to be stale if its sequence number is lower than the latest sequence number assigned to any copy of the data item.

The detection-based schemes allow stale data copies to reside in a client's cache. As a result the transactions must check the validity of any cached page that they access before they can be allowed to commit. The server is responsible for maintaining information that will enable clients to perform this validity checking. The detection-based schemes are so named because the presence of stale data in a client's cache is explicitly checked for and detected. Transactions that have accessed stale data are not allowed to commit.

In contrast, under the avoidance-based protocols, transactions never have the opportunity to access stale data. Avoidance-based protocols are based on a read one/write all (ROWA) approach to replica management. A ROWA protocol ensures that all existing copies of an updated item have the same value when a transaction commits. The avoidance-based protocols can thus be said to *avoid* access to stale data by making such access impossible. In a ROWA scheme, a transaction is allowed to read *any* copy of a data item (this will typically be the one in its client's cache, if such a copy exists). Updates, however, must be reflected at *all* of the copies existing in the system before the transaction can be allowed to commit. The ROWA protocol used in a page server DBMS is a specialized one however, as it is managed with help from (or by, in some cases) the server. As described in Section 2.3.3, the use of second-class replication allows the server to eliminate unreachable copies from the protocol so that transaction processing can continue. As a result, the problem of network partitions

[2] It is important to note that values become stale only as the result of a transaction commit. Some of the avoidance-based approaches allow multiple transactions to simultaneously access different values of the same page, provided that serializability is not violated.

[Davi85], which can seriously impact the availability of data under a traditional ROWA protocol, is not as critical in a page server DBMS.

The main argument for the detection-based approach is simplicity. Because consistency actions involve only a single client and the server, the cache management software on the clients can be greatly simplified compared to the ROWA approach. For example, using detection, the replica validation protocol is initiated by clients sending requests to servers. As a result, the system software can be structured such that clients do not have to be prepared to receive asynchronous messages from the server. The current EXODUS storage manager implementation [Exod93] uses a detection-based approach largely for this reason. In EXODUS, the use of detection allows the client software to be linked with the application code as a library, rather than having to be a separate UNIX process. Also, by using detection the need for certain distributed protocols can be obviated. For example, distributed global deadlock detection is not needed for detection-based algorithms that use locking because the servers have all of the information that is necessary to perform deadlock detection. The disadvantage of the detection-based approaches, however, is a greater dependency on the server. This can result in significant performance overhead, such as increased communication.

Before proceeding, it should be noted that detection and avoidance of stale data are not mutually exclusive techniques. In fact, three of the six detection-based algorithms shown in Figure 4.1 augment detection with asynchronous update notifications (i.e., installing new values or removing stale values at remote clients) in order to reduce the probability of having stale data in the client caches.[3] As is discussed in the next subsection, lowering this probability can improve performance by reducing the need to abort transactions under certain detection-based consistency protocols. These three algorithms lie strictly in the detection-based camp, however, as the notifications are sent only as "hints". All three ultimately depend on detection to ensure that committing transactions have not accessed any stale data.

4.2.2 Detection-based Protocols

A number of detection-based protocols (shown in Figure 4.1) have been proposed and studied in the literature [Kim90, Wilk90, Care91a, Wang91, Lisk92]. Because of their centralized nature, detection-based consistency maintenance protocols can often be constructed by extending an existing centralized (non-replicated) concurrency control protocol in a fairly straightforward manner. In fact, most of the protocols that appear in this part of the taxonomy are natural extensions of traditional two-phase locking or optimistic concurrency control algorithms. There are three levels of differentiation in the detection-based portion of the taxonomy: consistency action

[3] These algorithms have entries in the "Remote Update Implementation" column in Figure 4.1.

initiation, change notification hints, and remote update implementation.

Consistency Action Initiation

The coarsest level of differentiation for the detection-based taxonomy is based on the point (or points) during transaction execution, at which the validity of accessed data is checked. The detection-based protocols do not guarantee the consistency of data across transaction boundaries, and furthermore, the consistency of the data accessed by a transaction must be determined before that transaction can be allowed to commit. As a result, consistency checks for all data touched by a transaction must begin and complete during the execution of the transaction. In the taxonomy, there are three classes of consistency checking strategies:

- Synchronous, on each initial access to a page by a transaction.
- Asynchronous, on the initial access.
- Deferred, until a transaction enters its commit processing phase.

All three classes have the property that once the validity of a client's copy of a data item is established, that copy is guaranteed to remain valid for the duration of the transaction. To implement this guarantee, the server must exclude other transactions from committing updates to the validated items until the validating transaction finishes (commits or aborts). As a result, transactions must obtain permission from the server before they are allowed to commit the update of a data item.[4]

Synchronous checking is the simplest of the three classes. On the first access that a transaction makes to a particular data item, the client must check with the server to ensure that its copy of the item is valid. This is done in a synchronous manner — the transaction is not allowed to access the item until its validity has been verified. Once the validity of the client's copy of the item has been established (which may involve sending the client a new, valid copy) the copy is guaranteed to remain valid at least until the transaction completes. Asynchronous checking is similar, but the transaction does not wait for the result of the check. Rather, it proceeds to access (or write) the copy under the assumption that the check will succeed. If this optimism turns out to be unfounded, then the transaction must abort. Finally, deferred checking is even more optimistic than asynchronous checking. No consistency actions are sent to the server until the transaction has completed its execution phase and has entered its commit phase. At this point, information on all the data items read and written by the transaction is sent to the server, and the server determines whether or not the transaction should be allowed to commit.

[4] Although it is not strictly necessary, all of the protocols shown in Figure 4.1 use the same initiation method for update permission requests as they do for validation requests. If this were not the case, validation and update requests would have to be separate dimensions in the taxonomy.

These three classes provide a range from pessimistic (synchronous) to optimistic (deferred) techniques. Therefore, they represent different tradeoffs between checking overhead and aborts. Deferring consistency actions can have two advantages. First, and most importantly, consistency actions can be bundled together in order to reduce consistency maintenance overhead. Secondly, any consistency maintenance work performed for a transaction that ultimately aborts is wasted; deferred consistency actions can avoid some of this work. There are also, however, potential disadvantages to deferring consistency actions. The main disadvantage is that deferral can result in the late detection of data conflicts. The resolution of conflicts that are detected after they have occurred typically requires aborting one or more transactions. Such aborts can hurt performance — particularly for conflicts that could have been resolved by blocking if they had been detected when they first arose. Furthermore, in the highly interactive OODBMS environments for which page servers are intended, aborts are undesirable as they can result in significant lost work for users. Policies that have high abort rates may therefore be inappropriate for such environments. The asynchronous approach is a compromise; it aims to mitigate the cost of interaction with the server by removing it from the critical path of transaction execution, while lowering the abort rate and reducing wasted work through the earlier discovery of conflicts.

Change Notification Hints

The emphasis on optimistic (i.e., asynchronous and deferred) techniques found in the literature on detection-based protocols is an artifact of the particular cost tradeoffs of the page server environment. Because communication with the server is an inherently expensive operation, designers of detection-based protocols are forced to rely upon optimism in an attempt to reduce this cost. Optimistic techniques are oriented towards environments in which conflicts are rare and the cost of detecting conflicts is high, and can provide good performance in such situations. While there is currently no definitive understanding of page server DBMS workloads, it is generally assumed that page server DBMSs have lower levels of conflict than other DBMS environments, such as transaction processing [Catt91b]. In a caching environment, however, the notion of conflict must include *sequential sharing* in addition to simultaneous data sharing. Sequential sharing arises when transactions that *do not run concurrently* access the same data. Because caching strives to keep data at a site even after a transaction has completed, the cache consistency maintenance protocol must also deal effectively with this type of sharing. Recent studies of file system workloads [Rama92, Sand92] indicate that sequential sharing may, in fact, be quite common in the types of situations in which page servers are intended to be used. If this is the case, then the naive use of optimistic techniques may result in unacceptably high abort rates.

Two approaches for reducing the potential for aborts in optimistic techniques have been put forward by the

designers of such algorithms. One is to simply treat "hot spot" data differently, e.g., by switching to a more pessimistic protocol for such data. The details of such adaptive algorithms have not yet been published, however. The other approach is to adopt techniques from the avoidance-based (read one/write all) protocols to reduce the amount of stale data that resides in client caches. Such techniques are called *change notification hints*.

As can be seen in Figure 4.1, three of the six protocols found in the literature use some form of change notification hints (or simply, "notifications"). A notification is an action that is sent to a remote client as the result of an update that could impact the validity of an item cached at the client. Removing (or updating) a stale copy reduces the risk that a transaction will be forced to abort as a result of accessing that copy. For the detection-based protocols, notifications are simply "hints", since the correctness of the protocol is ultimately maintained by detection. As a result, notifications are sent asynchronously in these protocols.

Notifications can be sent anytime during the execution of an updating transaction, or after such a transaction commits. Sending notifications before commit can be dangerous, however, if the notifications actually update the remote copies rather than simply remove them. If the transaction on whose behalf the notification was sent eventually aborts, then the remote updates will have to be undone, adding significant complexity and expense to the protocol. Notifications that simply purge copies from remote caches are less problematic. Still, they too can cause unnecessary aborts at remote sites if active transactions have accessed the invalidated copies there. Because of these complexities, all three of the protocols shown in Figure 4.1 that use notification [Wilk90, Wang91, Lisk92] do so only after the updating transaction has committed.

Remote Update Implementation

The final level in the taxonomy for detection-based protocols is concerned with the action taken when a notification arrives at a remote site. There are three options here: propagation, invalidation, and choosing dynamically between the two. Propagation results in the newly updated value being installed at the remote site in place of the stale copy. Invalidation, on the other hand, simply removes the stale copy from the remote cache, so it will not be accessed by any subsequent transactions. After a page copy is invalidated at a site, a subsequent transaction that wishes to access the page at that site must obtain a new copy from the server. A dynamic protocol can chose between invalidation and propagation heuristically in order to optimize performance for varying workloads. The concepts of propagation and invalidation are analogous in some sense to the notions of write-broadcast and write-invalidate (respectively) used in multiprocessor cache consistency algorithms [Arch86]. The tradeoffs involved differ greatly between multiprocessors and page servers, however. The investigation of the tradeoffs among these options for page servers is one of the main subjects of the performance study presented in Chapter 5.

4.2.3 Avoidance-based Protocols

Avoidance-based protocols, which are based on the read one/write all (ROWA) replica management paradigm, are the other major class of transactional cache consistency maintenance protocols. The taxonomy for avoidance-based protocols is shown in Figure 4.2. As stated previously, avoidance-based protocols enforce consistency by making it impossible for transactions to ever access stale data in their local cache. They accomplish this by directly manipulating the contents of remote client caches in response to (or prior to) client updates. Because consistency actions manipulate page copies in remote client caches, the clients must have additional mechanism to support these actions (e.g., some schemes require that clients have a full function lock manager). While this additional responsibility increases the complexity of the client software somewhat, the effect is to reduce reliance on the server and thereby (hopefully) offload shared resources. This is in keeping with the principle of utilizing client resources outlined in Section 2.3.1, and thus, much of the work in the following chapters is geared towards avoidance-based protocols.

In addition to their requirement for support at the clients, avoidance-based protocols also require extra information to be maintained at the server. Specifically, all of the avoidance-based protocols described here require that the server keep track of the location of all page copies. In order to satisfy the "write all" requirement of the ROWA paradigm, it must be possible to locate all copies of a given page. One way to accomplish this is through the use of broadcast, as in snooping protocols for small multiprocessor caching [Good83]. Reliance on broadcast, however, is not a viable option in a page server DBMS due to cost and scalability issues. As a result, a "directory-based" approach must be used. As discussed in Section 2.3.3, the server is the focal point for all transaction management functions and is responsible for providing clients with requested data; as a result, all of the protocols in this part of the taxonomy maintain the directory at the server.

There are four levels of differentiation in the avoidance-based portion of the taxonomy: write intention declaration, write intention duration, remote conflict action, and remote update implementation. As is discussed in the following sections, two of these dimensions, namely, write intention declaration and remote update implementation, are similar to dimensions that appear in the detection-based portion of the taxonomy.

Write Intention Declaration

As with the detection-based protocols, the avoidance-based protocols can be categorized according to the time at which transactions initiate consistency actions. The nature of the consistency actions, however, is somewhat different than those used by the detection-based schemes. Because of the ROWA protocol, transactions executing

under an avoidance-based scheme can always read any page copy that is cached at their local client. Thus, interaction with the server is required only for access to pages that are not cached locally and for updates to cached pages. Interactions with the server to obtain copies of non-cached pages must, of course, be done synchronously. On a cache miss, the client must request the page from the server. When the server responds with a copy of the page, it also implicitly gives the client a guarantee that the client will be informed if another client performs an operation that would cause the copy to become invalid.

While all of the protocols use the same policy for handling page reads, they differ in the manner in which consistency actions for *updates* are initiated. When a transaction wishes to update a cached page copy, the server must be informed of this intention prior to transaction commit, so that it can implement the ROWA protocol. As discussed in the following subsection, write intentions can be obtained on behalf of a particular transaction or can be obtained by a client (for all transactions that execute at the client). When the server acknowledges to a client that a write intention has been registered, it guarantees that the client can update the page without again having to ask the server for permission until the client gives up its intention. A client may give up its intention to write a page on its own accord or as the result of a request from the server.

A *write intention fault* is said to occur when a transaction attempts to update a page copy for which the server does not currently have a write intention registered. The taxonomy contains three options for when clients must declare their intention to write a page to the server:[5]

- Synchronous, on a write intention fault.
- Asynchronous, on a write intention fault.
- Deferred, until an updating transaction enters its commit processing phase.

In the first two options, clients contact the server when they wish to update a page for which the server does not currently have a write intention registered. As in the detection-based case, such requests can be performed synchronously (i.e., the transaction must wait for the server to acknowledge the registration of the write intention), or asynchronously (i.e., the transaction does not wait). In the third option, declarations of write intentions are deferred until the transaction finishes its execution phase.

The tradeoffs among synchrony, asynchrony and deferral are similar in spirit to those previously discussed for the detection-based protocols: synchronous protocols are pessimistic, deferred ones are optimistic, and asynchronous ones are a compromise between the two. The magnitude of the tradeoffs, however, are quite

[5]These options are analogous to the "consistency action initiation" options for detection-based schemes discussed previously.

different for the avoidance-based protocols. The global nature of the protocols implies that that consistency actions may be required at remote clients before the server can register a write intention for a client (or transaction). Therefore, consistency actions can involve a substantial amount of work. The relatively high cost of consistency maintenance actions tends to strengthen the case for deferral of such actions.

It is also important to note that with avoidance-based protocols, the remote consistency operations are in the commit path of transactions. That is, a transaction cannot commit until it is verified that all of the necessary consistency operations have been successfully completed at the remote clients. As is described in the algorithm descriptions in Section 4.3, this restriction may require coordination protocols such as two-phase commit to be performed as part of consistency operations. One other implication of the difference between the consistency actions here and those in the detection-based case is that the actions here must be done with more care, as they can change the contents of remote client caches[6]. For example, if the result of a consistency action is to install a new value in a client's cache, then the remote updates will have to be undone (or removed), if the transaction that created the value eventually aborts.

Write Intention Duration

In addition to *when* write intentions are declared, avoidance-based algorithms can also be differentiated according to *how long* write intentions are retained for. There are two choices at this level of the taxonomy: write intentions can be declared for the duration of a particular transaction, or can span transactions at a given client. In the first case, transactions start with no write intentions so they must declare write intentions for all pages that they wish to update. At the end of the transaction, all write intentions are automatically revoked by the server. In the second case, write intentions can be retained at a client until the client chooses to drop the intention (e.g., when it drops its copy of a page). In either case, the server may request a client to drop its write intention on a page copy in conjunction with the performance of a consistency action,

Remote Conflict Action

The next level of differentiation is the priority given to consistency actions at the remote clients to which they are sent. There are two options here: wait and preempt. A *wait* policy states that consistency actions that conflict with the operation of an ongoing transaction at a client must wait for that transaction to complete. In contrast, under a *preempt* policy, ongoing transactions can be aborted as the result of an incoming consistency action. Under the preempt policy, the guarantees that are made to clients regarding the ability to read cached

[6] Recall that a similar problem arises for notification hints under the detection-based protocols.

page copies are somewhat weaker than under the wait policy. This is because the wait policy will force a remote writer to serialize behind a local reader if a conflict arises, whereas under the preempt policy, writers always have priority over readers, so conflicting readers are aborted. The majority of the avoidance-based protocols that have been proposed use the wait policy. The one exception is the "Notify Locks" algorithm of [Wilk90], in which transactions are aborted if any data that they have accessed (read or written) is modified by a concurrent transaction that enters its commit phase first.

Remote Update Implementation

As with the detection-based protocols, the final level of the taxonomy for the avoidance-based protocols is based on how remote updates are implemented. The options here are the same as in the detection-based case, namely: invalidation, propagation, and choosing dynamically between the two. As stated previously, the propagation of updates can be problematic if consistency actions are sent to remote sites during a transaction's execution phase. As a result, both of the algorithms in the taxonomy that send remote consistency actions during the execution phase rely on invalidation as the mechanism for implementing remote updates.

As stated previously, an important difference between remote update implementation under the avoidance-based protocols and under the detection-based ones is that in the avoidance-based case, the remote operations are initiated (and must be completed) on behalf of a transaction *before the transaction is allowed to commit.* This has implications for the choice of the remote update implementation. For example, if propagation is used, all remote sites that receive the propagation must participate in a two-phase commit with the server and the client at which the transaction is executing in order to ensure that all copies remain consistent. In contrast, invalidation does not require two-phase commit, as data is simply removed from the remote client caches.

4.3 Cache Consistency Maintenance Algorithms

As should be clear from the previous section, there are a large number of dimensions to be considered in the design of cache consistency algorithms for page server DBMSs. Furthermore these design decisions are not orthogonal, but rather, they interact in a number of complex ways. The cache management strategy is one of the fundamental aspects of a page server DBMS architecture. Therefore, an understanding of the design tradeoffs is a necessary input to the process of designing and implementing a high-performance page server DBMS. A significant part of the research performed for this thesis has been the detailed investigation of the performance tradeoffs of cache consistency maintenance algorithms. A range of algorithms, encompassing many of the dimensions discussed in

the previous section, have been studied. In this section, eight of those algorithms are described and motivated. A performance analysis of these algorithms is then presented in the chapter that follows. The algorithms fall into three families: Server-based two-phase locking (S2PL), Optimistic 2PL (O2PL), and Callback Locking.

4.3.1 Server-Based Two-Phase Locking (S2PL)

The server-based two-phase locking algorithms use detection-based replica update control. They perform consistency actions on the initial access to an object and are synchronous (i.e., they wait for the server to respond before proceeding with the access). The variants that are studied in this thesis do not send notification hints. This is because the implementation of notification hints removes one of the main advantages of detection-based schemes, namely, the fact that clients do not have to accept asynchronous messages from the server.

Server-based 2PL schemes are derived from the *primary copy* approach to replicated data management [Alsb76, Ston79]. In a primary copy algorithm, one copy of each data item is designated as "primary". Before a transaction is allowed to commit, it must first access the primary copy of all of the data items that it reads or writes. In a page server DBMS (with no server replication), the primary copy of all pages is the one that resides at the server. For reads, the client's copy of the page must be verified to have the same value as the server's copy. For writes, the new value created by the transaction must be installed as the new value of the primary copy in order for the transaction to commit.

Results for two server-based 2PL algorithms will be presented in the performance study that follows: *Basic 2PL* (B2PL) and *Caching 2PL* (C2PL). B2PL is not a caching algorithm — the client's buffer pool is purged upon transaction termination. Since every transaction starts with an empty buffer pool, there is no need to validate any data with the server. Rather, page requests are combined with lock requests. Clients obtain a read lock from the server upon the initial request for a page and may later request an upgrade to a write lock if that page is to be updated. Strict two-phase locking is used — all locks are held until the transaction commits or aborts. Deadlocks can arise and are handled through a centralized detection scheme at the server. Deadlocks are resolved by aborting the youngest transaction involved in the deadlock. B2PL is used in the study as a baseline in order to gauge the performance improvements that can be obtained through client caching. A similar algorithm was used as a baseline in [Wilk90].

The C2PL algorithm is a simple extension of B2PL that allows pages to be retained in client caches across transaction boundaries. Consistency is maintained using a "check-on-access" policy. All page copies in the

system are tagged with a sequence number which uniquely identifies the state of the page.[7] When a transaction initially accesses a page, it sends a request for a read lock on that page to the server (as in B2PL). The client includes the sequence number of its cached copy of the page (if the page is cache-resident) along with the lock request message. When the server grants a read lock to a client it also determines whether or not the client has an up-to-date cached copy of the requested page. If not, then the server piggybacks a valid copy of the page along with the lock response message returned to the client. In C2PL, deadlock detection is performed at the server in the same manner as for B2PL. C2PL is one of the least complex algorithms that supports inter-transaction caching. Because of this, algorithms similar to C2PL have been implemented in several systems. These include the ORION-1SX prototype [Kim90] and the EXODUS storage manager [Exod93]. An algorithm similar to C2PL has also been studied in [Wang91].

An example execution under C2PL is shown in Figure 4.3. The example uses two clients and a server. Client 1 runs a transaction that does the following:

1. Read page Y
2. Read page X
3. Write page X

Client 2 runs a transaction that performs the following operations:

1. Read page X
2. Read page Z

Each of the clients initially has valid cached copies of all the pages accessed by its transaction (as shown at the top of the figure). In the example, client 1 begins its transaction first. Since it has valid copies of all the necessary pages, it simply obtains locks from the server (messages 1-6) — the server does not need to send any new page copies to client 1. At the end of the transaction, client 1 sends a copy of the new value of page X back to the server (message 8) and commits (thereby releasing the locks held at the server). Prior to the commit of transaction 1, however, client 2 begins its transaction by trying to obtain a read lock on page X from the server (message 7). This request conflicts with the write lock held by transaction 1, so transaction 2 blocks. After transaction 1 commits, the read lock can be granted to transaction 2, but the copy of page X residing at client 2 is now stale, so the server includes a new copy of the page along with the lock grant reply (message 9). Client 2

[7] Data pages are typically tagged with such numbers, called Log Sequence Numbers (LSNs), in systems that use a Write-Ahead-Logging protocol [Gray93] for recovery (e.g., [Moha92]).

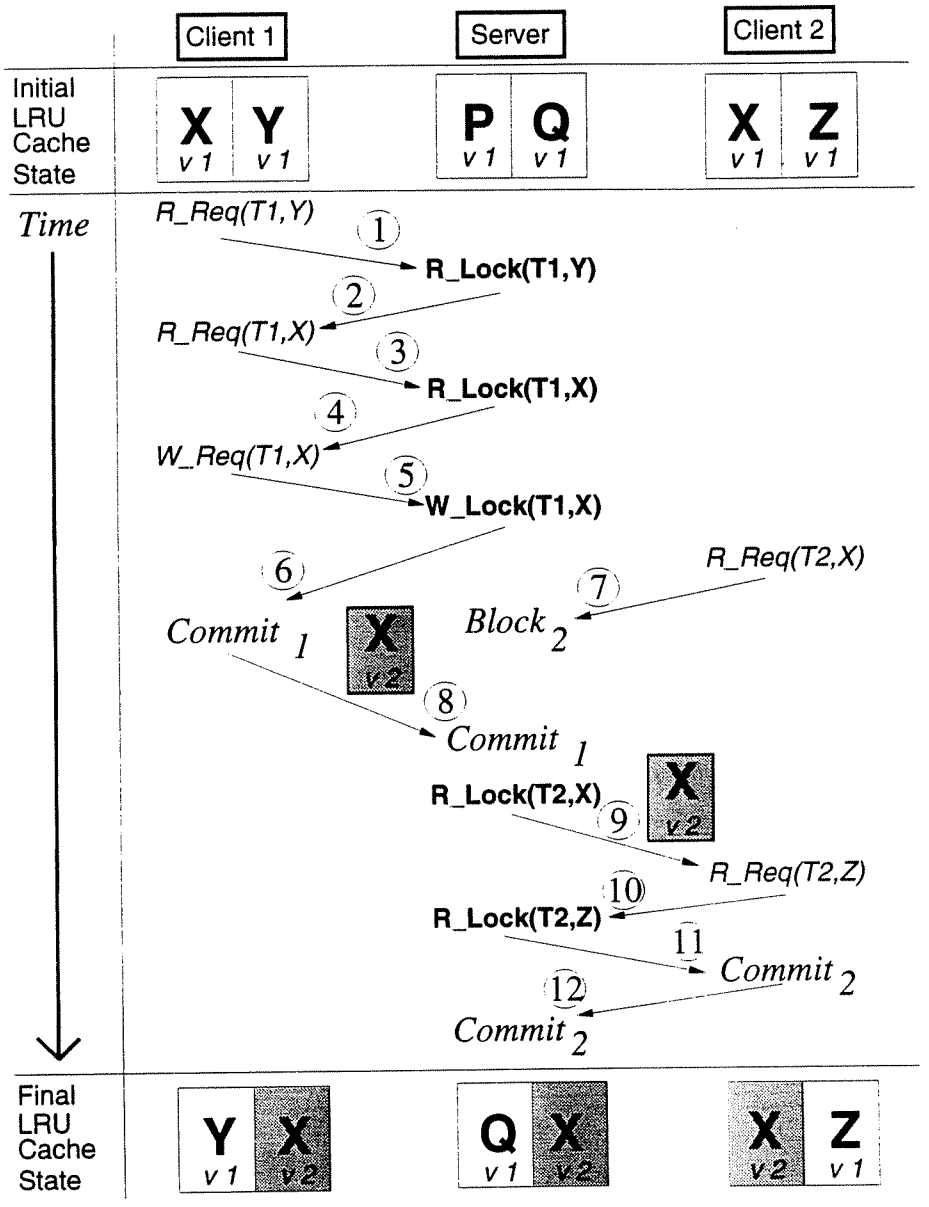


Figure 4.3: Example Execution Under C2PL

then continues its transaction, ultimately committing and releasing its locks at the server (messages 10-12). At the end of the execution both clients have copies of the latest version of page X in their local cache. However, this is only because transaction 2 accessed page X after it was written by transaction 1. If transaction 2 had not read the page, then a stale version would have remained in its cache. C2PL, being detection-based, does not prevent stale data from residing in client caches.

4.3.2 Optimistic Two-Phase Locking (O2PL)

Optimistic two-phase locking (O2PL) algorithms are derived from a concurrency control protocol for replicated distributed databases that was studied in [Care91b]. All of the O2PL algorithms support inter-transaction caching of pages at clients. The O2PL algorithms use an avoidance-based replica management protocol in which each client (in addition to the server) has a full-function lock manager. The O2PL algorithms are optimistic in the sense that they defer consistency actions for updates until the end of a transaction's execution phase.

During their execution phase, transactions acquire only local (to the client) read and write locks on their accessed pages. These local locks are held until the transaction completes. When a transaction needs to access a page that is not cache-resident, it sends a request for the page to the server. A short-term, non-two-phase read lock (i.e., latch) is obtained briefly at the server when a data item is in the process of being prepared for shipment to a client. The latch ensures that the client is given a transaction-consistent copy of the page. The server is responsible for keeping track of where pages are cached in the system. Clients inform the server when they drop a page from their buffer pool by piggybacking that information on the next message that they send to the server. Thus, the server's copy information is conservative, as there may be some delay before the server learns that a page is no longer cached at a client.

Transactions update pages in their local cache, and these updates are not allowed to migrate back to the server until the transaction enters its commit phase. When an updating transaction is ready to enter its commit phase, it sends a message to the server containing a copy of each page that it has updated. The server then acquires exclusive *update-copy* locks on its copies of these pages on behalf of the transaction. Update-copy locks are exclusive locks (similar to write locks) that allow certain deadlocks to be detected early. The update-copy locks obtained at the server allow it to safely install the new page values, as the locks are held until the transaction completes.

Once the required locks have been granted at the server, the server sends a message to each client that has cached copies of any of the updated pages. The remote clients obtain update-copy locks on their local copies of

the updated pages on behalf of the committing transaction.[8] Update-copy locks conflict with read locks, so an update transaction may have to wait for an active reader transactions to complete before it can continue commit processing. Once all of the required locks have been obtained at a remote site, the site performs variant-specific consistency actions (i.e., invalidation or propagation) on its copies of the updated pages (these are discussed in the following subsections).

With O2PL, each client maintains a local waits-for graph which is used to detect deadlocks that are local to the client. Global deadlocks are detected using a centralized algorithm *a la* [Ston79]. The server periodically requests local waits-for graphs from the clients and combines them to build a global waits-for graph. Deadlocks involving consistency actions can be detected early, due to the use of update-copy locks. When a conflict is detected between two update-copy locks or between an update-copy lock and a write lock, it is known that a deadlock has occurred (or will shortly occur), so the deadlock can be resolved immediately [Care91b]. As in the server-based case, deadlocks are resolved by aborting the youngest transaction.

Three O2PL variants are analyzed in the performance study in Chapter 5. These variants differ in the actions that they take once locks have been obtained on copies of updated pages in remote caches. O2PL-Invalidate (O2PL-I) removes copies of updated pages from remote caches. O2PL-Propagate (O2PL-P) instead sends the new copies of the updated pages to the remote caches so that they can continue to retain valid copies of the pages. O2PL-NewDynamic (O2PL-ND) uses a simple heuristic to choose between invalidation and propagation on a page-by-page basis. These variants are described below.

Invalidation

In the invalidation-based variant of O2PL (O2PL-I), remote consistency actions cause the removal of all remote copies of pages that have been modified by a committing transaction. When an update-copy lock is granted on a copy at a remote client, it is known that no active transactions are using that copy so the copy is purged from the client's cache. When all of the copies at a remote client that were affected by the committing transaction have been purged, the client sends an acknowledgement to the server and then releases the update-copy locks that it had obtained from its local lock manager. When the server has received acknowledgements from all of the remote sites, it commits the transaction, releases its locks, and informs the originating client. The originating client then releases its local locks. At this point, only the server and that one client have copies of the updated pages.

[8] If an update-copy lock request is made for a page that is no longer cached at a given client, the client simply ignores the request and informs the server.

Figure 4.4 shows an execution of the two transactions of the previous example when the O2PL-I algorithm is used. Because O2PL-I is an avoidance-based algorithm, transactions can access locally-cached page copies without communicating with the server. Therefore in the example, transaction 2, which is read-only, can execute in its entirety without sending any messages. Transaction 1 runs locally until it finishes its execution phase. At that point, it sends a copy of the updated page X to the server (message 1). The server obtains an exclusive lock on page X on behalf of transaction 1 and then sends a consistency action request to client 2 (message 2). Client 2, in the meantime, has already obtained a read lock on its local copy of page X, so the consistency action is blocked. Client 2 continues its transaction, ultimately committing and releasing its locks. At that point, an exclusive lock for page X is obtained on behalf of the consistency action for transaction 1. Once that lock is granted, page X is purged from client 2's cache. The locks held at client 2 on behalf of transaction 1 are then released and the server is informed. The server then commits transaction 1 and informs client 1, thus releasing transaction 1's locks at both sites. At the end of this execution, client 1 and the server have valid copies of page X, while client 2 no longer has a copy of the page. Comparing this example with the C2PL example shown in Figure 4.3, the striking difference is the much smaller number of messages that are sent in the O2PL-I execution. The message savings in this example is the result of the use of an avoidance-based replica update protocol. Another important difference between the two executions is the serialization order. In this case, transaction 1 was blocked by transaction 2 due to its use of deferred consistency actions. This is the opposite of the serialization order that was seen in the C2PL example.

Propagation

In the propagation-based variant of O2PL (O2PL-P), remote consistency actions result in the installation of new values of updated pages in the remote client caches. As in O2PL-I, a committing update transaction acquires update-copy locks on all copies of its updated pages at the server and remote clients. When a remote client has obtained all of the required locks on behalf of the committing transaction, it informs the server. The server treats this as the first round of a two-phase commit protocol. When the server has received acknowledgements from all remote clients, it enters the second phase of the protocol. It installs all of the updated page copies in the server-resident database, sends the new copies of pages to the remote clients, and informs the originating client that the commit has succeeded. The server and originating client then release their locks. Upon receiving a message containing updates, the remote clients install the new page copies in their cache and release their locks.[9] At this point, all sites that had copies of the updated pages at the time the transaction entered its commit phase

[9]Note: The receipt of a propagated page copy at a client does not affect the LRU status of the page at that site.

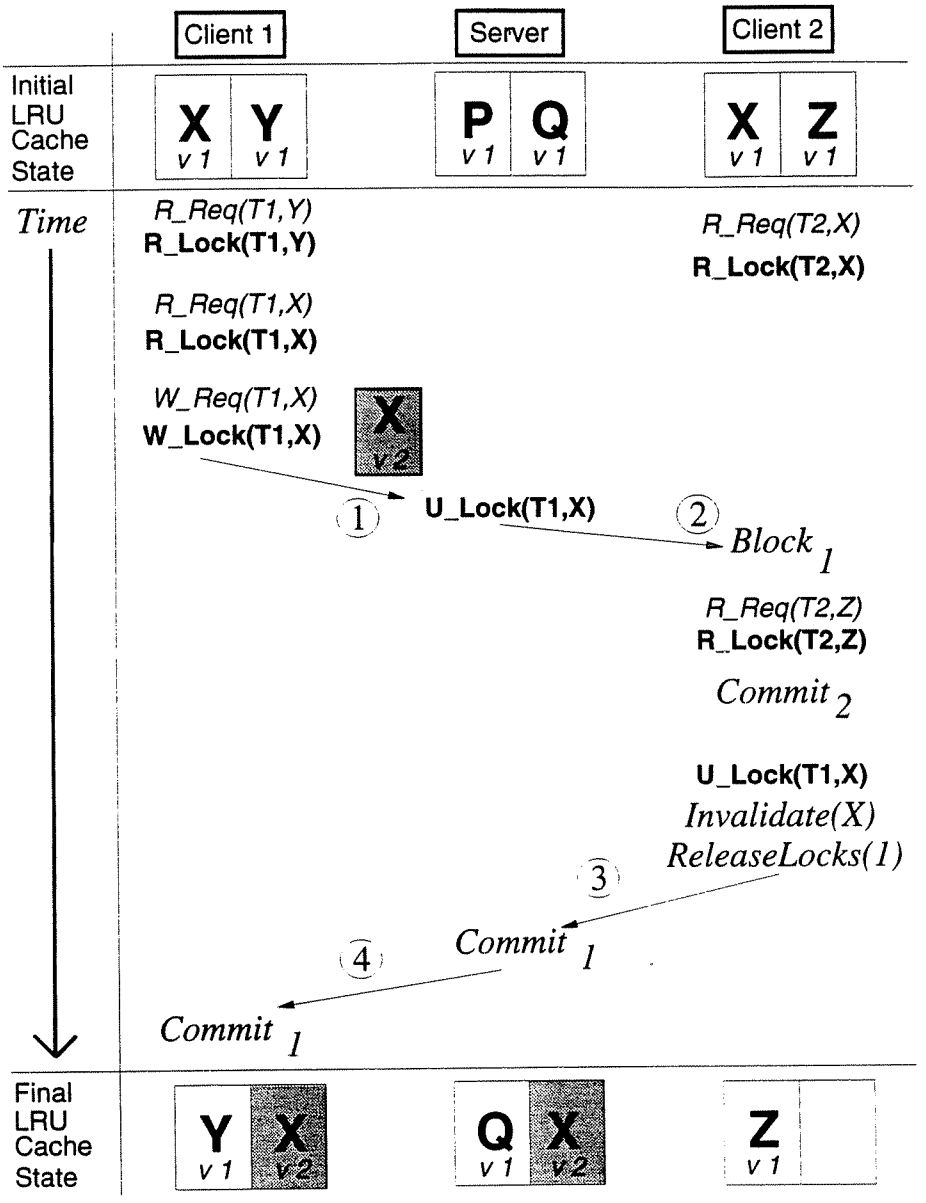


Figure 4.4: Example Execution Under O2PL-Invalidate

now have copies containing the new value of those pages.

Figure 4.5 shows the example execution under the O2PL-P algorithm. This execution proceeds in the same fashion as the previous example until the exclusive lock for page X is granted on client 2 for the consistency action. At this point, client 2 informs the server that it has obtained the necessary lock. The server interprets this message as a prepared-to-commit message so it proceeds to commit transaction 1. To do this, it sends an updated copy of page X to client 2, which installs the copy in its cache and commits transaction 1, releasing the exclusive lock on page X. Client 1 and the server also commit transaction 1 and release its locks. At the end of this execution, all sites have a copy of the current version of page X. This was accomplished through the use of an additional page-sized message compared to the O2PL-I execution.

Dynamic O2PL Algorithms

As will be shown in Chapter 5, the O2PL-I and O2PL-P algorithms can each have significant performance advantages under certain workloads. The dynamic variant of O2PL (O2PL-ND) decides on a case-by-case basis whether to invalidate a particular page copy or to update it with its new value. Dynamic algorithms attempt to provide the benefits of O2PL-P for those specialized workloads in which propagation performs well, while providing the performance of invalidation in other (more typical) cases.

The dynamic algorithm that is described and studied in this thesis, O2PL-ND, is an improvement over a previous dynamic algorithm called O2PL-Dynamic (O2PL-D) [Care91a]. The results of the performance study of [Care91a] showed that O2PL-D's heuristic for choosing between invalidation and propagation generally performed well, but it was never quite able to match the performance of the better of the O2PL-I and O2PL-P algorithms for any given workload (except in a workload with no data contention, where all O2PL algorithms performed identically). The heuristic described here was developed to solve this problem.

The original O2PL-D algorithm initially propagates updates, invalidating copies on a subsequent consistency operation if it detects that the preceding page propagation was wasted. Specifically, O2PL-D will propagate a new copy of a page to a site if *both* of the following conditions hold:

1. *The page is cache-resident at the site when the consistency operation is attempted there.*
2. *If the page has been previously propagated to the site, then that page copy has been accessed at the site since the last propagation.*

In [Care91a], it was shown that the algorithm's willingness to err on the side of propagation resulted in its performance being somewhat lower than that of O2PL-I in most cases. The new heuristic, therefore, has been

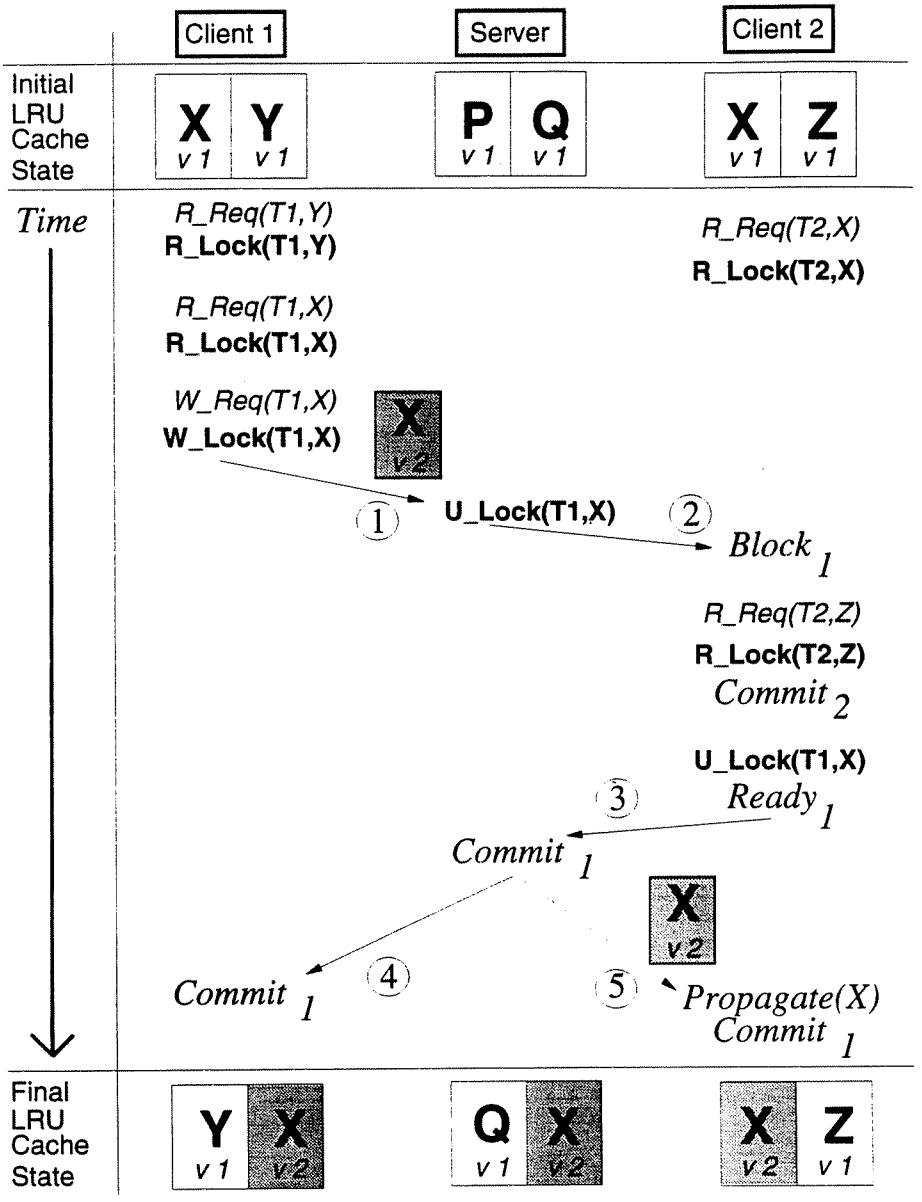


Figure 4.5: Example Execution Under O2PL-Propagate

designed to instead err on the side of invalidation if a mistake is to be made. In the new dynamic algorithm, O2PL-ND (for "New Dynamic"), an updated copy of a page will be propagated to a site only if conditions 1 and 2 of O2PL-D hold plus:

3. *The page has been previously invalidated at the site* and *the most recent invalidation was a mistake.*

The new condition ensures that O2PL-ND will invalidate a page at a site at least once before propagating it to that site. This new heuristic was shown to perform better than O2PL-D under most conditions [Fran92a]. Therefore, only results for O2PL-ND are reported in this thesis. A detailed comparison of the two heuristics can be found in [Fran92a].

In order to implement the new condition, it is necessary to have a mechanism for determining whether or not an invalidation was a mistake. This is not straightforward since invalidating a page removes the page and its buffer control information from the site, leaving no place to store information about the invalidation. To solve this problem we use a structure called the *invalidate window*. The invalidate window is a list of the last n distinct pages that have been invalidated at the site. The window size, n, is a parameter of the O2PL-ND algorithm. When a page is invalidated at a site, its page number is placed at the front of the invalidate window on that site. If the window is full and the page does not already appear in the window, then the entry at the end is pushed out of the window to make room for the new entry. If the page already appears in the window, it is simply moved from its present location to the front. When a transaction's page access request results in a page being brought into the site's buffer pool, the invalidate window is checked and if the page appears in the window, the page is marked as having had a mistaken invalidation applied to it. The page then remains marked for as long as it resides in the client's buffer.[10]

The O2PL-ND algorithm works as follows: When a consistency action request arrives at a client, the client obtains update-copy locks on the affected pages of which it cache-resident copies. The client then checks the remaining two conditions for propagation for each cache-resident page, and if both conditions are met the client will elect to receive a new copy of that page. The client retains its locks on the pages it has chosen to receive, while it invalidates (and releases its locks on) the other pages affected by the consistency request. The client then sends a message to inform the server of its decision(s). If all decisions were to invalidate, then the client is finished with the consistency action (as in O2PL-I). Otherwise, this message acts as the acknowledgement in the first phase of a two-phase commit protocol (as in O2PL-P). When the server has heard from all involved clients,

[10]Marking the page at re-access time limits the algorithm's sensitivity to the window size. If the window was checked on the arrival of a consistency maintenance action for a page, then a small window size could result in a useful page being invalidated because it has been "pushed out of the window" even though the page was recently used at the client.

it sends copies of the necessary pages to those sites that requested them. This message serves as the second phase of the commit protocol. Upon receipt of the new page copies, the clients install them in their buffer pools and release the locks on those pages.

4.3.3 Callback Locking (CB)

The third family of caching algorithms studied in the thesis is Callback Locking. As with O2PL, Callback Locking algorithms use an avoidance-based protocol. The callback algorithms, however, are based on synchronous consistency actions and are thus, more pessimistic than the O2PL algorithms.

In contrast to the O2PL algorithms, with Callback Locking a client must declare its intention to update a page to the server immediately (rather than at commit time) before it can grant a local lock to a transaction if the client does not already have an update intention registered for the page. When a client declares its intent to update a page of which other copies are cached at remote clients, the server "calls back" the conflicting copies by sending messages to the sites that have them. The write intention is registered at the server only once the server has determined that all conflicting copies have been successfully called back. Because intentions are obtained during transaction execution, transactions do not need to perform consistency maintenance actions during the commit phase (as they do under O2PL). In this thesis, two Callback Locking variants are analyzed: Callback-Read (CB-R), in which write intentions are registered only for the duration of a single transaction, and Callback-All (CB-A), which associates intentions with clients, so that they can span multiple transactions at those clients.

As with the O2PL algorithms, the server keeps track of the locations of cached copies throughout the system. Transactions obtain locks from the local lock manager at the client on which they execute. Read lock requests and requests for update locks on pages for which the client has already registered its write intention, can be granted immediately without contacting the server. Write lock requests for which an intention has not yet been registered cause a "write intention fault". On a write intention fault, the client must register its intention with the server and must wait until the server responds that the intention has been registered before continuing.

CB-R, which associates intentions with individual transactions, works as follows: When a request to register a write intention for a page arrives at the server, the server issues callback requests to all sites (except the requester) that hold a cached copy of the page. At a client, such a callback request is treated as a request for an exclusive (update-copy) lock on the specified page. If the request cannot be granted immediately, (due to a lock conflict with an active transaction) the client responds to the server saying that the page is currently in use. When the callback request is eventually granted at the client, the page is removed from the client's buffer and an acknowledgement message is sent to the server. When all callbacks have been acknowledged to the server, the

server registers the write intention on the page for the requesting client and informs that client. Any subsequent read or write requests for the page by transactions from other clients will be blocked at the server until the write intention is released by the holding transaction. At the end of the transaction, the client sends copies of the updated pages to the server and revokes its write intentions, retaining copies of the pages (and hence, implicit permission to read those pages) in its cache.

Figure 4.6 shows the execution of the example transactions under the CB-R algorithm. Once again, transactions can read cache-resident data without contacting the server. Thus transaction 1 runs locally until it wishes to update page X. At this point client 1 sends a write intention declaration to the server and the server obtains an exclusive lock for page X on behalf of transaction 1. Before it can register the intention, however, it must first call back client 2's copy of the page. In the meantime, transaction 2 has started and has obtained a local read lock on page X. When the callback request arrives at client 2, it is blocked behind this read lock. Client 2 sends a message to the server informing it of the block, so that it can check for global deadlocks. Transaction 2 eventually commits, which releases the read lock on page X and allows the callback to proceed. The callback results in page X being removed from client 2's cache, the server is informed that the callback has been successful, and the server then informs transaction 1 that it's intention has been registered. Transaction 1 completes and informs the server that it has committed. The server can then release the exclusive lock on page X (which represented transaction 1's write intention). At the end of this execution, client 1 and the server have new copies of page X, while client 2 has had its copy of page X removed. It should be noted that although the message count in this example is similar to that under the O2PL algorithms, CB-R would typically pay a higher price in messages because it has to send a separate message to the server for each page that it intends to update.

The CB-A algorithm works similarly to CB-R, except that write intentions are not revoked at the end of a transaction. If a read request for a page arrives at the server and a write intention for the page is currently registered for some other client, the server sends a *downgrade* request to that client. A downgrade request is similar to a callback request, but rather than removing the page from its buffer, the client simply informs the server that it no longer has a registered write intention on the page. At a remote client, a downgrade request for a page copy must first obtain a read lock on the page in order to ensure that no transactions active at the client are currently holding write locks on the page. If a conflict arises, the downgrade request blocks, and a message is sent to the server informing it of the conflict.

As in CB-R, clients send copies of the pages dirtied by a transaction to the server when the transaction commits. However, in CB-A this is done only to simplify recovery, as no other sites can access a page while it is exclusively cached at another site. Thus, it is possible in theory, to avoid sending dirty pages back to the server

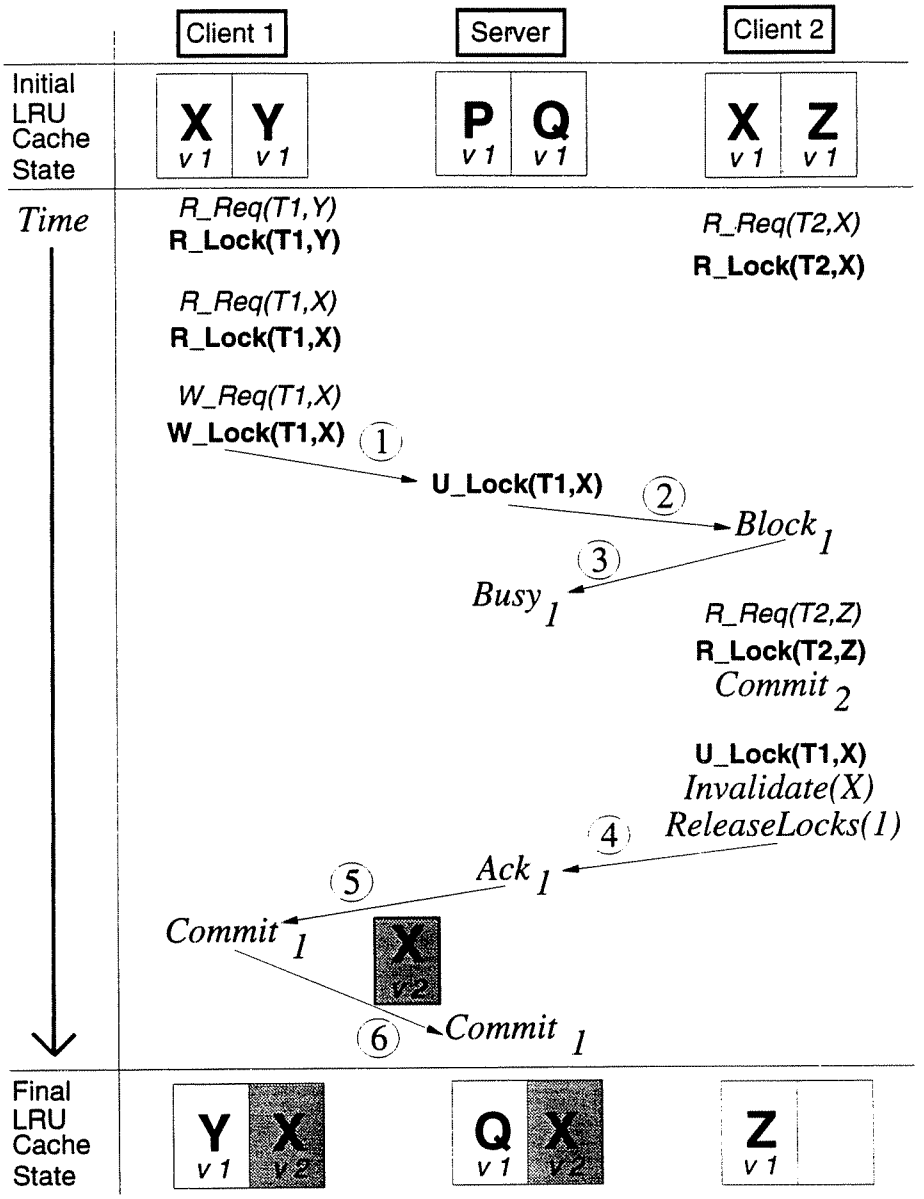


Figure 4.6: Example Execution Under Callback-Read

until the write intention on a dirty page is downgraded or dropped. It should be noted that CB-R does not have to send downgrade requests because it is known that a conflicting write intention registered by another client will be revoked in any case when the transaction at the other client terminates.

Callback-style algorithms were originally introduced to maintain cache consistency in distributed file systems such as Andrew [Howa88] and Sprite [Nels88], both of which provide weaker forms of consistency than that required by database systems. More recently, a Callback Locking algorithm that provides transaction serializability has been employed in the ObjectStore OODBMS [Lamb91]. An algorithm similar to CB-R was also studied in [Wang91]. The algorithm studied there differed from CB-R with regard to deadlock detection, as it considered all cached locks to be in-use for the purpose of computing the waits-for graph at the server. Such an approach can suffer aborts due to phantom deadlocks, especially with large client caches. To avoid this problem, CB-R and CB-A both adopt a convention from ObjectStore: copy sites always respond to a callback request immediately, even if the page is currently in use at the site. This allows deadlock detection to be performed at the server with accurate information.

4.3.4 Other Proposed Algorithms

Several of the algorithms that appear in the taxonomy of Section 4.2 are not addressed in the performance study that is presented in Chapter 5. These algorithms are discussed briefly here. The first paper that appeared on cache consistency for client-server OODBMSs was from HP Labs [Wilk90]. In that paper, two algorithms were proposed and studied:. *Cache Locks* is a detection based protocol that defers validation of transactions until commit time. Special lock modes and long-running "envelope transactions" are used to determine when transactions have accessed stale data. Notification hints are sent after a transaction commits in order to reduce aborts due to access to stale cached data. *Notify Locks* is a avoidance-based protocol that uses deferred propagations as consistency actions. The server tracks page copies and sends new values to remote clients when a transaction commits. Furthermore a transaction is allowed to commit before its consistency actions are acknowledged by the remote clients. As a result, remote clients must abort any active transactions that have accessed a modified page. Both of these algorithms require (sometimes multiple) handshakes between the client and the server to avoid race conditions at commit.

No-wait locking algorithms are studied in [Wang91]. No-wait algorithms are detection-based algorithms that try to hide the latency of validations at the server by sending consistency messages asynchronously. As with the [Wilk90] algorithms, transactions must abort if they have accessed stale data. By initiating the consistency checks before commit time, however, the window during which data can become invalid is shortened. In an

attempt to further reduce the possibility of stale data access, the no-wait algorithm was also extended with a propagation-based notification hint scheme. The performance of this algorithm, called No-Wait Locking with Notifications, is also studied in [Wang91].

A validation-based optimistic scheme has been proposed for use in the Thor project at MIT [Lisk92]. Transactions execute at clients without performing consistency operations. At the end of the execution phase, read and write set information is sent to the server for validation and if a conflict is detected, the transaction is aborted. In order to reduce aborts caused by access to stale cached data, an invalidation-based notification protocol is used. To date, no performance analysis of this algorithm has been published.

4.3.5 Summary

This section briefly summarizes the cache consistency maintenance algorithms that have been described in this chapter. All of the algorithms except for B2PL allow inter-transaction caching of data pages. B2PL and C2PL are detection-based protocols that rely on synchronous consistency actions and do not send notifications. C2PL uses check-on-access to ensure the consistency of accessed data. The O2PL algorithms are lock-based, read one/write all protocols. Locks are obtained locally at clients during transaction execution, deferring global acquisition of write locks until the commit phase. The three O2PL variants use invalidation and/or propagation to maintain consistency. The Callback algorithms are also lock-based, avoidance-based protocols, but they rely on synchronous consistency actions in order to avoid aborts. Both of the Callback algorithms allow cached pages to be read without server interaction. CB-A also allows write intentions to be retained across transaction boundaries.

All of the algorithms that are studied in this thesis are based on the premise that cache consistency maintenance in a page server DBMS is an instance of the replicated data management problem. Therefore, the algorithms proposed and studied are all based on techniques from traditional database systems, which have well understood correctness properties. These techniques include two-phase locking, two-phase commit, and write-ahead-logging. As stated in Section 2.4.1, page server DBMSs are likely to be adopted in application environments that are less tolerant of aborts than more traditional DBMS environments. The algorithms of this thesis have been developed with this constraint in mind. The server-based 2PL and Callback algorithms are both pessimistic. The O2PL algorithms are only partially optimistic — reads are done in a pessimistic manner, while consistency actions for writes are deferred until commit. As a result, none of these algorithms will suffer from transaction aborts due to accessing stale cached data. In contrast, the no-wait locking algorithms, as well as the HP and MIT algorithms, do not protect transactions from accessing stale data and subsequently having to abort. Each of those

algorithms has been extended with a notification protocol to try to reduce the likelihood of such aborts.

This chapter has provided a survey of the design space that is available for the development of cache consistency maintenance algorithms and has provided a framework for reasoning about some of the options. A number of cache management algorithms that utilize different combinations of techniques for ensuring cache consistency have been proposed and described. The analysis of these algorithms under a range of workloads is the subject of the performance study presented in the following chapter.

Chapter 5

Performance of Cache Consistency Algorithms

This chapter presents a performance analysis of the three families of cache consistency maintenance algorithms described in Section 4.3, namely, Server-based Two-Phase Locking (S2PL), Optimistic Two-Phase Locking (O2PL), and Callback Locking (CB). Seven algorithms are studied under a range of workloads and system configurations. In addition to measuring the performance of these specific algorithms, the experiments presented here also provide insight into many of the tradeoffs involved in cache maintenance, including: optimism vs. pessimism, detection vs. avoidance, and invalidation vs. propagation. The analysis is performed using the simulation model described in Chapter 3. The parameter settings and workloads that are used in the study are described in the following section. The results are then presented in the two subsequent sections. Performance results for the server-based 2PL and optimistic 2PL algorithms are presented first. The performance of the callback locking algorithms is then studied and compared to the better of the S2PL and O2PL algorithms.

5.1 System Configuration and Workloads

5.1.1 System Parameter Settings and Metrics

The settings used for the system and resource parameters (as described in Section 3.2) are presented in Table 5.1. The number of client workstations in the system is varied from one to 25. In order to make detailed simulations of systems with fractionally large client and server buffer pools computationally feasible, a relatively small database size is used. The server buffer pool and client caches (buffer pools) are sized accordingly, and are

Parameter	Meaning	Setting
PageSize	Size of a page	4,096 bytes
DatabaseSize	Size of database in pages	1250
NumClients	Number of client workstations	1 to 25
ClientCPU	Instruction rate of client CPU	15 MIPS
ServerCPU	Instruction rate of server CPU	30 MIPS
ClientBufSize	Per-client buffer size	5% or 25% of DB
ServerBufSize	Server buffer size	50% of DB size
ServerDisks	Number of disks at server	2 disks
MinDiskTime	Minimum disk access time	10 millisecond
MaxDiskTime	Maximum disk access time	30 milliseconds
NetBandwidth	Network bandwidth	8 or 80 Mbits/sec
FixedMsgInst	Fixed no. of inst. per message	20,000 instructions
PerByteMsgInst	No. of addl. inst. per msg. byte	10,000 inst. per 4Kb
ControlMsgSize	Size in bytes of a control message	256 bytes
LockInst	Instructions per lock/unlock pair	300 instructions
RegisterCopyInst	Inst. to register/unregister a copy	300 instructions
DiskOverheadInst	CPU Overhead to perform I/O	5000 instructions
DeadlockInterval	Global deadlock detection frequency	1 second (for O2PL)

Table 5.1: System and Overhead Parameter Settings

specified as a percentage of the database size. Two settings are used for client caches: "small", in which each client has enough memory available to cache 5% of the database, and "large", in which each client can cache 25% of the database. The server buffer pool is large enough to hold 50% of the database in all of the experiments presented here. It is the ratio of these cache sizes to each other, to the size of the database, and to the access patterns of the workloads that are the important factors in these experiments, rather than the absolute size of the database or the caches themselves.[1] All of the clients in these experiments are diskless; the use of client disks is addressed in Chapter 7. Results are presented for two different network bandwidths, called "slow" (8 Mbits/sec) and "fast" (80 Mbits/sec); these correspond roughly to discounted bandwidths of Ethernet and FDDI technology respectively.

For most of the experiments in this chapter, the main metric presented is throughput, measured in transactions per second.[2] Where necessary, auxiliary performance measures are also discussed. A number of metrics such as message counts, message volume, and disk I/Os are presented as "per commit" values. These metrics are computed by taking the total count for the metric (e.g., the total number of messages routed through the network) over the duration of the simulation run and dividing by the number of transactions that committed during that run. As a result, these averages also take into account work that was done on behalf of aborted transactions.

[1] This is the reason that analytical models of buffer behavior can be developed (as in [Dan92]).

[2] Because the simulation is a closed model, throughput and response time are equivalent metrics.

To ensure the statistical validity of the results, it was verified that the 90% confidence intervals for transaction response times (computed using batch means) were sufficiently tight. The size of these confidence intervals was within a few percent of the mean in all cases, which is more than sufficient for the purposes of this study.

5.1.2 Workloads

Table 5.2 summarizes the workloads that are used in this study. The HOTCOLD workload (which was described

Parameter	HOTCOLD	PRIVATE	FEED	UNI-FORM	HICON
TransSize	20 pages	16 pages	5 pages	20 pages	20 pages
HotBounds	p to p+49, $p = 50(n\text{-}1)+1$	p to p+24, $p = 25(n\text{-}1)+1$	1 to 50	-	1 to 250
ColdBounds	rest of DB	626 to 1,250	rest of DB	all DB	rest of DB
HotAccProb	0.8	0.8	0.8	-	0.8
ColdAccProb	0.2	0.2	0.2	1.0	0.2
HotWrtProb	0.2	0.2	1.0/0.0	-	0.0 to 0.5
ColdWrtProb	0.2	0.0	0.0/0.0	0.2	0.2
PerPageInst	30,000	30,000	30,000	30,000	30,000
ThinkTime	0	0	0	0	0

Table 5.2: Workload Parameter Settings for Client n

in Section 3.3) has a high degree of locality per client and a moderate amount of sharing and data contention among clients. The PRIVATE workload has a private hot region that is read and written by each client, and a shared cold region that is accessed in a read-only manner. This workload is intended to model an environment such as a large CAD system, in which each user has a portion of the design that they are currently working on and modifying, while accessing parts from a shared library of components. The FEED workload was also described in Section 3.3. It is intended to model a directional information flow among clients, as would exist, for example, in a stock quotation system in which one site produces data while the other sites consume it. UNIFORM is a low-locality, moderate write probability workload which is used to examine the consistency algorithms in a case where caching is not expected to pay off significantly. Finally, HICON is a workload with varying degrees of data contention. It is similar to skewed workloads that are often used to study shared-disk transaction processing systems, and it is introduced here to investigate the effects of data contention on the various algorithms.

5.2 Server-based 2PL and Optimistic 2PL

In this section, the performance of the two Server-based 2PL algorithms (B2PL and C2PL) and the three variants of Optimistic 2PL are examined. In all of the experiments described here, the adaptive variant of O2PL (O2PL-ND) was run with a window size of 20 pages. An early concern with the heuristic used by O2PL-ND was its potential sensitivity to the invalidate window size (as described in Section 4.3.2). A series of tests using the parameters from [Care91a] found that in all cases but one, the algorithm was insensitive to the window size within a range of 10 to 100 pages (0.8% to 8% of the database size). The exception case and the reasons for the insensitivity of the algorithm in the other cases will be discussed in the following sections.

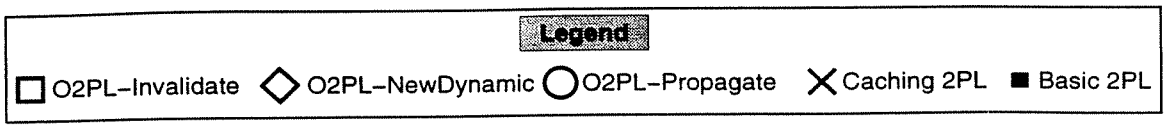
5.2.1 Experiment 1 : The HOTCOLD Workload

The first results presented are those for the HOTCOLD workload. In this workload, as indicated in Table 5.2, each client has its own 50-page hot region of the database to which 80% of its accesses are directed, while the remaining accesses are directed to the rest of the database. This workload represents a situation where client transactions favor disjoint regions of the database, but where some read-write sharing exists. This workload tests the algorithms' ability to exploit the affinity between clients and particular data items, while also testing their ability to manage read-write sharing efficiently.

HOTCOLD Workload, Small Client Caches

Figure 5.1 shows the aggregate system throughput as a function of the number of clients in the system when using the slower network and small client buffer pools (5% of the database size). In this experiment, the O2PL algorithms perform best, with C2PL performing at a somewhat lower level. B2PL, the non-caching algorithm, has the lowest performance throughout the entire range of client populations. Furthermore, the dynamic variant of O2PL (O2PL-ND) matches the performance of O2PL-I (the better of the two static O2PL algorithms) across the range client populations tested here. Initially, the three O2PL algorithms perform alike, with near-linear scale-up in the range from 1 to 5 clients. C2PL also scales reasonably well in this range, while B2PL exhibits the poorest scaling.

The superior performance of the O2PL algorithms in this range is due to communication savings compared to the server-based algorithms. The O2PL algorithms require only 18-20 messages per committed transaction in this range, while the server-based algorithms both average nearly 52 messages per transaction. This discrepancy highlights one of the major differences between cache management overhead for O2PL and that for C2PL. In this



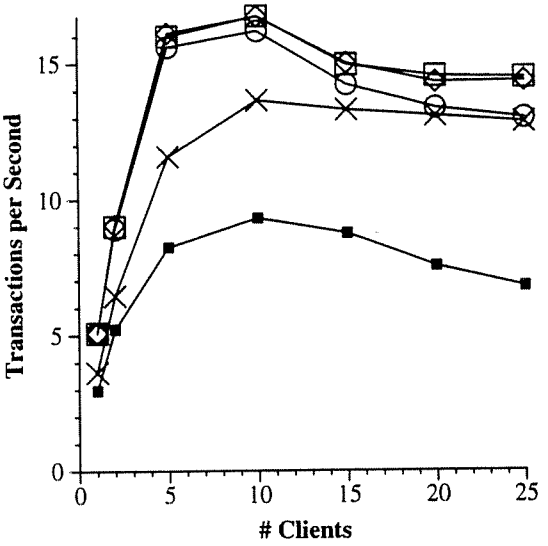


Figure 5.1: Throughput
(HOTCOLD, 5% Client Bufs, Slow Net)

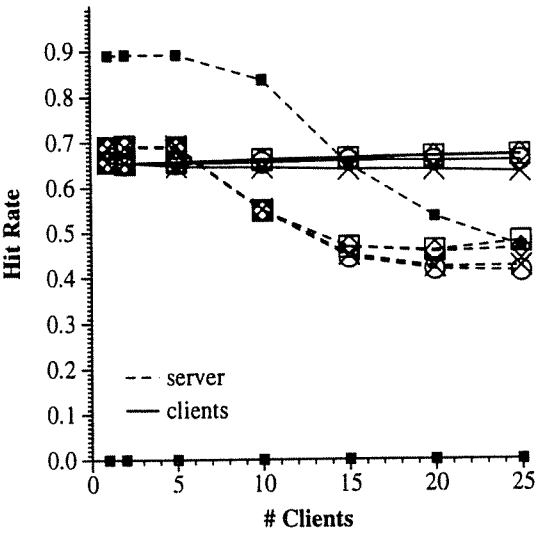


Figure 5.2: Buffer Hit Rates
(HOTCOLD, 5% Client Bufs, Slow Net)

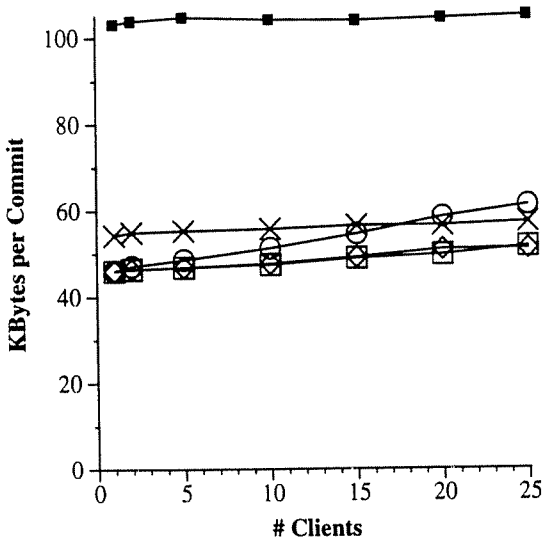


Figure 5.3: Message Volume per Commit
(HOTCOLD, 5% Client Bufs, Slow Net)

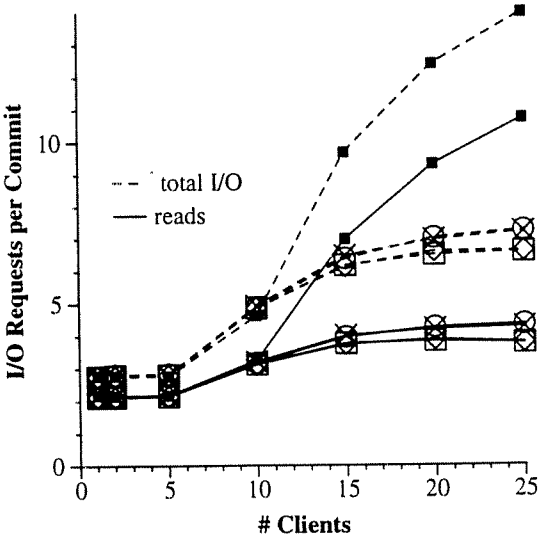


Figure 5.4: I/Os per Commit
(HOTCOLD, 5% Client Bufs, Slow Net)

experiment, each client can cache 5% of the database locally, while drawing 80% of its page accesses from only 4% of the database (i.e., the client-specific hot range). Thus, as can be seen in the client buffer hit rates (shown in Figure 5.2), clients using O2PL and C2PL can satisfy the majority of their page references from their local cache. The O2PL algorithms are better able to capitalize on the cache hits, however, as they allow transactions to read locally-cached data without having to contact the server. In contrast, C2PL must send a lock request to the server on each initial access, even if the required page is cached locally. The main cost of these extra messages is additional path length due to CPU processing at both the clients and the server. Communication costs are also the cause of B2PL's poor performance in this case. Because B2PL does not cache across page boundaries, each initial page access at a client requires a page transfer from the server. Thus, even though B2PL sends the same number of messages as C2PL, it requires many more bytes to be transferred across the network (this is shown in Figure 5.3). Especially with the slow network setting, this additional traffic can lead to the network becoming a bottleneck for B2PL (at 5 clients, the network for B2PL is about 86% utilized). The fact that communication is indeed the cause for these performance differences is confirmed by Figure 5.4, which shows that all five of the algorithms perform the same amount of disk I/O in the 1-5 client range.

Looking out beyond 5 clients in Figure 5.1, it can be seen that all of the algorithms reach their peak throughput with a client population of 10. Beyond 10 clients, the algorithms all exhibit a "thrashing" behavior in which the aggregate throughput actually decreases as clients are added to the system. Thrashing continues until 25 clients, at which point the performance of the caching algorithms levels out. The reasons behind this behavior can be seen in the server buffer hit rates (the dashed lines in Figure 5.2) and in the resulting disk requirements shown in Figure 5.4. Recall that each client has a 4% hot range, and also accesses pages outside of this range, while the server's buffer pool can hold 50% of the database. At 10 clients and beyond, the server is no longer able to contain the hot region pages for all active clients, so it is able to satisfy fewer client requests from its cache and its hit rate begins to drop off.[3] Eventually, the server hit rate drops to slightly below 50%, where 50% is what would be expected with a uniform (rather than skewed) workload. The lower server hit rate results in an increase in the number of I/Os that must be performed per transaction. This adds I/O pathlength to each transaction, resulting in the thrashing behavior seen in Figure 5.1. In this experiment, all of the algorithms eventually become disk bound, and at the point that they reach the disk bottleneck, their relative performance is determined by the per transaction I/O requirements, as shown in Figure 5.4.

Focusing on the individual algorithms at 10 clients and beyond, B2PL can be seen to exhibit thrashing

[3]Note that although each client buffer pool is larger than its hot region, only a portion of that hot region can be cache-resident at a client due to the presence of cold range pages that have also been referenced by the client.

behavior similar to that of the caching algorithms. As with the other algorithms, B2PL suffers when the server can no longer hold the hot pages of all active clients. Furthermore, the resulting increase in disk I/O eventually causes the disk to become the critical resource for B2PL (rather than the network), and B2PL becomes disk bound at 20 clients. C2PL outperforms B2PL at first because of its lower network capacity requirements, and later, because it requires far fewer I/Os than B2PL. The O2PL algorithms also thrash, for the reasons stated previously. However, O2PL-I and O2PL-ND are still able to maintain the best throughput for all client populations. O2PL-I and O2PL-ND perform similarly because O2PL-ND primarily chooses invalidation over propagation in this workload. The reason that the invalidation-based O2PL algorithms outperform C2PL as the disk becomes the dominant resource is because they have somewhat lower I/O requirements; this is because they have a larger *effective client cache* than C2PL. Because C2PL is detection-based, stale data can reside in the client buffer pools. This stale data takes up space that could otherwise be used for holding valid pages. In contrast, the O2PL invalidation mechanism removes data pages from remote caches as they become invalid, thus allowing the entire cache to be used to hold valid pages.[4]

Turning to the O2PL algorithms, the lower performance of O2PL-P, as compared to O2PL-I and O2PL-ND is due to both messages and disk I/O. The additional messages are due to the propagation of updated pages to remote clients. As can be seen in Figure 5.2, the additional disk I/O is caused by slightly lower server and client buffer hit rates. These lower hit rates result from the fact that with propagation of updates, pages are removed from the client buffers only when they are aged out due to disuse. Aged out pages are not likely to be found in the server buffer pool if they are needed (thus lowering the server buffer hit rate), and propagated pages take up valuable space in the small client buffer pool if they are not needed (thus lowering the client hit rate). The O2PL-ND algorithm avoids these problems because the majority of its consistency operations are invalidations. This is because the invalidate window is small relative to the number of pages in the database that can be updated. Sensitivity tests showed that while the number of propagations per committed transaction increases with the window size, it remains quite small in the range of window sizes tested (10 to 100) and the number of useless propagations grows slowly and smoothly with the invalidate window size in that range. Therefore, there is reasonable room for error in choosing the window size as long as it is kept well below the size of the portion of the database that is subject to updates.

[4]To verify this effect, the client buffer pool code was instrumented to keep track of the average number of valid pages in the buffer pool for C2PL and O2PL-I. The results showed that in this experiment, C2PL's effective cache size is about 5% smaller than O2PL-I's in the 25-client case. An experiment was then run in which C2PL's client cache was increased by this amount, and the resulting I/O activity indeed matched that of O2PL-I.

The HOTCOLD Workload, Large Client Caches

Figure 5.5 shows the throughput results for the HOTCOLD workload and slow network when the client buffer size is increased to 25% of the database size (312 pages). In this case, all of the algorithms except for B2PL and O2PL-P benefit from the additional memory. Again, B2PL has the worst performance, and O2PL-I and O2PL-ND behave similarly and have the best performance as clients are added to the system. The main difference here is that O2PL-P's performance (both relative to the other algorithms and in absolute terms) actually degrades due to the larger client caches.

B2PL does not cache across transaction boundaries so it cannot use the additional client memory; it performs exactly as in the previous (smaller client cache) case. As before, it is limited by high network bandwidth requirements when few clients are present and by disk I/O requirements when many clients are present. In contrast, C2PL is able to exploit the additional client memory and thus, it achieves an improvement in its client buffer hit rate (shown in the solid lines of Figure 5.6). In this case, each client's buffer pool is large enough (25% of the database) to hold the client's hot region pages, with ample room to spare (21% of the database size) for holding accessed cold pages. This improved hit rate translates into a reduction in the *volume* of data sent per transaction (Figure 5.7) but does not affect the *number* of messages that must be sent, as C2PL must contact the server for access permission even for cache-resident pages. The improved hit rate also reduces C2PL's disk read requirements.

Turning to the O2PL algorithms, O2PL-I and O2PL-ND perform the best here, and have a higher peak throughput than in the previous case. Once again, the algorithms perform similarly because O2PL-ND chooses invalidation over propagation for the majority of its consistency actions. Both algorithms thrash somewhat beyond 10 clients due to an increase in I/O requirements, as before. Here, however, the additional I/Os are *writes*. As clients are added to the system, the rate of page requests to the server and the arrival of dirty pages at the server both increase, causing an increase in the turnover rate of pages in the server buffer pool and a corresponding reduction in the residency time for individual pages. When dirty pages are replaced from the server's buffer pool, they must be written to disk. The reduction in page residency time reduces the probability that multiple transaction writes can be applied to a page during its residency. The result is an increase in the actual number of disk writes required per transaction as clients are added. O2PL-I and O2PL-ND are ultimately disk-bound in this case.

In contrast to the invalidation-based O2PL algorithms, O2PL-P can be seen to suffer a significant degradation in performance beyond five clients; eventually performing even below the level of C2PL. The reason for O2PL-P's

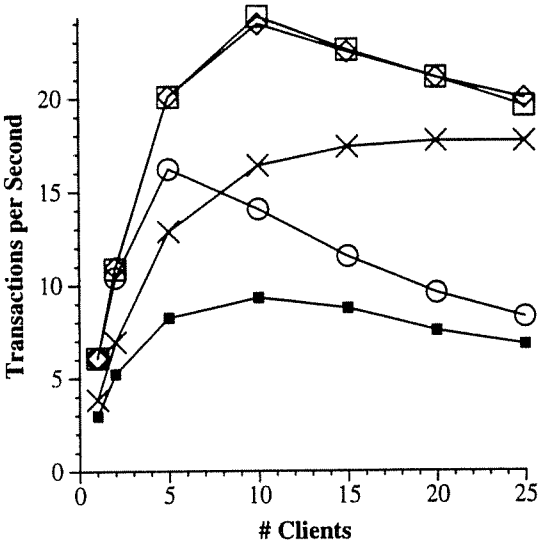


Figure 5.5: Throughput
(HOTCOLD, 25% Client Bufs, Slow Net)

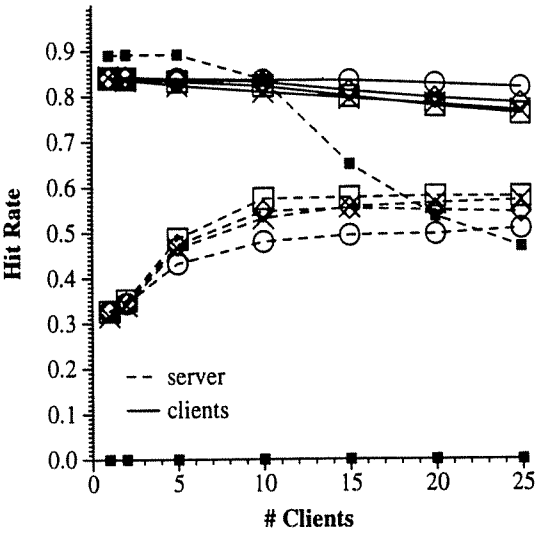


Figure 5.6: Buffer Hit Rates
(HOTCOLD, 25% Client Bufs, Slow Net)

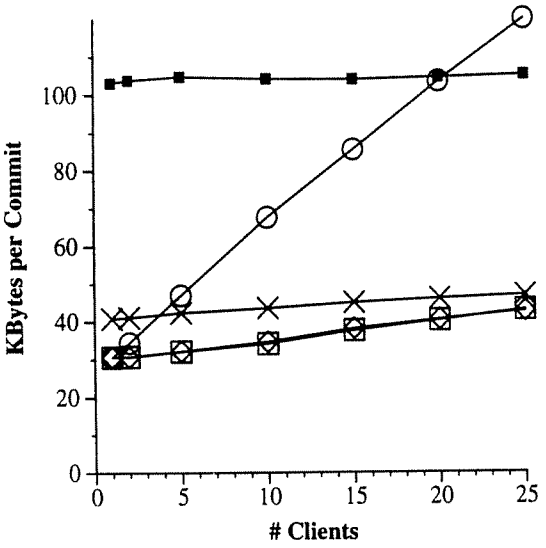


Figure 5.7: Message Volume per Commit
(HOTCOLD, 25% Client Bufs, Slow Net)

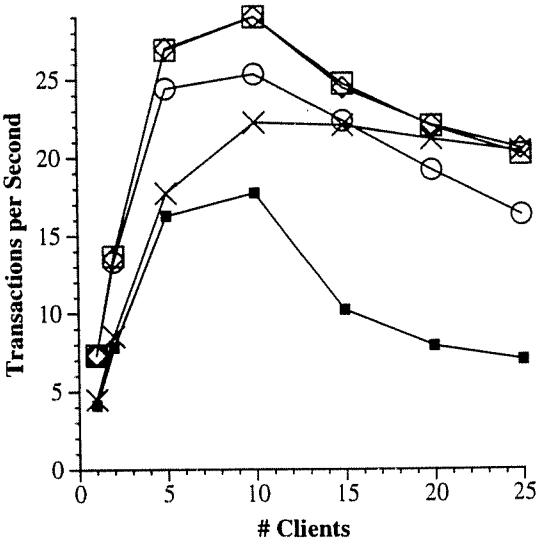


Figure 5.8: Throughput.
(HOTCOLD, 25% Client Bufs, Fast Net)

poor performance in this case is a dramatic increase in message volume as clients are added. This increase is shown in Figure 5.7 and is due to the page-sized messages that are used to propagate updates to remote clients. Due to the larger client caches, the number of sites to which propagations must be sent is much higher here than in the small client buffer case. At 25 clients, O2PL-P sends propagations to 13 remote clients per transaction on average, versus an average of about 2 remote clients previously. Furthermore, the vast majority of these propagations are "wasted". That is, the remote copies are propagated to again, or else dropped from the cache before the propagated value is used.[5] The reason that O2PL-P performs much worse here than in the previous case is that with small client caches, the tendency of O2PL-P to send worthless propagations is reduced as cold pages are quickly pushed out of the caches. This experiment demonstrates that using propagation to implement consistency actions is a potentially dangerous policy, as its performance is very sensitive to the size of the client caches.

The throughput results for the system with large client caches and the faster network setting are shown in Figure 5.8. The ordering of the algorithms is largely the same as was seen when using the slow network. With small client populations, the faster network results in improved performance for all algorithms. With larger populations, however, the algorithms that became disk-bound in the previous case (B2PL, O2PL-I , and O2PL-ND) also become disk-bound here, and as a result, they have the same throughput as in the slow network case. In contrast, C2PL, which did not reach a disk bottleneck before, has improved performance throughout the entire population range in this case. It ultimately approaches the performance of the invalidation-based O2PL algorithms as it also nears a disk bottleneck. The major beneficiary of the improved network bandwidth, however, is O2PL-P. The cost of propagations is greatly reduced in this case, and O2PL-P thus fares somewhat better here. Excessive propagation is still its downfall, however, as O2PL-P ultimately becomes bottlenecked at the server CPU due to the cost of sending propagation messages.

One other interesting effect that arises with the use of large client caches should be noted here. As can be seen in the server hit rates shown in Figure 5.6, all of the algorithms that allow caching experience a poor server buffer hit rate with small client populations. This poor hit rate is due to a high degree of *correlation* between the contents of client caches and the server buffer pool. A page that a client requests from the server is first located in the server's buffer pool and then subsequently cached at the client. As a result, with small client populations, much of the server's buffer pool is filled with pages that are also cache-resident at clients. This leaves a smaller effective server buffer pool from which to service client cache misses. As clients are added, the degree of buffer

[5]The simulator has been instrumented to measure the usefulness of propagations. In this case, about 90% of O2PL-P's propagations are wasted across the entire range of client populations.

correlation between any one client and the server is reduced and the effect dissipates. While this affect has only a minor impact on the performance results in this chapter, it will turn out to be one of the main factors in the performance of the global memory management algorithms studied in Chapter 6.

5.2.2 Experiment 2: The PRIVATE Workload

The next results to be discussed are those obtained using the PRIVATE workload. As stated earlier, the PRIVATE workload is intended to represent an environment such as a large CAD-based engineering project, in which each engineer develops a portion of an overall design while referencing a standard library of parts. In this workload, as shown in Table 5.2, each client has a private 25 page region of the database to which half of its accesses are directed; the other half of its accesses are directed to a 625 page region of the database that is shared in a read-only manner. Thus, under this workload, there is significant locality of access per client and no read-write sharing of data among clients. As a result, this is a very favorable workload for client caching. The PRIVATE workload is also useful for gauging the best case performance advantages for optimistic techniques, as the absence of any data conflicts works to the advantage of optimism.

The throughput results for this workload with small (5%) client caches and the slow network are shown in Figure 5.9. Due to the absence of read-write sharing in this workload, updates never require consistency actions to be performed at remote sites, and thus, the three O2PL algorithms all have the same performance. C2PL performs below the level of the O2PL algorithms. With 5 clients, O2PL's throughput is about 25% higher than C2PL's, but the gap narrows as clients are added. At 25 clients, C2PL performs nearly as well as O2PL. With small client populations, C2PL's performance is hurt by its need to validate pages at the server — it requires 39 messages per committed transaction, compared to 22 messages for the O2PL algorithms. In terms of I/O, however, C2PL and O2PL have similar buffer hit rates at the clients and the server due to the lack of read-write sharing here. As clients are added, the server disks become the dominant resource, and therefore, the performance of C2PL approaches that of the O2PL algorithms. B2PL again has the poorest performance among the algorithms tested. As in the HOTCOLD cases, its poor performance is initially due to high message volume requirements, and later, as the server becomes disk-bound, B2PL suffers due to its high per-transaction disk I/O requirements. When the fast network is used in conjunction with the small client caches (not shown), the ordering of the algorithms is the same as those obtained with the slow network. There are, however, several differences in those results. One difference is that prior to becoming disk bound (i.e., in the range up to 10 clients), B2PL performs nearly as well as C2PL because the network is no longer a bottleneck. C2PL and B2PL both perform below the level of the O2PL algorithms until all of the algorithms become disk-bound at the server.

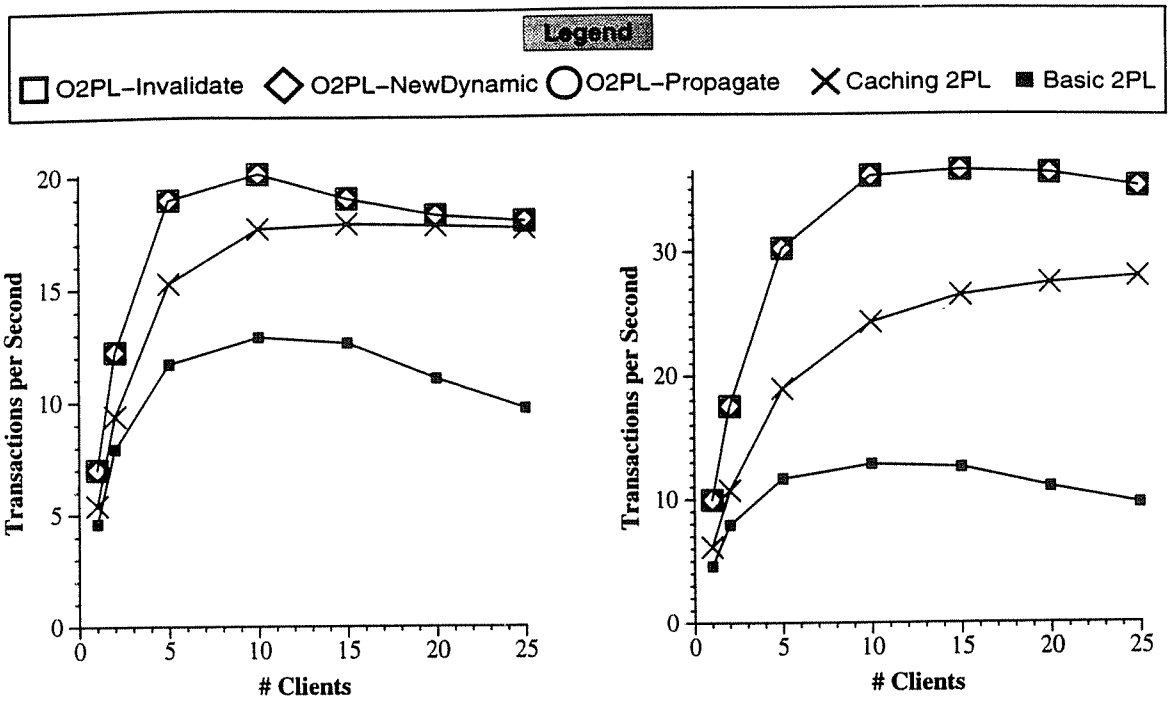


Figure 5.9: Throughput
(PRIVATE, 5% Client Bufs, Slow Net)

Figure 5.10: Throughput
(PRIVATE, 25% Client Bufs, Slow Net)

Once the disk bottleneck is reached, C2PL and the O2PL algorithms perform identically, while B2PL performs at a lower level.

When the larger (25%) client caches are used, the throughput differences among O2PL, C2PL, and B2PL become more pronounced. Figure 5.10 shows the throughput obtained with the large client caches and the slow network. B2PL does not use the larger client caches, so it has the same performance as with small client caches. In contrast, O2PL and C2PL both see an improvement in client buffer hit rates for cold pages This improvement results in a savings in the number of messages sent by O2PL — it sends only 12 messages per transaction, as compared to 22 messages when the small caches are used. C2PL does not realize any savings in this regard, however, as it still needs to validate its cache-resident pages with the server. Thus, the optimism of deferring consistency actions on writes pays significant dividends for the O2PL algorithms in this case.

5.2.3 Experiment 3: The FEED Workload

The FEED workload is a very different type of workload from the two that have been examined so far. In the FEED workload (as shown in Section 3.3), all clients share a common 50 page hot area to which 80% of their accesses are directed. Client #1 writes every hot region page that it accesses, while the remaining clients

per transaction. In this case, however, O2PL-P significantly outperforms O2PL-I. The larger buffer pools allow all of the clients to keep the 50 hot pages in memory, so many fewer of O2PL-P's propagations are wasted due to pages being forced out of the buffer pool. As a result, O2PL-I sends more page requests to the server than O2PL-P, which results in increased path length for transactions under O2PL-I. This additional path length includes both additional messages (O2PL-I sends about 6 messages per transaction versus about 3 per transaction for O2PL-P) and additional lock requests at the server. This case demonstrates the potential advantages of propagation for certain workloads.

The other important result of this test is that the dynamic O2PL-ND algorithm is able to closely track the performance of O2PL-P. O2PL-ND performs slightly worse than O2PL-P here, but it does much better than O2PL-I. This case demonstrates that the simple heuristic used by O2PL-ND is effective in adapting to the performance of the better of the two static O2PL algorithms over a range of workloads. Also, in contrast to what was seen with the small client caches, O2PL-ND is relatively insensitive to the invalidate window size when client caches are large. This is because hot range pages are dropped from clients very infrequently, being removed mainly as the result of invalidations. Invalidated hot pages are likely to be re-referenced quickly, and if so, they will still be in the invalidate window when the re-reference occurs.

5.2.4 Experiment 4: The UNIFORM Workload

The previous experiments have investigated two workloads where invalidation is advantageous and one workload in which propagation performs well. This section briefly examines the UNIFORM workload, where caching is not expected to provide much of a performance benefit due to the absence of locality of access in the workload. This expectation holds true when the UNIFORM workload is run with small (5%) client caches (not shown), as there is little performance difference among the algorithms. Significant differences do arise, however, when the client buffer size is increased to 25%. As shown in Figure 5.13, when the fast network is used, O2PL-I and O2PL-ND perform similarly, followed by O2PL-P and C2PL, while B2PL has the lowest performance. The difference between B2PL and C2PL shows the benefit of exploiting the large client caches — B2PL performs more disk reads, and the disk becomes the critical resource at 5 clients and beyond, in this case. Comparing C2PL with O2PL-I and O2PL-ND, the detection-based C2PL incurs a modest performance penalty due to an effect that was also encountered in the HOTCOLD workload — it has a smaller effective client cache due to the presence of stale data pages in the client caches. This difference can be seen in Figure 5.14, which shows the buffer hit rates at the client caches (solid lines) and at the server (dashed lines). In that figure, C2PL can be seen to have a substantially lower client hit rate (although a small part of the O2PL advantage there is due to

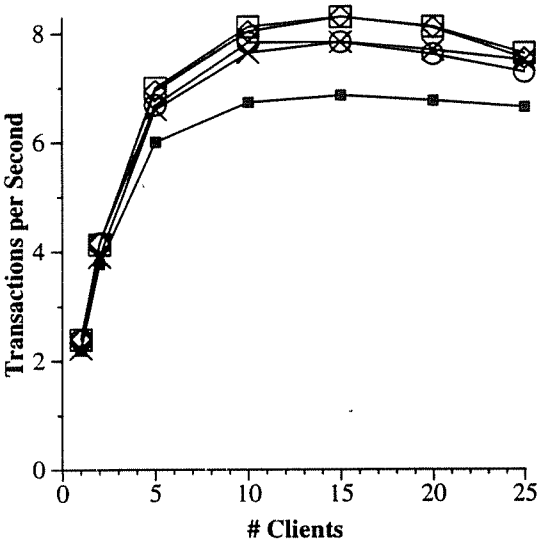


Figure 5.13: Throughput
(UNIFORM, 25% Client Bufs, Fast Net)

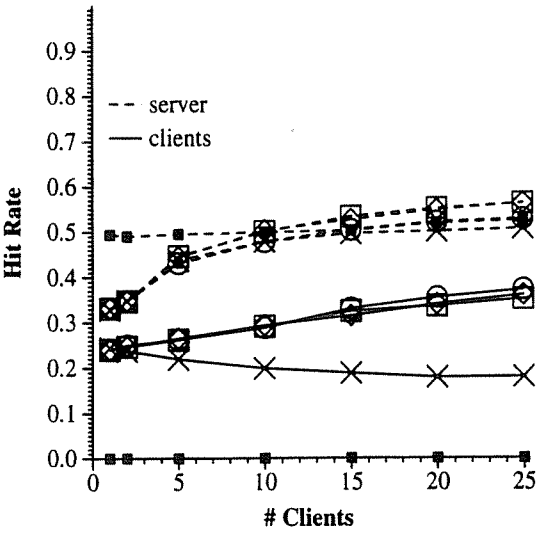


Figure 5.14: Buffer Hit Rates
(UNIFORM, 25% Client Bufs, Fast Net)

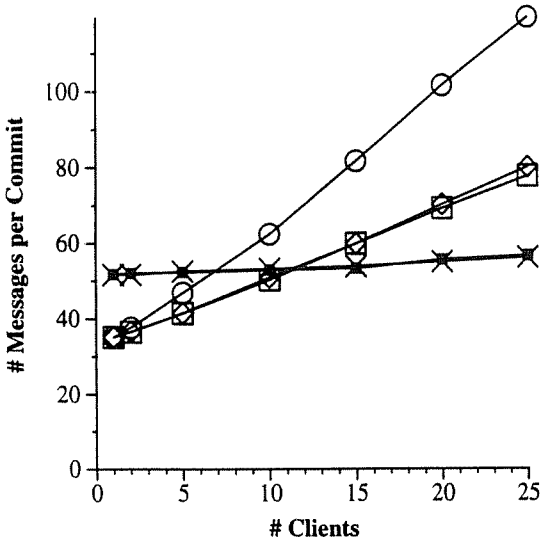


Figure 5.15: Messages Sent/Commit
(UNIFORM, 25% Client Bufs, Fast Net)

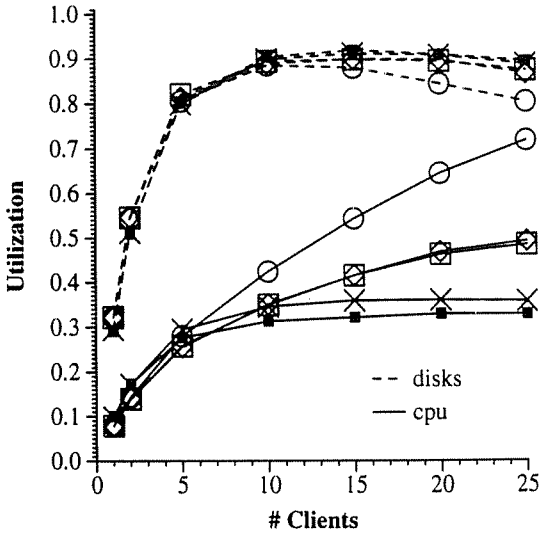


Figure 5.16: Server Resource Utils
(UNIFORM, 25% Client Bufs, Fast Net)

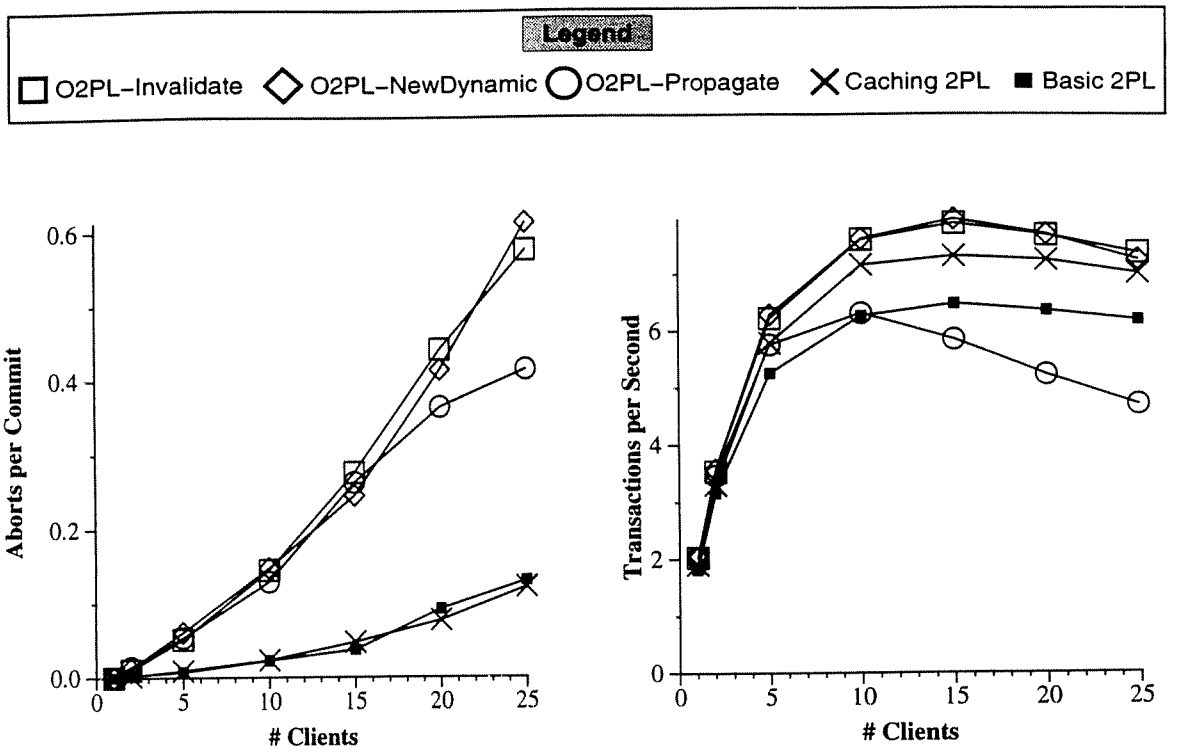


Figure 5.17: Aborts per Commit
(UNIFORM, 25% Client Bufs, Fast Net)

Figure 5.18: Throughput
(UNIFORM, 25% Client Bufs, Slow Net)

restart-induced buffer hits, as is discussed below).

O2PL-ND performs as well as O2PL-I again, as most of its consistency actions are invalidations. Due to the lack of locality in the workload, most pages are not re-referenced quickly after invalidation at a client. O2PL-P performs below the level of the other two O2PL algorithms here, as it sends many more messages (Figure 5.15), resulting in a high penalty in server CPU cycles (shown in the solid lines in Figure 5.16). The large client caches result in O2PL-P sending many propagation messages, with few (under 15%) of those propagations actually being used. Again, due to the lack of locality in the UNIFORM workload a page is likely to be aged out of the cache or updated elsewhere before it accessed again at that site. Therefore, doing fewer propagations results in fewer messages and better performance under this workload.

One interesting aspect of the UNIFORM workload is that it tends to have higher level of data contention than the workloads that were examined previously. In fact, data contention is limiting the performance here somewhat; the throughput of the invalidation-based O2PL algorithms levels off before the server disks become totally saturated as can be seen in Figure 5.16. This increased level of data contention results in an increased number of aborted transactions. Figure 5.17 shows that as data contention is increased by adding clients to the

system, the O2PL algorithms suffer many more aborts due to data conflicts than the detection-based algorithms. The additional aborts are due to O2PL's late discovery of conflicts. In O2PL, conflicts between two updaters cause one of those updaters to be aborted, and read-write conflicts between different clients can result in global deadlocks. Many of these types of aborts can be avoided by C2PL and B2PL by using blocking, and thus the detection-based algorithms have a much lower abort rate. In this case, O2PL aborts are fairly inexpensive though, as transactions that are restarted find that many of their required pages are already in the client cache. Such references are part of the reason that the client cache hit rate can be seen to increase with the client population in Figure 5.14.

Finally, as would be expected, using the slow network in this workload exacerbates the problems of O2PL-P. In this case (shown in Figure 5.18) O2PL-P can be seen to perform even worse than B2PL, which does not perform client caching. This case again emphasizes the potential danger of using propagation indiscriminately, as the benefits gained by caching are overwhelmed here by the additional overhead of propagations.

5.2.5 Summary

The experiments presented in Sections 5.2.1 through 5.2.4 examined the detection-based Server 2PL algorithms and the avoidance-based O2PL algorithms using four different workloads. In general, the O2PL algorithms had the best performance, followed by C2PL, followed by B2PL. Since B2PL purges client caches upon transaction termination, it was used as baseline to gauge the performance gains due to caching. By comparing the performance of C2PL and B2PL, it was shown that the use of the client caches can reduce network utilization, server CPU demand, and disk read requirements, usually resulting in substantial performance gains. Furthermore, the use of caching was never seen to be detrimental. Compared to the O2PL algorithms, C2PL was found to send more messages per transaction due to its need to check the validity of cache-resident pages with the server. These messages hurt the performance of C2PL in the cases where client caching is most beneficial, namely, workloads with high locality and low degrees of read-write sharing, and in configurations with large client caches. Furthermore, in addition to its extra message sends, which were expected, C2PL was found to have a smaller effective client cache size in many cases; the presence of stale pages in its client caches left fewer slots available for valid pages.

Among the O2PL algorithms, invalidation was the best policy except for when the highly directional FEED workload was used. Moreover, propagation was seen to be highly sensitive to the level of sharing and the client cache sizes. In some cases, the overhead of propagations resulted in O2PL-P performing worse than C2PL and even B2PL. Thus, propagation is not a technique on which a general-purpose algorithm should be based. The

dynamic O2PL-ND algorithm, however, was found to equal the performance of O2PL-I in most cases where invalidation was the proper strategy, while successfully switching to propagation under the FEED workload. Thus, O2PL-ND is the best of the O2PL algorithms for handling a wide range of workloads and configurations. One additional consideration, however, is that under higher levels of data contention the O2PL algorithms were seen to have a higher abort rate than the detection-based algorithms due to their optimistic deferral of consistency actions.

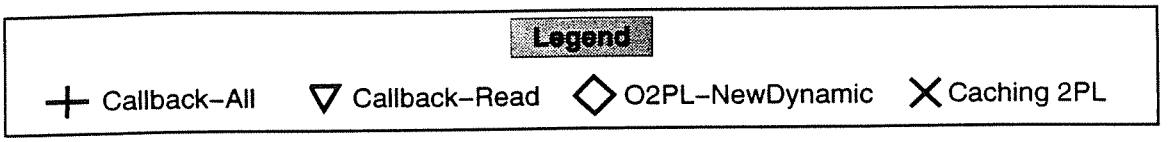
5.3 Callback Locking

This section examines the performance of the third family of algorithms that were presented in Section 4.3 — the callback locking algorithms. The two algorithms that are studied here are Callback-Read (CB-R), which associates write intentions with individual transactions, and Callback-All (CB-A), which associates write intentions with clients so that they can be retained at the client across transaction boundaries. Callback locking is an interesting compromise between O2PL and C2PL, as it is avoidance-based and pessimistic. As seen in the previous section, using avoidance can substantially reduce the number of messages required for cache consistency, while pessimism can cope more gracefully with data contention.

In the experiments that follow, the performance of O2PL-ND and C2PL are also shown for comparison purposes. Experiments were run with client cache sizes of 5% and 25% of the database size. Results are shown only for the 25% case, as the important aspects of the performance of the CB algorithms are slightly more pronounced (but not qualitatively different) with the larger client cache size.

5.3.1 Experiment 5: The HOTCOLD Workload

Figure 5.19 shows the throughput results for the HOTCOLD workload using the slow network and a client cache size of 25%. Of the four algorithms shown, C2PL has the lowest throughput due to the combination of high message requirements (shown in Figure 5.19) and higher disk requirements (compared the others) that result from lower buffer hit rates. In general, the CB algorithms both perform at a somewhat lower level than O2PL-ND, which has the highest throughput overall. These results are driven primarily by the message requirements of the algorithms, as the disk requirements of the two CB algorithms and O2PL-ND are nearly identical. This is because the CB algorithms are invalidation-based, and O2PL-ND chooses to use invalidation for over 90% of its consistency operations. As shown in Figure 5.20, the CB algorithms send significantly more messages per commit than O2PL-ND (although not nearly as many as C2PL). The difference between CB-R and O2PL-ND



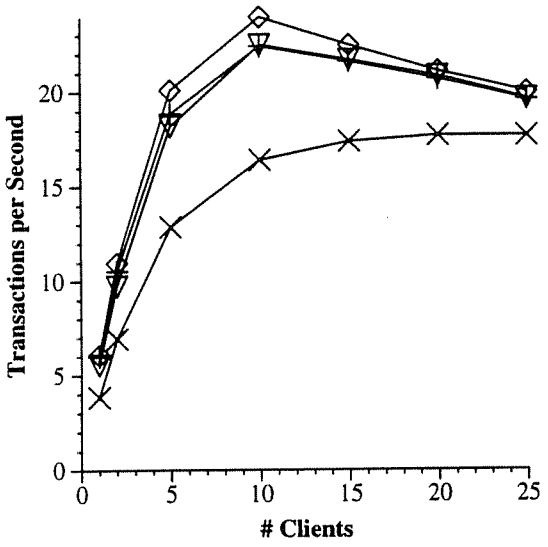


Figure 5.19: Throughput
(HOTCOLD, 25% Client Bufs, Slow Net)

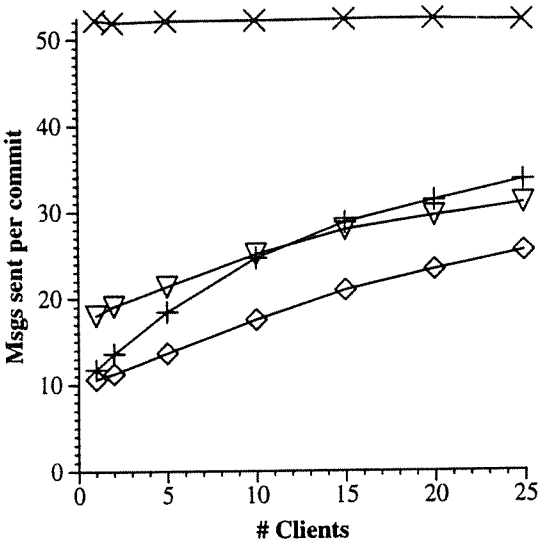


Figure 5.20: Messages Sent/Commit
(HOTCOLD, 25% Client Bufs, Slow Net)

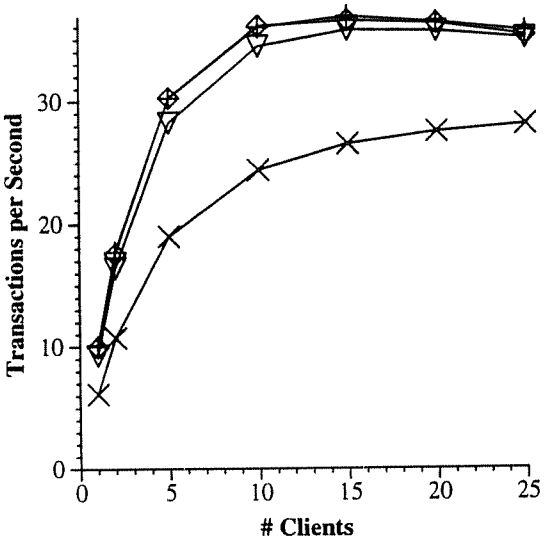


Figure 5.21: Throughput
(PRIVATE, 25% Client Bufs, slow Net)

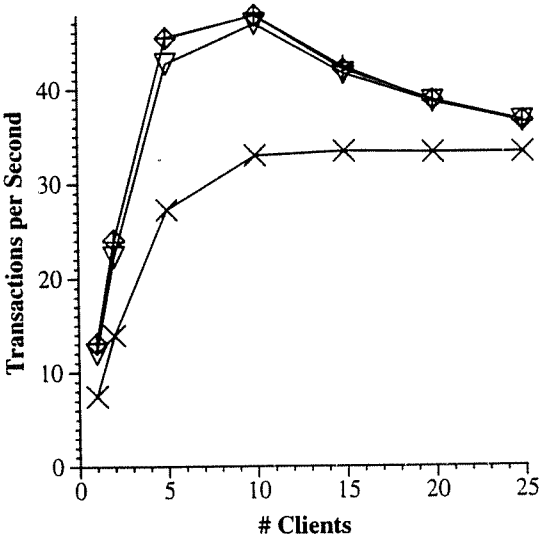


Figure 5.22: Throughput
(PRIVATE, 25% Client Bufs, Fast Net)

is due to the fact that CB-R performs individual consistency operations on a per-page basis, while O2PL-ND performs consistency operations only at the end of the execution phase. O2PL-ND thus saves messages by sending fewer requests to the server for consistency actions (one per transaction, versus as many as one per write for CB-R) and, to a lesser extent, by grouping multiple consistency requests for the same client into a single message. In contrast to CB-R, CB-A is allowed to retain write intentions across transaction boundaries; this is the reason for its sending fewer messages than CB-R when small numbers of clients are present. However, retaining write intentions results in a net increase in messages beyond 10 clients. For the CB algorithms, the advantage of retaining write intention on a page at a client is that a round trip message with the server can be saved if a transaction on that site attempts to write lock the page. However, the penalty for retaining a write intention that ultimately that conflicts with a read request at a different site is also a round trip message with the server. In the HOTCOLD workload, retaining write intentions becomes a losing proposition when the number of clients is sufficient to make it more likely that a page will be accessed at some remote site before it is re-written at a site with a retained write intention. When the fast network is used (not shown), all of the algorithms eventually become disk-bound, at which point the messaging effects just discussed have no impact on the throughput, and CB-R, CB-A and O2PL-ND all perform similarly.

5.3.2 Experiment 6: The PRIVATE Workload

The PRIVATE workload has high locality but no data contention. As a result, avoidance-based algorithms perform very well for this workload. The throughput results for the PRIVATE workload using the slow network are shown in Figure 5.21. The results seen here are again due to the message requirements of the algorithms. C2PL requires an average of 39 messages per commit throughout the range of client populations, while CB-R requires 15 messages and CB-A requires 12 messages. O2PL-ND's message requirements range from 12 messages per commit with 1 client to 13.3 with 25 clients.[8] All of the algorithms except C2PL become network-bound at 10 clients and beyond. C2PL eventually reaches a network bottleneck at 25 clients; it does not reach a network bottleneck earlier because most of the messages it sends are small control messages (e.g., lock requests and replies), which increases the transaction path length due to CPU processing at the server and the clients. The other algorithms send fewer messages per transaction, but many of these messages contain data pages, consuming more of the network bandwidth and making it a larger factor in their path length. C2PL's additional messages in this case are the result of its being detection-based. The performance of CB-R also suffers because it does not

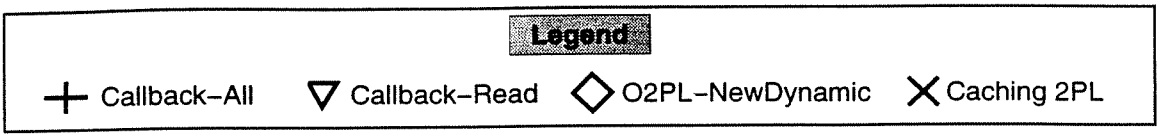
[8]The slight increase in message requirements for O2PL-ND is due to messages for global deadlock detection: during each deadlock detection interval, a round-trip message is sent between the server and every client. In this case, the throughput remains roughly constant as clients are added to the system (beyond 10 clients) so the additional deadlock detection messages increase the per commit total.

retain write intentions, leading it to send a message to the server for every page written by a transaction. When the PRIVATE workload is run with the fast network (Figure 5.22), the C2PL algorithm becomes CPU-bound at the server due to message overhead, while the other algorithms become disk-bound at 15 clients and beyond. Prior to hitting the disk bottleneck, CB-R pays a slight cost due to added path length for obtaining write permission from the server. Note that in this workload, there is no cost for retaining write intentions because of the absence of read-write and write-write data sharing among clients. This provides an advantage for CB-A over CB-R in regions where messages have a significant impact on performance. Once the disk bottleneck is reached, however, CB-R, CB-A, and O2PL-ND all perform similarly.

5.3.3 Experiment 7: The FEED Workload

The reader and writer throughput results for the FEED workload with the slow network are shown in Figure 5.23. O2PL-ND has the best reader throughput prior to 10 clients, while CB-R and CB-A have the best reader throughput beyond 10 clients. The writer throughput is similar for all of the algorithms at 5 clients and beyond. O2PL-ND and CB-A have better writer throughput when there is at most one reader site. Once again, the throughput results are driven by the message requirements of the algorithms. Figure 5.24 shows the number of messages sent per commit averaged over the writer and all of the readers for each of the algorithms.[9] O2PL-ND's higher initial reader throughput is due to its lower message requirements, which result from its reduced consistency message requirements (as described previously) and a better client buffer hit rate due to propagations. All of the algorithms except for C2PL approach a network bottleneck at 15 clients and beyond. As the network becomes saturated, the cost of wasted propagations performed by O2PL-ND cause its reader performance to suffer somewhat, as compared to the callback algorithms. In terms of the callback algorithms, the writer site in CB-R has to register write intentions with the server, while in the CB-A algorithm, the writer has to re-register intentions that have been downgraded (this is virtually all write intentions at five clients and beyond). CB-A also requires extra messages for readers to downgrade the write intentions at the writer site. As a result, CB-A's writer site receives and sends up to 6 times more messages per commit than CB-R's writer site in this case. Thus, callback requests for writes are the cause of the difference in the message requirements for the CB algorithms seen in Figure 5.24. The C2PL algorithm has the highest message requirements for reader sites, as such sites must send a message to the server for each page accessed. Many of these messages are small, so their main impact is an increase in server and client CPU requirements.

[9] Since client number 1 is the writer site, the data points for one client in Figure 5.24 show the messages per writer transaction in the absence of conflicting readers.



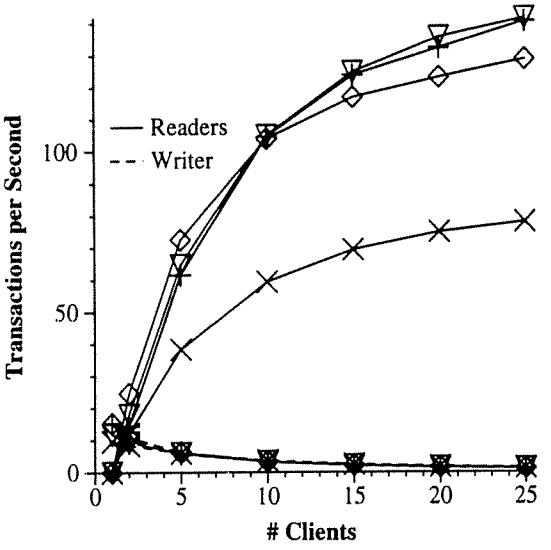


Figure 5.23: Throughput
(FEED, 25% Client Bufs, Slow Net)

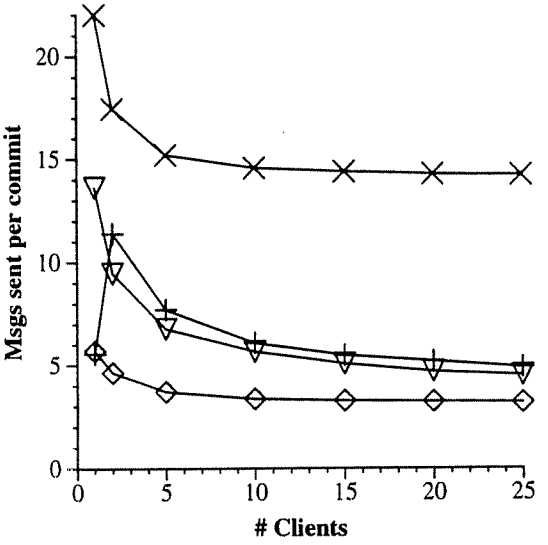


Figure 5.24: Messages Sent/Commit
(FEED, 25% Client Bufs, Slow Net)

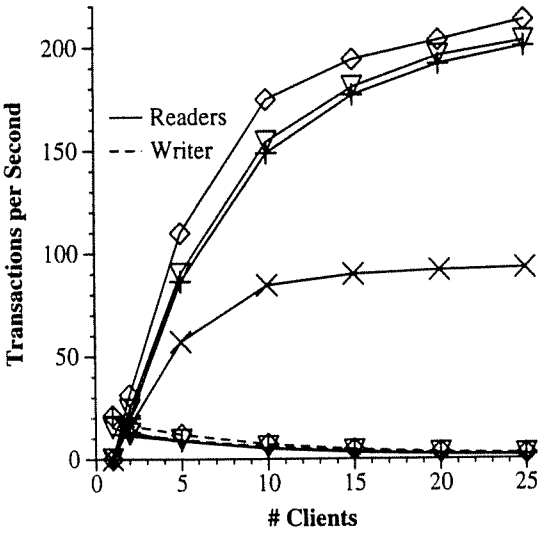


Figure 5.25: Throughput
(FEED, 25% Client Bufs, Fast Net)

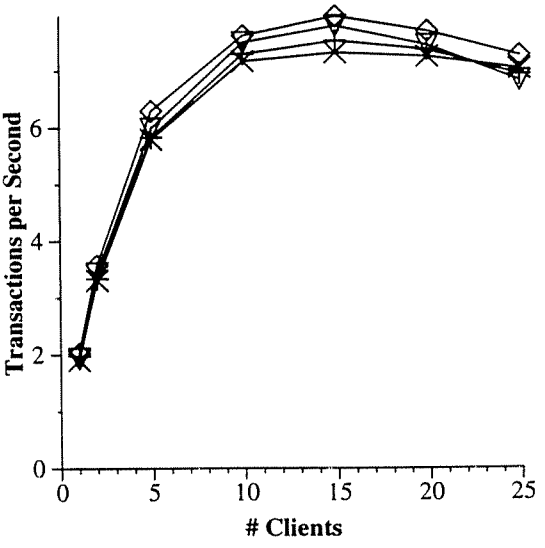


Figure 5.26: Throughput
(UNIFORM, 25% Client Bufs, Slow Net)

The throughput results for the FEED workload with the fast network are shown in Figure 5.25. Again, all of the algorithms except for C2PL approach (but do not quite reach) a disk bottleneck at 25 clients. C2PL eventually becomes server CPU-bound due to lock requests that are sent to the server. The propagations performed by O2PL-ND give it a better buffer hit rate at the reader clients. However, this translates to only a slight reduction in disk reads compared to the callback algorithms, as hot pages missed at clients due to invalidations are likely to be in the server's buffer pool. Thus, in the range of clients studied, the relative performance of the algorithms is still largely dictated by the message characteristics described for the slow network case. With the fast network, however, the size of the messages is less important, and therefore, the relative performance of the algorithms is more in line with the message counts shown in Figure 5.24.

5.3.4 Experiment 8: The UNIFORM Workload

The results for the UNIFORM workload using the slow network are shown in Figure 5.26. None of the algorithms hits a resource bottleneck in the range of clients shown. O2PL-ND achieves the highest throughput across the range of client populations, with the callback algorithms performing below O2PL-ND but better than C2PL through most of the range. CB-R performs slightly better than CB-A because retaining write intentions is costly due to the lack of locality and the low write probability. The decline in throughput for the three avoidance-based algorithms is due to their increased message requirements for consistency operations in this low-locality workload. At 15 clients and beyond, all three of the avoidance-based algorithms actually send more messages than the detection-based C2PL. While C2PL's message requirements per commit remain constant as clients are added, the avoidance-based algorithms are forced to send consistency operations (e.g., invalidations or callbacks) to more sites. C2PL's lower performance throughout most of the range is due to its higher disk requirements, which result from low client and server buffer hit rates. The reason that the algorithms do not quite reach a bottleneck in this case is a higher level of data contention than is present in the other workloads. When the fast network is used (not shown), the network effects are removed and the server disks become the dominant resource. Therefore, the avoidance-based algorithms perform similarly, and are slightly better than C2PL due to buffering characteristics.

5.3.5 Experiment 9: The HICON Workload

The final workload examined in this chapter is the HICON workload. While it is not expected that high data contention will be typical for page server DBMS environments, this workload is used to examine the robustness of the algorithms in the presence of data contention and to gain a better understanding of their different approaches

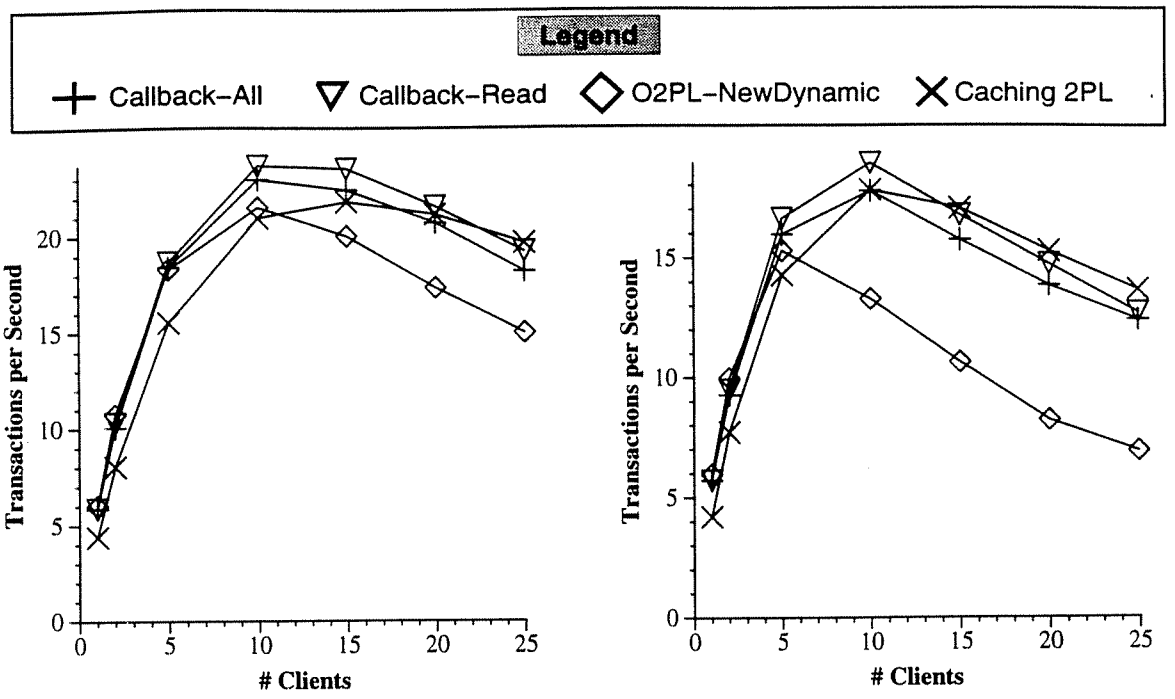


Figure 5.27: Throughput
(5% HICON, 25% Client Bufs, Fast Net)

Figure 5.28: Throughput
(10% HICON, 25% Client Bufs, Fast Net)

to detecting and resolving conflicts. As described in Section 5.1, this workload has a 250 page hot range that is shared by all clients. The write probability for hot range pages is varied from 0% to 50% in order to study different levels of data contention. This section briefly describes the HICON results using the fast network and large client buffer pools. Figure 5.27 shows the throughput for HICON with a hot write probability of 5%. In the range of 1 to 5 clients, the avoidance-based algorithms perform similarly and C2PL has the lowest performance. C2PL's lower performance in this range is due to its significantly higher message requirements here. These results are latency-based, as no bottlenecks develop in this range — in fact, no resource bottlenecks are reached by any of the algorithms in this experiment. At 10 clients and beyond, the effects of increased data contention become apparent; O2PL-ND's performance suffers and it has the lowest utilization of all three major system resources (disk, server CPU, and network). O2PL-ND has a somewhat higher level of blocking for concurrency control than the other algorithms (e.g., at 15 clients, approximately 42% of its transactions are blocked at any given time versus approximately 37% for the callback algorithms and 35% for C2PL). O2PL-ND's blocking level is higher because it detects global deadlocks using periodic detection, while the other algorithms all detect deadlocks immediately at the server. All of the algorithms suffer from increased blocking as clients are added. In addition, the avoidance-based algorithms also incur increased consistency message costs as clients are added.

These two factors account for the thrashing behavior seen in Figure 5.27. Again, retaining write intentions causes CB-A to send more messages than CB-R in this workload. C2PL actually performs best at 25 clients because at that point it sends the fewest messages; C2PL's message requirements remain fairly constant as clients are added to the system, while the other algorithms all incur an increase in messages for consistency operations as clients are added. The slight downturn in performance seen for C2PL beyond 15 clients is due to data contention.

Figure 5.28 shows the throughput of the algorithms when the hot write probability is increased to 10%. Here, the trends seen in the 5% write probability case are even more pronounced. C2PL becomes the highest performing algorithm at 15 clients and beyond, and O2PL-ND performs at a much lower level than the other algorithms. C2PL sends fewer messages per transaction than the other algorithms at fifteen clients and beyond in this case. An increase in data contention causes all of the algorithms to exhibit thrashing behavior beyond 10 clients. In this case, the thrashing is due not only to additional blocking, but also to a significant number of aborts. For example, at 20 clients, O2PL-ND performs nearly 0.6 aborts per committed transaction, while the other algorithms perform about 0.3 aborts per commit. In most of the HICON experiments, O2PL-ND was found to have a significantly higher abort rate than the other three algorithms. This is due to the fact that it resolves write-write and some read-write conflicts using aborts, whereas the other three algorithms can resolve them using blocking. The trends seen in these two cases become even more pronounced as the hot write probability is increased beyond the two cases shown here, and the result is that O2PL-ND performs worse relative to the other algorithms as the hot write probability is increased.

5.3.6 Summary

The results described in Sections 5.3.1 through 5.3.5 show that the CB algorithms have similar, but often slightly lower, performance than O2PL-ND. The CB algorithms typically send more messages than O2PL-ND because they perform consistency maintenance on a per-page basis. In situations where disk I/O is the dominant factor, however, they perform comparably to O2PL-ND. Retaining write intentions across transaction boundaries (as done by CB-A) decreases message traffic in cases with few clients or with low data contention, but results in additional message costs in most other cases. The CB algorithms performed much better than C2PL under most workloads, while retaining the lower abort rate of that algorithm (compared to O2PL-ND). This is because the CB algorithms use avoidance without relying on optimistic techniques. The combination of pessimism and the ability to perform deadlock detection locally at the server allow the CB algorithms to be much more robust in the presence of data contention than O2PL-ND. Under very high data contention C2PL had the best performance because it has a low abort rate and fast deadlock detection (similar to CB) together with message requirements

that remain constant as client sites are added to the system. However, in all other cases, C2PL performed below the level of the CB algorithms and the better of the O2PL algorithms.

5.4 Related Work

The main results that have been presented in this chapter have appeared in two recent papers, [Care91a] and [Fran92a]. As stated earlier, the performance of cache consistency maintenance algorithms for client-server architectures has also been addressed in papers from HP Labs [Wilk90] and Berkeley [Wang91]. The HP work compared the Cache Locks and Notify Locks algorithms (described in Section 4.3.4) to a non-caching algorithm similar to B2PL. The results in that paper showed that the Cache Locks algorithm performed as well as or better than non-caching 2PL in all experiments, whereas the propagation-based Notify Locks algorithm was quite volatile, performing either much better or much worse than the other algorithms depending on the client cache size. This latter result is similar to the results for the O2PL-P algorithm in this study. A major difference, however, is that buffer management was not modeled in [Wilk90]; instead, a specified client buffer hit rate was provided as a parameter to the model. As a result, the study did not show that the reason for the algorithm's erratic behavior was that it was propagation-based, and it did not show the effects of LRU page replacement with small buffer pools. Thus, the need for invalidation-based protocols was not recognized.

The Berkeley work appeared at the same time as [Care91a]. It compared four algorithms, including the two No-Wait Locking algorithms described in Section 4.3.4, a Callback Locking algorithm (similar to CB-R), and a Caching 2PL algorithm. The results of that study showed that if network delay was taken into account, Callback Locking was the best among the algorithms studied when locality was high or data contention was low, with Caching 2PL being better in high conflict situations. These results mostly agree with those presented in this chapter, but there are several differences between the two studies. First, the workload model used in [Wang91] provided a notion of locality that was more dynamic than that of this study, but it was not as flexible in terms of adjusting data sharing among clients. [Wang91] did not study the implications of retaining write intentions across transaction boundaries, and did not examine algorithms such as O2PL-ND, that defer consistency actions until commit. Also, as described previously, the Callback algorithm in [Wang91] used a method for deadlock detection that was susceptible to aborts caused by phantom deadlocks.

No other work on dynamic algorithms for choosing between propagation and invalidation for a client-server DBMS environment has been published. Similar problems have been addressed in work on Non-Uniform Memory Access (NUMA) architectures, however. In such systems, an access to a data object that is located

in a remote memory can result in migrating the data object to the requesting site (invalidation), replicating the object (propagation), or simply accessing the object remotely using the hardware support of a shared-memory multiprocessor. These issues are addressed in Munin [Cart91] by allowing the programmer to annotate data declarations with a designation of the sharing properties of each variable. These annotations serve as hints to Munin in deciding among propagation and invalidation, making replication decisions, etc. DUnX [LaRo91] uses a parameterized policy which chooses dynamically among page replacement options based on reference histories. The parameters allow a user to adjust the level of dynamism of the algorithms. [LaRo91] demonstrated that it was possible to find a set of default parameter settings that provided good performance over a range of NUMA workloads.

A recent effort that is closely related to client-server caching involved using Distributed Shared Virtual Memory (DSM) to support database systems [Bell90]. DSM is a technique that implements the abstraction of a system-wide single-level store in a distributed system [Li89]. Three algorithms for maintaining cache consistency were studied in [Bell90], one of which was a 2PL variant that used callbacks. The callback algorithm was compared to several broadcast-based algorithms using a simulation model that had an inexpensive broadcast facility. Given this facility, the callback-style algorithm typically had lower performance than one of the broadcast algorithms.

A number of papers on shared-disk caching have been written by a group at IBM Yorktown. One of the earlier papers examined consistency protocols that were integrated with the global lock manager of a shared-disk system [Dias87]. A check-on-access (detection-based) algorithm was compared to an algorithm which broadcasts invalidations to *all* nodes in the system. The check-on-access scheme was found to perform better due to the high cost for processing system-wide broadcasts and their acknowledgements. That paper also investigated a the use of a shared "intermediate" memory.

In [Dan90a], an analytic model and a skewed workload (similar to the HICON workload) were used to investigate the effects of skew and contention on the performance of two algorithms: a 2PL algorithm which uses broadcast invalidate and an optimistic algorithm. The study found that under high contention, the invalidation of pages reduces the per-node buffer hit rate, even in the case of restarted transactions. A similar model and workload were used in [Dan90b] to compare four algorithms for cache consistency: a check-on-access algorithm, two invalidation algorithms (one that broadcasts to all sites and one that sends only to the relevant sites), and a check-on-access algorithm that piggybacks information about invalid pages to sites along with messages for lock responses. The study focused on the tradeoff between the cost of messages required to do invalidations and the lower buffer hit rate that comes from allowing stale pages to remain in the buffer. The study found, as did

the work reported here, that the check-on-access scheme has a smaller effective buffer pool than the invalidation schemes.

The issue of hierarchical buffering (i.e., private and shared buffers) was addressed in [Dan91]. This study investigated policies for determining which pages to place in the shared buffer. Three types of pages were identified: 1) pages that were recently missed in both shared and private buffers, 2) pages that were recently updated in a private buffer, and 3) pages that were recently replaced from a private buffer. Algorithms that favored each of these page types, as well as combinations of page types were proposed. The paper primarily discussed the model and provided only a brief comparison of the policies themselves.

Finally, callback-style algorithms for the shared-disk environment have been investigated in [Dan92]. An algorithm that was similar to CB-R (i.e., it did not allow write intentions to be retained) was shown to perform relatively well. In that study, the performance differences seen among the algorithms were largely due to the disk I/Os needed to force pages to stable storage for recovery purposes prior to transferring a page from one node to the next. This is a different problem than what is encountered in a page-server system, as such systems have a central server that is responsible for managing a log, and in the algorithms covered here, all page transfers go through that server.

As mentioned earlier, callback algorithms were initially developed for use in distributed file systems. The Andrew File System [Howa88] uses a callback scheme to inform sites of pending modifications to the files that they have cached. This scheme does not guarantee consistent updates, however. Files that must be kept consistent, such as directories, are handled by simply not allowing them to be updated at cached sites. Sprite [Nels88] provides consistent updates, but it does so by disallowing caching for shared files that are open concurrently with at least one of the opens being for writes. Caching is disallowed using a callback mechanism that informs sites that a file is no longer cachable.

5.5 Conclusions

In this chapter, seven algorithms from three families of cache consistency maintenance algorithms have been examined and compared. The use of client memory for inter-transaction caching was seen to provide substantial performance benefits in many cases, and it did not cause performance to degrade compared to not caching except under very high data contention or when propagation was used indiscriminately.

In terms of the tradeoffs between the detection-based Server 2PL algorithms and the avoidance-based O2PL algorithms, the message savings obtained by deferring consistency actions for the O2PL algorithms was seen

to provide performance advantages over the detection-based C2PL algorithm in those situations where caching was the most beneficial, namely, high locality with minimal read-write sharing of data. Among the avoidance-based O2PL algorithms, the invalidation of remote copies was found to provide the best performance in most cases examined. Propagation, however, was found to have significant performance advantages under a very specialized information feed workload, but also displayed a high degree of volatility, especially with regard to client cache sizes. This volatility results in propagation being unsuitable for a general-purpose environment. Finally, a dynamic variant of O2PL (O2PL-ND) was found to match the performance the invalidation-based O2PL algorithm in the cases where it had the highest performance, and it also performed well in the cases where propagation was the right approach to take.

Two variants of Callback Locking were described and studied: CB-R, which associates write intentions with individual transactions, and CB-A, which allows write intentions to be retained by clients across transaction boundaries. CB-R was found to perform as well as or slightly better than CB-A except in cases with small client populations or minimal data contention. In most cases, the differences among them were not large. However, the results of the study indicate that a fairly simple heuristic could be used to decide when to allow clients to retain write intentions. A possible heuristic is one in which the server assigns a write intention to a requesting client (rather than a transaction) only if no other clients have cached copies of the page at the time of the declaration. Such a heuristic would be able to efficiently detect pages that are not shared (as in the PRIVATE workload), but is also self regulating; if sharing arises, the intention will be downgraded and given only to the requesting transaction upon a subsequent declaration.

Both of the CB algorithms were seen to have slightly lower performance than O2PL-ND (which typically sent fewer messages) in situations where network usage plays a large factor in determining performance, but their performance was similar to that of O2PL-ND in cases where disk I/O was the dominant factor. However, the CB algorithms were also seen to have a lower abort rate than O2PL-ND, and they were much more robust than O2PL-ND in the presence of data contention. As stated earlier, a low abort rate is particularly important in a workstation-based environment. Finally, C2PL, which is detection-based, was found to be the best algorithm for workloads with very high data contention, but it performed below the level of the CB algorithms and O2PL-ND in all other cases. Because of its combination of overall good performance, a low abort rate, and relative ease of implementation, the CB-R algorithm is used as the basis for the work reported in the remainder of the thesis.

Chapter 6

Global Memory Management

The preceding two chapters were concerned with exploiting client memory by caching in order to reduce dependence on the server. This chapter takes client caching a step further by treating the aggregate memory of the clients in the system as a global cache.

6.1 Introduction

As shown in the previous chapter, client caching can provide substantial performance benefits over a wide range of workloads and configurations. For example, in nearly every case studied the non-caching B2PL algorithm performed well below the level of the caching algorithms. Due to its heavy reliance on the server for data pages, B2PL becomes bottlenecked at the server with smaller client populations than the other algorithms. This dependence on the server reduces B2PL's scalability as well as its performance. In contrast, the algorithms that allowed inter-transaction caching were often able to obtain significant performance and scalability benefits due to their ability to access pages from the local client memory. Despite this advantage, however, the performance and scalability of the caching algorithms were also often eventually limited by excessive demands for pages from the server.

From the point of view of a client, caching extends the amount of the database that can be accessed without requiring server disk I/O to include not only the server's buffer pool, but also the client's memory as well. In all of the algorithms for client caching described and surveyed in Chapter 4, each client used a three-level memory hierarchy consisting of: 1) the local workstation's memory, 2) server memory, and 3) server disk. Thus, each client workstation had access to only a fraction of the total memory in the system. As a result of this, the experiments of the previous chapter showed that in many cases, significant disk I/O was required — even though

the aggregate memory of the system was as large or larger than the portion of the database being accessed by the workstations.

Furthermore, two additional inefficiencies of this type of data caching were identified. First, when small numbers of workstations were present, there was often a high correlation between the pages resident in the server buffer pool and those resident in the client caches. This correlation reduced the effectiveness of the server buffer pool, as buffer misses at clients often resulted in buffer misses at the server. Secondly, it was shown that with large numbers of clients, each with a fairly large buffer pool, excessive replication of pages in client caches could lead to significant overhead for updates. In some of the cases examined, this overhead even outweighed the performance gains of inter-transaction data caching.

While the client caching algorithms studied in Chapter 5 were effective in many situations, they were ultimately limited by their primarily *local* nature. Performance was hindered because clients were unable to exploit a large portion of the memory available in the system and because the memory that was available was not efficiently utilized. However, in all of the avoidance-based cache consistency algorithms studied, which typically performed better than the detection-based ones, the server was required to have knowledge of the location of all copies of pages in the system. This information provides an opportunity to improve upon the previous techniques through the use of a *global* approach to memory management. This chapter investigates the tradeoffs involved with three specific global memory management techniques. First, the clients are allowed to exploit the entire memory of the system by obtaining pages from other clients. Second, buffer replacement policies at the server are modified to reduce the replication of the buffer pool contents of the server and its clients. Third, a simple protocol between clients and servers is used to extend the client caches by moving some of the pages that are forced out of a client's cache into the server's memory.

In the following sections, a set of algorithms that utilize these three techniques are proposed and analyzed. The implementation of these algorithms requires no additional information at clients and servers beyond what is already required for the avoidance-based caching algorithms described previously. In particular, these global memory management algorithms are designed to tolerate imperfect (but conservative) server knowledge of page copy locations. After describing these algorithms and the environment in which they are to be implemented, the results of simulation experiments that investigate the performance characteristics and tradeoffs of the three global memory management techniques are presented.

6.2 Algorithms for Global Memory Management

This section presents three techniques for implementing global memory management in a page server environment and outlines algorithms that use different combinations of these techniques. Also provided is an overview of the expected performance tradeoffs among the different techniques.

As stated previously, the caching algorithms of the previous two chapters all implement a three-level memory hierarchy, thus limiting the size of the memory from which to satisfy the page requests of any one client to a fraction of the total memory available to the database system. The goal of the global memory management techniques investigated in this study is to exploit the remaining fraction in an *opportunistic* way. That is, as in the previous caching algorithms, the contents of any one client's memory are dictated by the accesses made by that client, but in addition, those contents (if not exclusively locked) can also be sent to other clients to satisfy their local cache misses. The techniques also attempt to make better use of the server memory in light of this new capability.

The ability to exploit the contents of remote client memory results in a four-level memory hierarchy as shown in Table 6.1. The level closest to the client is the *local client memory*, which can be directly accessed

Rank	Level	Small Messages	Large Messages	Disk I/Os
1	Local Memory	0	0	0
2	Server Memory	1	1	0
3	*RemoteMemory*	2	1	0
4	Server Disk	1	1	1 or 2

Table 6.1: Memory Hierarchy Costs

by a client database process. The second level of the hierarchy is the *server memory*, which is managed by the server database process. In terms of response time, this memory costs one small message from the client to the server (for the page request) and one large message (containing the page itself) from the server to the requesting client. Messages incur costs not only for their actual on-the-wire time, but also for CPU processing at both the sender and the receiver. The third level of the hierarchy is *remote client memory*. The server is the only site with knowledge of where page copies are cached in the system, so access to this level of the hierarchy must go through the server. Therefore, access to remote client memory costs two small messages and one page-sized message: the client first sends a small message to request the page from the server, the server then forwards that request to another client, and the remote client sends a large message containing the page to the requesting

client.[1] Finally, the fourth level of the hierarchy is the server's disk. An access to this level of the hierarchy is the most expensive, costing one small message and one page-sized message as well as one or two disk accesses. (Two disk accesses are required if a dirty page must first be forced from the server's buffer pool in order to make room for the requested page to be read in from disk). In general, the goal of the global memory management techniques studied here is to move accesses from the lowest (and most expensive) level of the hierarchy to the higher levels. In particular, the techniques will attempt to convert what would have been disk accesses in a non-global scheme into cheaper accesses to the server memory or to remote client memories.

Up to this point, the discussion of memory management issues has been concerned primarily with the use of memory to avoid disk reads. However, in order to provide durability for the updates of committed transactions, the pages containing these updates must eventually be written to stable storage. In a page server system, the server is responsible for ensuring the durability of committed updates and also for ensuring that all sites see a transaction-consistent view of the database. The server implementation can be simplified if the server always has the most recent committed copy of a page (either in its memory or on disk). This can be achieved by requiring all pages dirtied by a transaction to be copied to the server before the transaction is allowed to commit. Dirty pages that are copied back to the server have two conflicting characteristics that complicate the buffer replacement policy at the server. On one hand, reclaiming a dirty page's buffer slot requires an I/O to write the page to disk, so keeping a dirty page in the buffer longer can reduce I/O by combining multiple writes to the same page into a single disk write [Chen84]. On the other hand, many of the dirty pages present in the server's buffer pool may not actually be valuable pages, as their placement in the server buffer pool is based on considerations other than their probability of being accessed; the presence of such dirty pages may result in additional disk reads for other pages that are relatively hotter. As will be seen in later sections, these conflicting characteristics will have an effect on the performance of the different global memory management techniques.

6.2.1 Three Global Memory Techniques

As stated previously, this study concentrates on three related global memory management techniques. These techniques were chosen on the basis of their potential for improving performance versus the complexity of their implementation in a page server environment. The three techniques are outlined below and then presented in more detail in the following section. The techniques are:

[1] We do not require perfect knowledge of page copy locations at the server, and therefore, there is a small possibility that the remote site will not be able to forward the requested page. Handling of this situation results in an extra message and possibly a disk access, as will be explained in Section 6.2.

Forwarding - The main technique investigated in this study is to allow a request for a page that is not in the server's buffer pool to be forwarded to a remote client if that client has a copy of the page in its buffer pool. Upon receipt of a forwarded request, the remote client sends a copy of the page directly to the requesting client. The goal of this technique is to reduce disk I/O by extending the amount of memory available to satisfy client page requests. This technique has the highest potential for performance improvement of the three studied, but also requires the most modification to existing data caching algorithms.

Hate Hints - Hate hints are a simple heuristic that can help to keep a larger portion of the database available in memory when the forwarding technique is in use. When the server transfers a page to a client, the server marks that page as hated (i.e., it makes it the "least recently used" page in its buffer pool). The page will then be likely to be replaced when a buffer frame is needed for a new page. This heuristic is an attempt to reduce page replication between the buffer contents of a server and its clients, thereby allowing a larger number of distinct pages to reside in the global memory. When a page is transferred to a client, it is known that the page is in memory elsewhere in the system, and thus, the copy at the server does not contribute to the percentage of the database available in global memory.

Sending Dropped Pages - This technique attempts to use the server buffer pool to prevent a page from being completely dropped out of the global memory. With this technique, a client informs the server when it intends to drop a particular page from its cache by piggybacking that information on a page request message it sends to the server. If the server determines that the copy to be dropped is the only copy of the page that resides in global memory, it asks the client to send it the page when it is replaced from the client's cache.

6.2.2 Algorithm Descriptions

This section describes four memory management algorithms that will be used to compare the effectiveness of the global techniques under different workloads and system configurations. The four algorithms are extensions of the Callback-Read (CB-R) algorithm that was described in Section 4.3.3 and analyzed in Section 5.3. Each algorithm extends CB-R with the forwarding technique along with none, one, or both of the other two global techniques.

Recall that in CB-R the server must keep track of all page copies located at clients in order to call them back in the event of a write access request on the page. It is this copy information that is exploited to produce the global memory hierarchy. Rather than send a message to the server each time it replaces a page, a client simply piggybacks the page numbers of any pages it has dropped on the next message that it sends to the server. As a result of this mechanism, the server's copy information is slightly conservative in that there is a window during

which the server may have an entry for a page copy that has just been dropped from a client. This conservatism does not affect correctness, but may result in an occasional unnecessary callback request. Also recall that with CB-R the server buffer is managed using an LRU policy. Pages become the "most recently used" (i.e., least likely to be replaced) page when they are accessed to be sent to requesting clients. Dirty pages that are copied back to the server by committing transactions are marked as most recently used when they arrive at the server.

The Forwarding Algorithm (FWD)

The first global algorithm, FWD, is simply the callback algorithm extended with the forwarding technique described in Section 6.2.1. When the server receives a request for a page (and hence, an implicit request for a read lock) from a client, it first it obtains a read lock on the page for the requesting transaction. Once the lock has been obtained, it checks to see if the page is in its local buffer pool and if so, it sends a copy of the page to the requester. If the page is not in the server's buffer pool, it checks to see if the page is cached at another client and if so, forwards the page request to a remote client that has a copy of the page. When a client receives a forwarded request, it checks to see if it has a copy of the page that it can send to the requesting client, and if so, sends it. A client cannot forward a page if it no longer has that page cached or if it is in the process of trying to obtain a write lock on the page from the server.[2] If the client can not forward the page, it returns the request with a negative acknowledgement to the server. If there are no sites that have copies of the page or if the server receives a negative acknowledgement from a remote client, it reads the page into its buffer pool from disk (as is done for all server buffer misses in the CB-R algorithm) and sends it to the requesting client.

An example of the use of forwarding is shown in Figure 6.1. In the example, client 1 requests page C from the server (message 1). The server checks its local buffer pool and finds that the page is not in memory. It then checks its copy information and learns that client 2 has a copy of the page. It sends client 2 a message, asking it to send a copy of the page to client 1 (message 2). If client 2 still has the page locally, and is not in the process of trying to update the page, then it sends a copy of the page to client 3 (message3). Thus, client 1 was able to satisfy its page request without the need to perform a disk I/O at the server.

Forwarding with Hate Hints (FWD-H)

The FWD-H algorithm is a simple extension of the FWD algorithm that uses the hate hints technique. The algorithm works similarly to FWD except that when the server sends a page to a client the page becomes "hated"

[2] In the forwarding technique, there is a brief window during which a client may request a write lock on a page at the same time the server is sending it a forwarded request for the page. In this situation, the write lock request will be blocked at the server; the forwarded read request will be rejected by the client and will be satisfied by a disk I/O at the server.

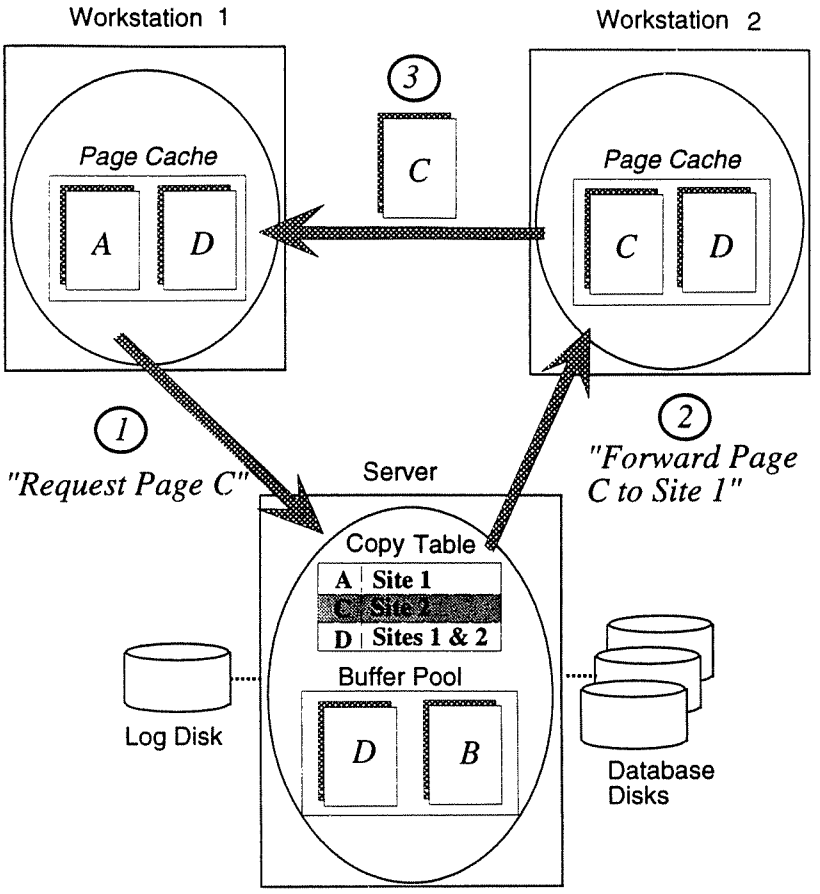


Figure 6.1: Forwarding Technique Example

(i.e., it is marked as the current "least recently used" page) at the server, making it likely to be replaced from the server's buffer pool. Using the LRU mechanism to implement hate hints has two effects: 1) a non-hated page will never be aged out of the server's buffer pool while the buffer pool contains any hated pages, and 2) hated pages are aged out in a LIFO manner.

As described previously, transactions send their dirty pages to the server when they commit. When a dirty page arrives at the server, it is marked as the most recently used page. If a page is present in the buffer pool when a dirty copy of the page arrives at the server, the dirty copy replaces the prior copy in the buffer pool, and it becomes the most recently used page. Conversely, if a page that is marked as dirty in the server buffer pool is sent to a client, it becomes a hated (and still dirty) page.

Forwarding with Sending Dropped Pages (FWD-S)

The next algorithm, FWD-S, is an extension of the FWD algorithm in which clients send some of the pages that they drop to the server. This algorithm takes advantage of the message patterns inherent in the baseline CB-R algorithm. When a client determines that it needs to request a page from the server, it also checks to see if the new page will force an existing cached page out of its cache. If so, the client piggybacks the page number of the page it plans to drop on the request message that it sends to the server. When the server receives such a page request, it checks to see if the page to be dropped is the only copy of the page that is currently in the global memory. If so, the server sets a flag in the message that it uses to respond to the page request; this flag informs the client that it should send the page (asynchronously) back to the server rather than simply drop it. When the dropped page arrives at the server, it is marked as the most recently used page.

There are two additional cases that the algorithm must handle. First, if the server forwards the request to a remote client, the remote client must forward the server's send-back decision to the requester along with the page. The second case occurs when the server determines that it will have the only remaining memory-resident copy of the page once the requester drops its copy. In this case, the server marks its copy of the page as most recently used and informs the client that it need not send the dropped page.

Forwarding with Hate Hints and Sending Dropped Pages (FWD-HS)

The final global algorithm is the FWD algorithm extended with both the hate hints and sending dropped pages techniques. It is simply the combination of the FWD-H and FWD-S algorithms.

6.2.3 Performance Tradeoffs

The previous sections described three techniques for improving performance through global memory management and presented algorithms that use these techniques to extend the CB-R algorithm, which uses only local memory management. Before presenting the detailed results from the simulation study of these algorithms, it will be useful to consider the expected performance tradeoffs among them. CB-R, the baseline algorithm, does not exploit remote client memory and must therefore rely only on the local client memory, the server memory, and disk. The FWD algorithm uses messages and some extra client CPU processing in an attempt to avoid doing disk I/O on server buffer misses. The FWD-H algorithm attempts to further reduce disk I/O by avoiding replication between the contents of the server and its clients, thus increasing the portion of the database that is available in memory. The FWD-S algorithm also tries to replace disk I/O by messages; it attempts to increase the portion of

the database retained in memory by sending a copy of a page to the server rather than dropping it, if that copy is the only one resident in global memory. In comparing the FWD-H and FWD-S algorithms, it can be noted that the hate hints and sending techniques have similar goals in that both try to increase the portion of the database that is available in memory. Hate hints is an indirect approach, which tries to accomplish its goal by reducing replication. In contrast, the sending technique is a more direct approach, as the system actively tries to keep pages from being dropped from the global memory. Finally, FWD-HS combines all of these techniques and, if the benefits of the hate hints and sending techniques are additive, should keep even more of the database in memory than the other algorithms.

6.3 Experiments and Results

This chapter presents the results of a simulation study of the global memory management algorithms described in Section 6.2. In order to perform the study, the client-server caching model described in Section 3 was extended to include the four algorithms. This was relatively straightforward, as the the algorithms are designed to require only minimal changes to the CB-R algorithm.

As in the previous chapter, the primary performance metric employed in this study is the throughput (i.e., transaction completion rate) of the system. A number of the additional metrics that were used in the previous study are also used here to aid in the analysis of the experimental results, including the server buffer hit rate, the client and server resource utilizations, the average number of messages required to execute a transaction, and several others. One special metric that is used in this study is the "database portion available in memory". This is the percentage of the pages of the database that are available to a client without performing a disk I/O. For the forwarding algorithms, this metric is the union of the contents of the server buffer pool and all client caches, whereas for the callback algorithm it is the union of the server buffer pool contents and the contents of only a single client. The various metrics that are presented on a "per commit" basis are computed by dividing the total count for the metric by the number of transaction commits over the duration of a simulation run.

The performance of the global memory management algorithms is examined under the HOTCOLD, PRIVATE, and UNIFORM workloads that are described in Section 5.1.2 and Table 5.2 of the previous chapter. In addition to these workloads, results are also presented for a read-only version of the HOTCOLD (called RO-HOTCOLD) in order to examine the memory management aspects of the algorithms in a simplified setting. All the physical resource and overhead parameter settings except for the server buffer pool size and the client cache sizes are the same as those used for the cache consistency study, as discussed in Section 5.1 and listed

in Table 5.1. In this set of experiments, the server is configured with a buffer pool capable of holding 30% of the database (in contrast to 50% in the caching study). Two sizes of client caches are tested: 5% (as before), and 15%. The smaller server and client memory sizes here (compared to those of the previous study) allow a finer-grained analysis of the global memory algorithms, as the aggregate memory scales in smaller increments with the number of clients.

6.3.1 Experiment 1: Read-Only HOTCOLD Workload

The first set of results to be examined uses a version of the HOTCOLD workload that performs no updates. Although such a read-only workload is not expected to be common, we analyze it first in order to examine the buffering behavior of the various algorithms in the absence of the complications that are introduced by dirty pages. Recall that in the HOTCOLD workload, (as shown in Table 5.2), each client has its own 50 page region of the database to which 80% of its accesses are directed. The hot region of one client is contained in the cold regions of all other clients, so there is substantial sharing of pages in this workload in addition to high per-client locality. Since this version of the workload is read-only, however, there are no data conflicts.

Read-Only HOTCOLD, Small Client Caches

The aim of each of the global memory management techniques is to reduce the need for disk I/O by increasing the portion of the database that is available in memory. However, there are two reasons why such an increase may not translate into a performance improvement: 1) the resources used to increase the portion of the database available in memory may be more expensive than the resources saved by the increase, and 2) in a skewed workload such as RO-HOTCOLD, some pages are more valuable than others, so a higher portion of the database available in memory does not necessarily imply a reduction in disk I/O. In the following, we first compare the algorithms based on the portion of the database that they keep available in memory, and then examine the resulting resource demands. Finally, we examine how these demands translate into throughput, given the system parameters of Section 5.1.

Portion of Database Available In Memory

Figure 6.2 shows the percentage of the database available in memory for each of the algorithms when running the Read-Only HOTCOLD workload with small client caches (5% of the database size). The dotted line shows the highest in-memory percentage that could be obtained ideally (based on the amount of memory in the system). Algorithms typically have less than the ideal amount of the database in memory due to replication among the

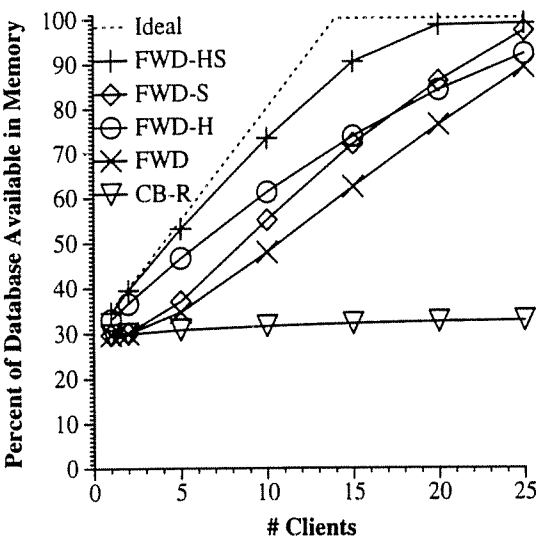


Figure 6.2: % of DB Available in Memory
(RO-HOTCOLD, 5% Client Bufs)

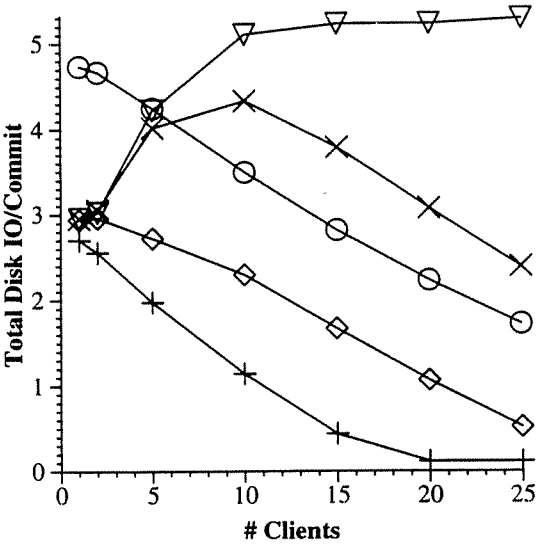


Figure 6.3: Total Disk I/O per Commit
(RO-HOTCOLD, 5% Client Bufs)

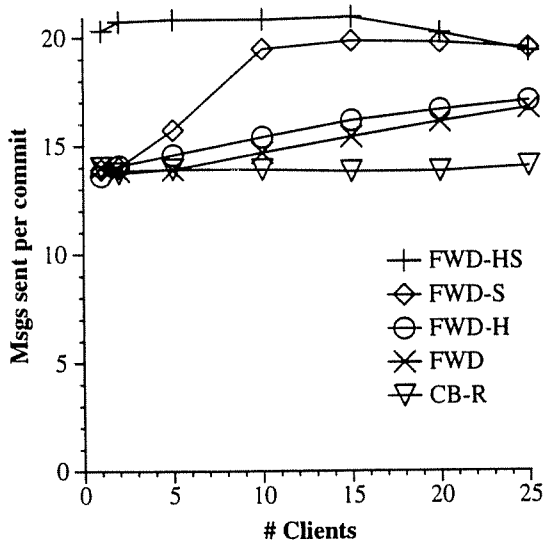


Figure 6.4: Messages Sent/Commit
(RO-HOTCOLD, 5% Client Bufs)

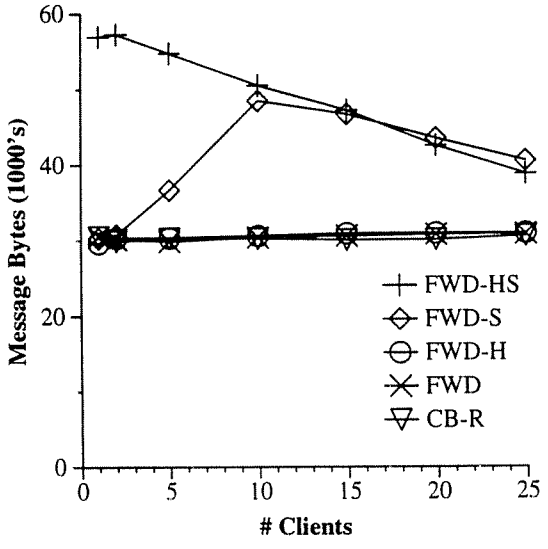


Figure 6.5: Message Volume/Commit
(RO-HOTCOLD, 5% Client Bufs)

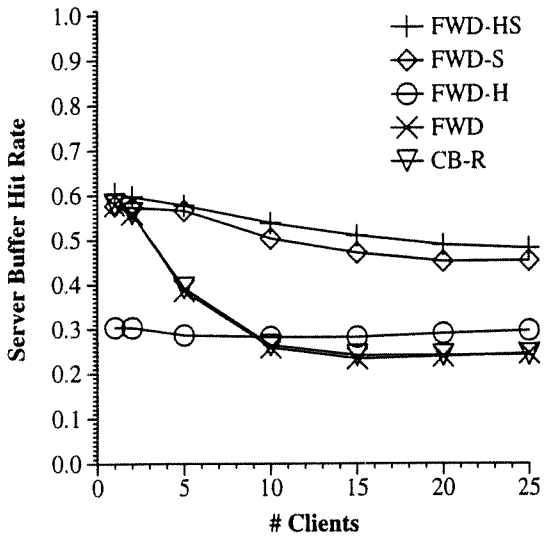


Figure 6.6: Server Buffer Hit Rate
(RO-HOTCOLD, 5% Client Bufs)

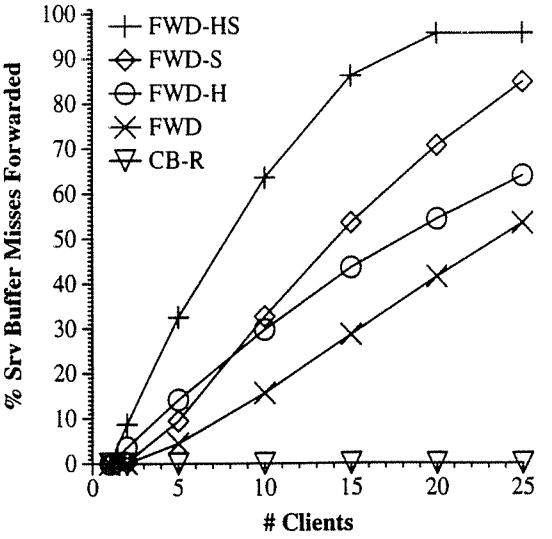


Figure 6.7: % Server Misses Forwarded
(RO-HOTCOLD, 5% Client Bufs)

contents of the system's buffers pools. There are two types of replication that can arise: server-client correlation, and client-client replication. Server-client correlation can arise when the server and the client buffer managers use the same page replacement policy (LRU). In this situation, when the ratio of the number of pages resident in the server buffer pool to the number of distinct pages that are resident in client caches is high (e.g., greater than one), a page that is in a client's cache is also likely to be in the server's buffer pool. Server-client correlation is most prominent with small client populations. As clients are added to the system, the ratio of pages at the server to pages at the clients becomes smaller, and as this ratio decreases, the server can replicate fewer of the pages that are kept at clients.[3] Client-client replication arises from overlapping client requests. The amount of client-client replication increases as clients are added to the system.

Turning to Figure 6.2, we first note that CB-R has the smallest portion of the database available in memory. CB-R does not use forwarding, so the addition of clients does not increase the amount of memory that can be used to service a particular client's requests. CB-R has a slight increase the percentage of the database it has available in memory as clients are added to the system, which is due to the reduction in server-client correlation. CB-R is unaffected by client-client replication, as each client has access only to the contents of the server's buffer pool and its own cache. In contrast to CB-R, the forwarding algorithms can capitalize upon the caches brought to the system by additional clients and thus, they all show a significant increase in the portion of the database available in memory as clients are added. However, as seen in Figure 6.2, none of the forwarding algorithms

[3] In this experiment the aggregate size of the client caches becomes larger than the size of the server buffer pool when more than 6 clients are in the system.

are able to attain the ideal in-memory percentage. The forwarding algorithms are affected by both types of replication described above. However, all of the forwarding algorithms incur the same level of client-client replication, as the global memory management techniques they use do not alter the buffering behavior at clients. Therefore, the differences among the forwarding algorithms shown in Figure 6.2 are the result of differences in their server-client correlation.

FWD, the simplest forwarding algorithm, initially shows very little improvement over CB-R. This is due to the correlation between server and client buffers — the contents of the additional client caches are replicated in the server's buffer pool. As clients are added, the impact of the server-client correlation decreases, until at 25 clients, FWD has access to almost 90% of the database in memory. The other three forwarding algorithms employ techniques that attempt to increase the percentage of the database available in memory. Compared to FWD, the FWD-H algorithm attains a relatively high in-memory percentage with small numbers of clients. This is because hate hints manage to reduce the server-client correlation. However, beyond 15 clients FWD-H's advantage over FWD begins to dissipate; at 25 clients, FWD-H is only slightly better than FWD. This is because as clients are added to the system, the fraction of FWD-H's page requests that are serviced by forwarding increases. Requests serviced by forwarding do not go through the server's buffer pool, so the hate hints become less effective at reducing the server-client correlation.

FWD and FWD-H both have somewhat less than the ideal in-memory database percentage at 25 clients. FWD-S, on the other hand, comes close to having the entire database in memory at this point. The success of FWD-S is due to its effective use of the server buffer pool — it uses the server buffer pool to retain pages that would otherwise have been dropped from the global memory. However, with small client populations, the sending technique has little effect. When server-client buffer correlation is high, pages that are aged out of client caches are likely to be in the server buffer pool, so few dropped pages are sent to the server. The sending technique is quite effective for larger client populations but less effective for smaller populations, while the hate hints technique has the opposite characteristics. For this reason, the available in-memory percentages for the two algorithms eventually cross.

FWD-HS, which combines the hate hints and sending techniques, keeps the largest portion of the database available in memory throughout the range of 1 to 25 clients. At 25 clients, FWD-HS has almost 100% of the database available in memory. The interaction of the two techniques is effective throughout the range of client populations, as it tends to keep a copy of a page in memory at either a client or at the server, but not at both.

Resource Requirements

We now turn our attention to the resource requirements of the five global memory management algorithms. As expected, the general trend is that in most cases, an increase in the percentage of the database available in memory results in a decrease in disk I/O (since more requested pages are found in memory) and an increase in messages (for serving such requests and for managing the contents of global memory). Figure 6.3 shows the total number of disk I/Os per committed transaction (in this workload, all disk I/Os are reads) for the various algorithms. The message requirements are shown in Figure 6.4, which shows the average number of messages sent per committed transaction, and Figure 6.5, which shows the total number of message bytes sent per committed transaction. The latter two metrics can differ because some messages are control messages (256 bytes), while other messages contain one or more 4K byte pages. The message and disk I/O requirements for transactions depend on the client buffer hit rate (not shown), the server buffer hit rate (shown in Figure 6.6), and the percent of server misses that are forwarded to other clients (shown in Figure 6.7). The sending algorithms also incur additional page-sized messages for sending dropped pages back to the server. All of the algorithms have the same client buffer hit rate (slightly over 65%) because they are all based on CB-R and because the global techniques do not affect buffering at the clients. As a result, for all of the algorithms, clients send the same number of page requests to the server. Overall, CB-R sends the fewest messages per commit, because it never forwards requests to other clients. For the same reason, its disk I/O requirements are inversely proportional to its server buffer hit rate. CB-R initially suffers a steep decrease in the server hit rate (as explained below) as clients are added to the system. Its ultimately low server hit rate and its inability to forward requests cause CB-R to have the highest disk I/O requirements at 10 clients and beyond. Note that beyond 10 clients, CB-R is the only algorithm for which disk requirements do not decrease as clients (and hence, more caches) are added to the system.

CB-R's server buffer hit rate drops from 58% to 24% (where 30% is what would be expected with a uniform access pattern). This drop is due to the combination of the skewed nature of the RO-HOTCOLD workload and the small client caches. Due to the small client caches (62 pages), the LRU mechanism at each client frequently ages out pages that belong to the client's hot range (at the clients, the hit rate for hot region pages is about 81%). With small numbers of clients in the system, the server buffer pool can hold all of the hot region pages for all of the active clients, and therefore, client misses due to aged-out hot pages are likely to be found at the server (e.g., with two clients, the hit rate for hot pages at the server is nearly 97%). However, as clients are added, the server can hold fewer of the active clients' hot region pages, so the server hit rate for each client's hot region pages drops; in this case, to below 18% at fifteen clients and beyond (compared to over 28% for cold

region requests). The lower hot region hit rate is due to another correlation phenomenon: the server tends to keep only the hot region pages that were most recently requested by a client. Unfortunately, these pages are the wrong pages to keep, as hot region pages tend to reside in a client's cache for a long time before they are finally aged-out and subsequently re-requested.[4] As shown in Figure 6.6, FWD has a similar overall server hit rate to CB-R. Forwarding has only a minimal effect on the server hit rate in this case — slightly lowering the hot region hit rate and raising the cold region hit rate. However, despite this similarity, FWD's resource requirements are different than CB-R's. FWD is able to satisfy a significant number of server buffer misses by forwarding requests to other clients (see Figure 6.7). Therefore, as clients are added, there is an increase in messages but a decrease in disk I/O. In contrast to FWD, the other forwarding algorithms take a more active role in affecting the server's buffering behavior. FWD-H has a server buffer hit rate that remains around 30%. The hate hints reduce the impact of the skewed workload on the server buffer hit rate, and thus, the hot region and cold region hit rates both remain close to 30%. Unfortunately, for small numbers of clients, FWD-H has a much lower hit rate than those obtained by the other algorithms. In its attempt to reduce server-client correlation, FWD-H removes hot pages from the server's buffer pool. Many of those pages, while replicated for a brief time, will be eventually aged out of the client's cache and re-referenced at the server. Thus, with small numbers of clients, the reduction in server-client correlation causes a lower server hit rate, and as a result, FWD-H has the highest disk requirements up to 5 clients (Figure 6.3). However, at 10 clients and beyond, FWD-H's server hit rate becomes better than that of CB-R and FWD because it avoids the server-client correlation that causes those algorithms to have a low hit rate for hot region pages. The reduction in server-client correlation also allows FWD-H to be more successful than FWD at forwarding requests missed at the server to other clients (Figure 6.7). As a result of the server hit rate and forwarding behavior, FWD-H sends more messages than CB-R and FWD, but beyond 5 clients, performs fewer disk I/Os.

The sending technique provides a substantial improvement in the server hit rate. As shown in Figure 6.6, FWD-S has the same initial hit rate as FWD and CB-R but it does not suffer as severe a drop in hit rate as clients are added. The high server hit rate is due to the sending technique's ability to keep hot region pages in memory. The influence of the sending technique can be seen in Figure 6.4, which shows the number of messages sent by FWD-S increasing until 10 clients are in the system. The number of dropped pages sent to the server (not shown) increases due to the reduction in server-client buffer correlation — hot pages that are aged out of client caches become less likely to be in the server's buffer pool as clients are added. Beyond 10 clients, the number of pages

[4] In fact, the 18% hit rate obtained for hot region pages is due primarily to requests for hot region pages that were recently accessed as *cold region* pages by other clients. If there were no overlapping cold region accesses, the hot region hit rate would approach zero.

sent by FWD-S begins to decrease, as it becomes more likely that a page dropped by a client is resident in the memory of another client. Despite the reduction in sent pages, FWD-S's message count remains fairly constant due to an increase in forwarded requests. However, its per transaction network bandwidth demands actually decrease (see Figure 6.5). As shown in Figure 6.7, at 10 clients and beyond, FWD-S forwards a larger percent of its server misses than the non-sending algorithms. It is important to note that the crossover point of the forwarded percentages of FWD-S and FWD-H occurs with fewer clients than the crossover of their respective in-memory percentages shown in Figure 6.2. This is because FWD-S does a better job of keeping hot range pages at the server so a miss at the server is likely to be for a cold range page. Such cold range pages are typically in the hot range of another client, and will often be cache-resident at that client.

The combination of hate hints and the sending technique gives FWD-HS the best server hit rate of the five algorithms. FWD-HS also has the highest forwarded percentage of all of the algorithms. In fact, at 20 clients and beyond, FWD-HS reads a page from disk only once; all subsequent transactions can access the page in memory. As a result, FWD-HS has the lowest disk I/O requirements and the highest message count of the five algorithms. FWD-HS also exhibits an interesting, and potentially expensive, behavior with small client populations: hot region pages are "bounced" between clients and the server. With 2 clients in the system, over 95% of the pages dropped by the clients are sent back to the server. This occurs at a large cost in messages and network bandwidth, and provides only a small savings in disk I/O. As with FWD-S, the number of dropped pages sent to the server decreases for FWD-HS as clients are added to the system. This reduction, combined with an increase in forwarded requests, results in a slight decrease in message count and a significant drop in network bandwidth requirements.

Throughput Results

The last two subsections examined the effectiveness of the algorithms in keeping pages available in memory and studied their resource requirements. With these results in mind, we now turn to the resulting performance of the algorithms. Figure 6.8 shows the throughput results for this experiment with the slow network setting (*NetworkBandwidth* = 1 MByte/sec). All of the forwarding algorithms eventually outperform CB-R, showing the potential advantages of avoiding I/O — even at the cost of additional messages. FWD-H has the highest throughput through much of the range, with FWD equaling it at 25 clients. Beyond 10 clients, the sending algorithms perform below the level of FWD and FWD-H. In this case, all of the forwarding algorithms eventually become network-bound and their relative performance becomes inversely proportional to their message

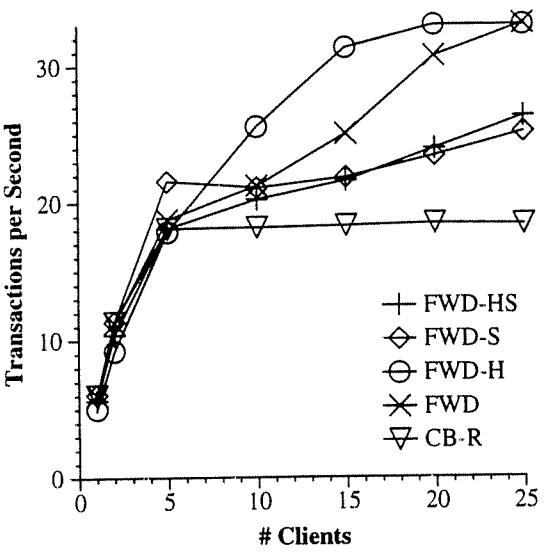


Figure 6.8: Throughput
(RO-HOTCOLD, 5% Cli Bufs, Slow Net)

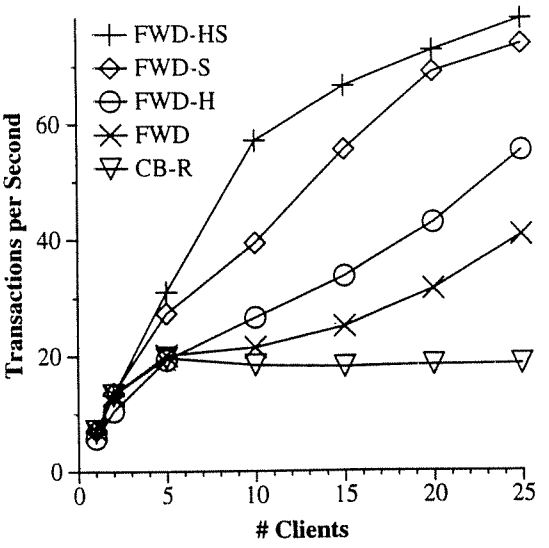


Figure 6.9: Throughput
(RO-HOTCOLD, 5% Cli Bufs, Fast Net)

bandwidth requirements. With small client populations, however, the relative performance results are somewhat different. The forwarding technique provides no clear performance improvement over the baseline CB-R algorithm; in fact, FWD-H and FWD-HS perform slightly worse than CB-R up to 5 clients.

The CB-R algorithm initially performs well because it has low message requirements and its disk requirements are in line with the other algorithms. However, the other algorithms soon produce a decrease in I/O requirements, while CB-R does not. CB-R approaches a disk bottleneck at 10 clients and ultimately has the lowest performance. FWD performs similarly to CB-R up to 5 clients, but decreasing I/O requirements allow it to eventually perform much better than CB-R, approaching a network bottleneck at 25 clients. FWD-H initially suffers due to high I/O requirements with small client populations. However, as clients are added to the system, its I/O requirements diminish and, because it has moderate message requirements, it becomes the best performing algorithm. FWD-S has the best performance at 5 clients due to its very low I/O requirements. Beyond 5 clients, however, the increase in dropped pages sent by FWD-S causes its performance to suffer relative to the FWD and FWD-H algorithms, which have lower bandwidth requirements. FWD-S is network-bound at 10 clients and beyond; its throughput improvement beyond this point is due to the reduction in network bandwidth requirements as fewer dropped pages are sent to the server. FWD-HS has the lowest I/O requirements throughout the range of client populations, but due to its high message requirements and the slow network, it performs poorly compared to FWD and FWD-H. FWD-HS is the first algorithm to hit the network bottleneck; it becomes network-bound at 5 clients. Its network bandwidth requirements cause it to perform below FWD-S prior to 15 clients, and only

slightly better than FWD-S thereafter.

Figure 6.9 shows the throughput results for the same workload and buffer pool sizes as the previous case, but with a faster (e.g., FDDI) network. The faster network has the effect of reducing the cost of using network bandwidth, and thus, the trade-off of messages for disk I/O becomes a better bargain. In this case, therefore, FWD-HS has the best performance, followed by FWD-S, FWD-H, and FWD. CB-R, the non-forwarding algorithm, has the lowest performance. The sending algorithms both become CPU-bound at the server due to the cost of processing large messages, while the other algorithms are negatively impacted by their greater I/O requirements.

Read-Only HOTCOLD, Large Client Caches

We now turn our attention to the Read-Only HOTCOLD workload with the client cache size increased to 15% of

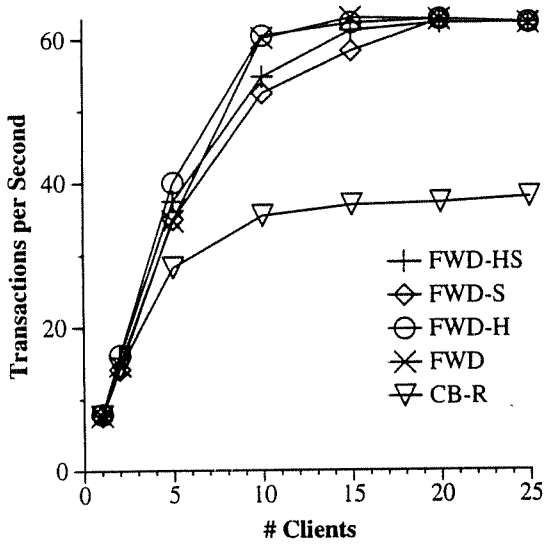


Figure 6.10: Throughput
(RO-HOTCOLD, 15% Cli Bufs, Slow Net)

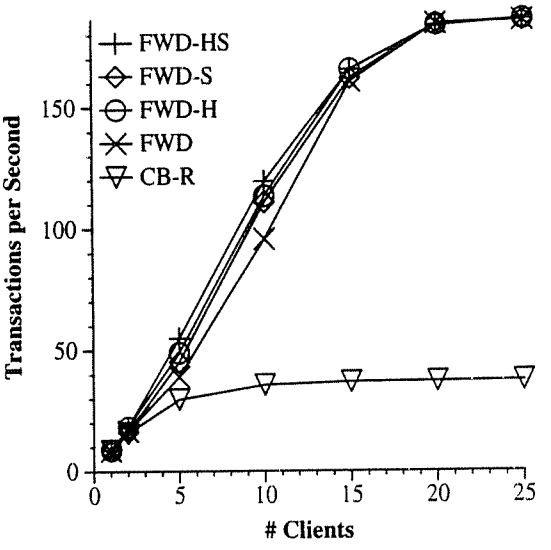


Figure 6.11: Throughput
(RO-HOTCOLD, 15% Cli Bufs, Fast Net)

the database size. The larger client caches have two important effects for this workload: 1) more of the database is available in memory with smaller numbers of clients than in the cases previously studied, and 2) the client buffers are large enough so that hot region pages are very rarely dropped from a client's cache. Figure 6.10 shows the throughput results for this case using the slow network. As can be seen in the figure, the forwarding algorithms all converge at 20 clients and beyond, while the CB-R algorithm has lower performance than the forwarding algorithms at five clients and beyond. The CB-R algorithm is once again disk-bound (although at a higher performance level than in the previous cases), and the forwarding algorithms all become network-bound at 15 clients and beyond. Prior to converging, the sending algorithms perform somewhat worse than the other

forwarding algorithms. This is because they incur large message costs for sending dropped pages back to the server, and in this case most dropped pages are cold region pages, so there is little benefit to having these pages at the server. FWD-H has a slight performance advantage up to 10 clients due to its effectiveness at reducing server-client correlation with small client populations. Unlike in the small buffer case, this reduction provides an improvement in the server buffer hit rate, as clients tend to request only cold region pages from the server. FWD-HS has a slightly better initial server hit rate than FWD-H, but its performance is penalized by its high message requirements.

The forwarding algorithms eventually converge in Figure 6.10 for three reasons: First, all of the forwarding algorithms have access to the entire database in global memory at 20 clients and beyond, so they perform no disk I/O at that point. Second, the large buffer pools cause the sending algorithms to send fewer pages back to the server as clients are added to the system — it becomes less likely that a client has the only copy of a cold region page. Third, at 20 clients and beyond, all of the forwarding algorithms have the same server hit rate (around 30%). This is because with a large number of clients, the potential overlap of each client's cache contents with the server is small, and since most page requests sent to the server are for cold region pages, the server sees a non-skewed access pattern.

The performance results for the faster network case (shown in Figure 6.11) are similar to those seen with the slower network, in that all of the forwarding algorithms converge at a performance level much higher than the CB-R algorithm (in this case, by a factor of about 5 at 25 clients). However, in this case, the fast network allows FWD-HS and FWD-H to capitalize on their high server hit rates prior to the convergence.

Summary of the Read-Only HOTCOLD Results

The study of the HOTCOLD workload in the absence of updates revealed a number of important aspects of the performance of the global memory management techniques. Most importantly, it was shown that forwarding page requests to remote clients can provide significant performance improvements. Forwarding a request saves a disk I/O while requiring no extra CPU work at the server, thus offloading an important shared resource. The cost of forwarding is an extra control message plus the latency due to message handling at the remote client. Even in network constrained situations, forwarding was found to be beneficial.

The hate hints technique was found to improve the performance of forwarding by reducing the correlation of buffer contents between clients and the server. An important exception to this was in cases with small client caches and small numbers of clients. In such cases, the hate hints were found to hurt the server hit rate by removing valuable hot region pages from the server buffer pool. As a result, there were cases in which the

FWD-H algorithm had a larger portion of the database available in memory, but had higher disk requirements than other algorithms. In contrast, the sending technique was found to be effective at keeping hot region pages in memory. However, the sending technique was found to pay a large price in message bandwidth to avoid disk I/Os, often causing the slow network to become a bottleneck while the disk became underutilized. Furthermore, when the size of the client buffer caches was large enough to keep hot region pages from being replaced at the clients, the sending technique was detrimental to performance because it used valuable network bandwidth to keep cold region pages in memory.

The FWD-HS algorithm, which uses a combination of the hate hints and sending techniques, was able to keep more of the database available in memory than any of the other algorithms studied due to its ability to make effective use of the server buffer pool. However, this did not always translate into better performance. With small client caches, the combination of the techniques resulted in hot region pages being bounced between clients and the server across the network, resulting in heavy network traffic. With larger caches, its performance was negatively affected by the tendency of the sending technique to waste network bandwidth on cold region pages. Message costs hurt the throughput of FWD-HS in the slow network cases, but when the fast network was used, FWD-HS was the best performing algorithm.

In general, the experiments showed that the global memory management techniques were effective in offloading the server's disks by increasing the amount of the database available in memory. However, it was also seen that while offloading the disk in this manner can provide substantial performance gains, doing so is not a guarantee of improved performance. In particular, if the wrong pages are kept in memory, or if the price paid to keep pages available in memory is too high, performance can suffer.

6.3.2 Experiment 2: Read-Write HOTCOLD Workload

The previous section analyzed the five memory management algorithms in the absence of writes in order to examine their behavior without the complications introduced by dirty pages. In this section, we investigate the performance of the algorithms using the HOTCOLD workload with a write probability of 20% (*HotWriteProb* = *ColdWriteProb* = 0.20). The following discussion concentrates on those aspects of performance that are caused by the introduction of writes.

Read-Write HOTCOLD, Small Client Buffer Pools

Figure 6.12 shows the percentage of the database available in memory for the Read-Write HOTCOLD workload with small client caches and the slow network. Compared to the read-only case (Figure 6.2) CB-R remains

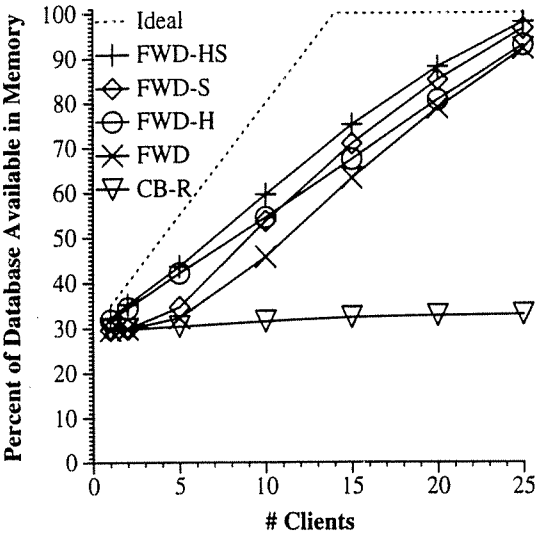


Figure 6.12: % of DB Available in Memory
(RW-HOTCOLD, 5% Cli Bufs, Slow Net)

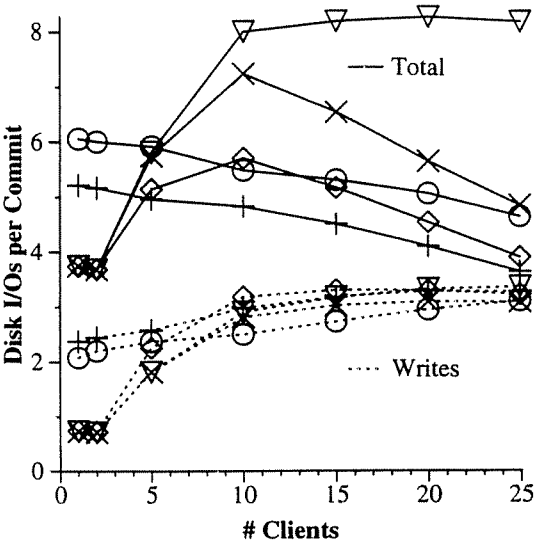


Figure 6.13: Disk Writes and Total I/O
(RW-HOTCOLD, 5% Cli Bufs, Slow Net)

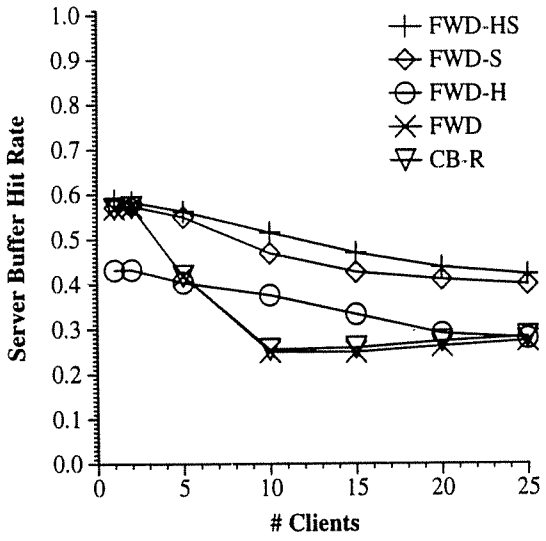


Figure 6.14: Server Buffer Hit Rate
(RW-HOTCOLD, 5% Cli Bufs, Slow Net)

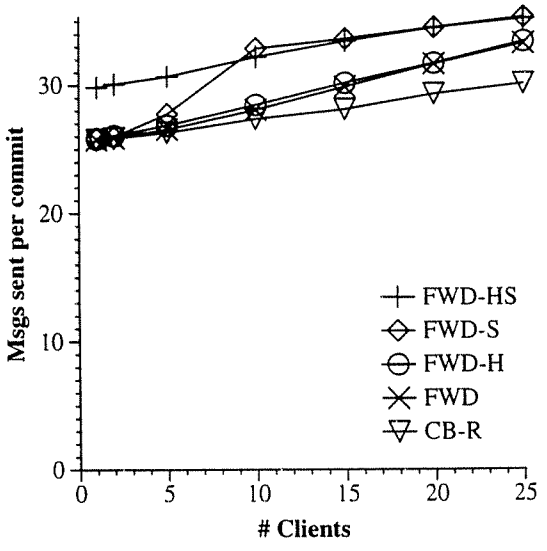


Figure 6.15: Messages Sent/Commit
(RW-HOTCOLD, 5% Cli Bufs, Slow Net)

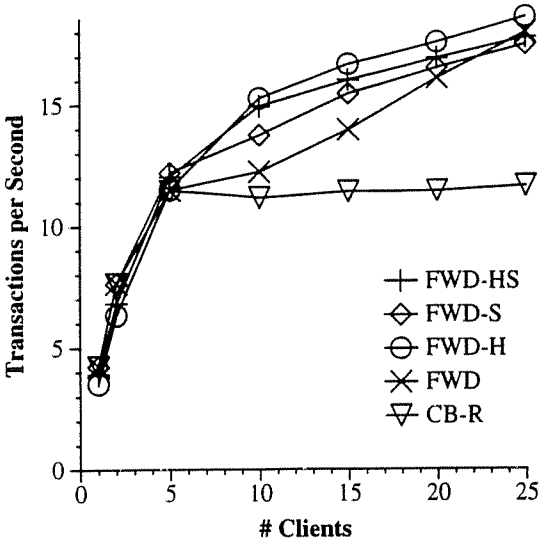


Figure 6.16: Throughput
(HOTCOLD, 5% Cli Bufs, Slow Net)

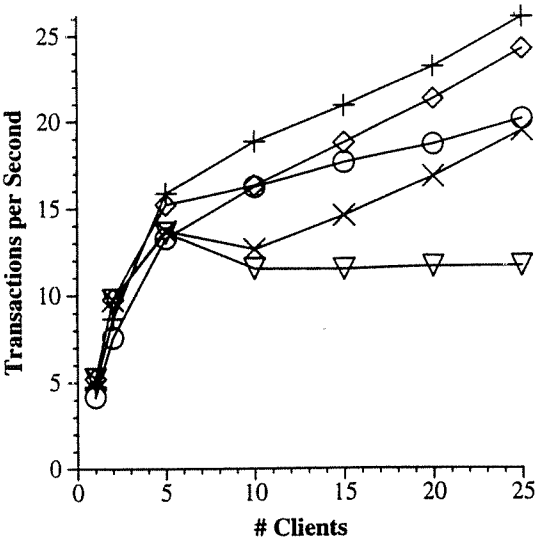


Figure 6.17: Throughput
(HOTCOLD, 5% Cli Bufs, Fast Net)

largely unchanged, FWD and FWD-H initially have a slight degradation but eventually have an improvement at larger client populations, and FWD-HS suffers degradation throughout the range of clients. Most strikingly, the large benefit of FWD-HS observed in the earlier experiment is not present here. With small client populations, the server-client buffer correlation is increased by the dirty pages that are sent to the server by committing transactions. This particularly hurts the FWD-H and FWD-HS algorithms, which gained by reducing this correlation in the read-only case. However, as more clients are added, callback requests begin to reduce the replication among client buffer contents; before a page is updated at one client, it is removed from the buffers at any other clients that have it cached. Therefore, the result of the addition of writes is to increase server-client correlation while slightly decreasing client-client replication.

Figure 6.13 shows the total number of disk I/Os (shown as solid lines) and the number of disk writes (shown as dotted lines) performed per committed transaction. The most noticeable change in overall disk requirements is that the FWD-HS and FWD-S algorithms no longer enjoy the large advantage that they had in the read-only case (shown in Figure 6.3). The disk requirements of the algorithms are affected by the need to perform disk writes for dirty pages that are aged out of the server's buffer pool and by the changes in the client and server buffer contents caused by the handling of dirty pages. Disk writes are an additional cost that is incurred by all of the algorithms, however, with 1 to 5 clients the FWD-H and FWD-HS algorithms write more pages than the other algorithms. This is because the hate hints cause requested dirty pages to be moved to the head of the LRU chain, reducing their residence time in the server's buffer pool and thereby reducing the opportunity for

combining multiple client writes into a single disk write. As more clients are added, the other algorithms also incur an increase in disk writes due to additional traffic through the server buffer pool. This traffic is caused by disk reads for the FWD and CB-R algorithms, and by dropped pages that are sent to the server for FWD-S.

The disk read requirements of the algorithms are changed by the introduction of writes in two ways. First, dirty pages sent to the server by committing transactions impact the server hit rate (shown in Figure 6.14) in an algorithm-dependent manner. For FWD-H, these pages improve the server hit rate significantly for small client populations (e.g., with 2 clients the server hit rate is 43%, as compared to 30% in the read-only case). The pages dirtied by a client are likely to be hot region pages for that client, so sending those pages to the server helps overcome FWD-H's tendency to remove hot pages from the server's buffer pool. The beneficial effect of the dirty pages is reduced as clients are added to the system. In contrast, FWD-S and FWD-HS suffer a reduction in server hit rate due to dirty pages. A page that is sent to the server because it is dirtied is not necessarily the only copy of the page in the system. Such dirty pages therefore reduce the effective use of the server buffer pool that was exhibited by the sending algorithms in the read-only case. The second way that writes affect the disk read requirements is via callbacks. Callbacks reduce client-client replication, which increases the effectiveness of forwarding, thereby reducing disk read requirements. Therefore the disk read requirements for the FWD algorithm at 15 clients and beyond are somewhat lower than in the read-only case.

Figure 6.15 shows the number of messages sent per committed transaction. The differences among the algorithms are much smaller here than those seen in read-only case (shown in Figure 6.4). This occurs for two reasons. First, writes increase the message requirements of all of the algorithms by introducing four new kinds of messages: 1) write lock requests, 2) callback requests for cached read locks, 3) subsequent re-requests for pages and locks that were called-back and 4) messages for dirty pages sent to the server prior to commit. Secondly, the relative message requirements of the FWD-HS algorithm are reduced for small client populations because the dirty pages sent to the server (which are sent by all algorithms) reduce the number of dropped pages that FWD-HS must send to the server.

Figure 6.16 shows the throughput for the RW-HOTCOLD workload with small client caches and the slow network. The forwarding algorithms all outperform CB-R at 10 clients and beyond. This is due once again to CB-R's high disk requirements. All of the forwarding algorithms have similar throughput at 25 clients. FWD-H attains the highest throughput at 10 clients and beyond, but has poor throughput with small client populations due to high disk read and write requirements. The sending algorithms, which reach a network bottleneck at 20 clients, perform close to FWD-H due to the reduction in differences in message requirements. FWD-H and FWD both approach, but do not quite reach, a network bottleneck at 25 clients. FWD-HS initially performs poorly due

to message costs and disk writes caused by the sending of dropped pages that force dirty "hated" pages out of the server's buffer pool. When the faster network is used, the throughput results (shown in Figure 6.17) display similar trends but smaller differences compared to what was observed in the read-only case (Figure 6.9). One difference is that here, all of the algorithms become disk-bound so FWD-H and FWD-HS are impacted by their increased disk requirements (compared to Figure 6.9).

Read-Write HOTCOLD, Large Client Buffer Pools

Figure 6.18 shows the throughput results for the Read-Write HOTCOLD workload run with larger client caches and the slow network. Once again, CB-R performs at a much lower level than the forwarding algorithms due to its high disk requirements. The forwarding algorithms all have similar throughput; their performance is primarily dictated by their message behavior (not shown), which becomes nearly identical at 15 clients and beyond. This convergence is similar to what was observed in the read-only case (Figure 6.10) and occurs for the same reasons. The sending algorithms perform closer to the others in this case because the sending of dirty pages and the effect of callbacks result in fewer dropped pages being sent to the server. When the fast network is used in conjunction with the larger client caches (Figure 6.19), the relative performance of the algorithms is driven by disk demands, and an interesting effect occurs: at 10 clients and beyond, the forwarding algorithms separate into two distinct classes — the algorithms that use hate hints, and those that do not. FWD and FWD-S outperform the other forwarding algorithms because hate hints lead to more disk writes by reducing the amount of time that dirty pages are retained in the server buffer pool. The sending technique has no differentiating effect in this case because few pages are sent back to the server beyond 10 clients.

Summary of the Read-Write HOTCOLD Results

The introduction of writes to the HOTCOLD workload was found to have a number of complex effects on the message and disk requirements of the global algorithms; however, the relative performance of the algorithms was not greatly affected in most of the cases studied here. When the slow network was used, the most important impact of the writes was a reduction in the differences among the message requirements of the algorithms. This reduction was the result of additional messages incurred by all algorithms and a decrease in the number of dropped pages sent to the server by the sending algorithms. When the fast network was used, the effects of updates on disk requirements played a greater role in determining the relative performance of the algorithms. These effects varied greatly among the algorithms. All algorithms had increased disk requirements due to disk writes. The sending algorithms incurred an additional increase in disk requirements due to a reduction in server buffer hit

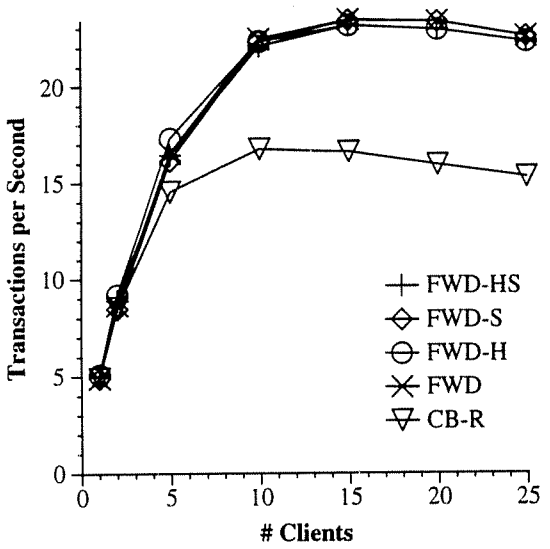


Figure 6.18: Throughput
(HOTCOLD, 15% Cli Bufs, Slow Net)

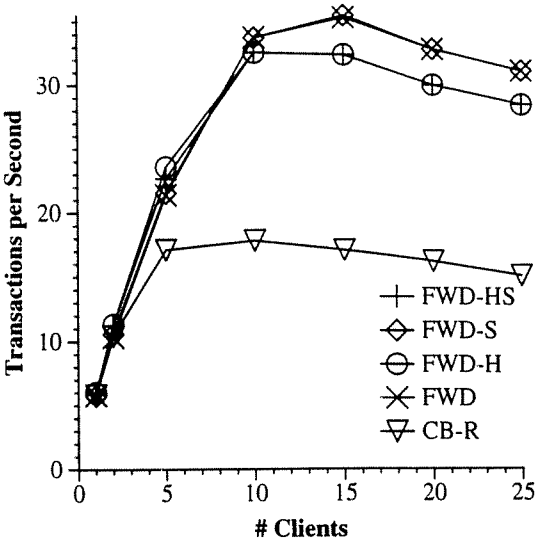


Figure 6.19: Throughput
(HOTCOLD, 15% Cli Bufs, Fast Net)

rates caused by dirty pages sent to the server by committing transactions, which increased the server-client buffer correlation. FWD-H and FWD-HS paid a high price in disk writes when few clients were present because the hate hints reduced the residency time of dirty hot region pages in the server buffer pool. The FWD-S, FWD, and CB-R algorithms all saw a slight decrease in disk reads because the dirty pages sent to the server increased their server buffer hit rates. (This was due to dirty pages being re-referenced by clients from which they were previously called-back). Also, the utility of forwarding was slightly improved by the callback mechanism, which reduces client-client replication.

As the preceding discussion indicated, the impact of sending dirty pages to the server on commit varies depending on the memory management algorithms, workload characteristics, and system resources used. In general, however, the sending of dirty pages to the server at commit had a negative impact on performance — requiring additional messages, increasing the cost of hate hints, and interfering with the strategies used for managing the server's buffer pool. The sending of dirty pages to the server at commit time helps to simplify the recovery system in a client-server DBMS [Fran92c], but it could be avoided or reduced at the expense of increasing recovery complexity. As will be discussed in Section 6.4.2, a similar issue (forcing dirty pages to disk) has been investigated for shared-disk environments [Moha91, Dan92]. Section 7.4.1 further investigates policies for handling dirty pages in the client-server environment.

6.3.3 Experiment 3: PRIVATE Workload

The third experiment that is reported here uses the PRIVATE workload. In this workload (see Table 5.2), each client has a 25-page hot region of the database to which 50% of its accesses are directed; the other 50% of its accesses are directed to a 625-page read-only area of the database. Clients do not access pages in each other's hot regions, so there is no read/write sharing of data in this workload. Thus, the PRIVATE workload has updates and high locality but has no data contention. Figure 6.20 shows the throughput results for the PRIVATE workload

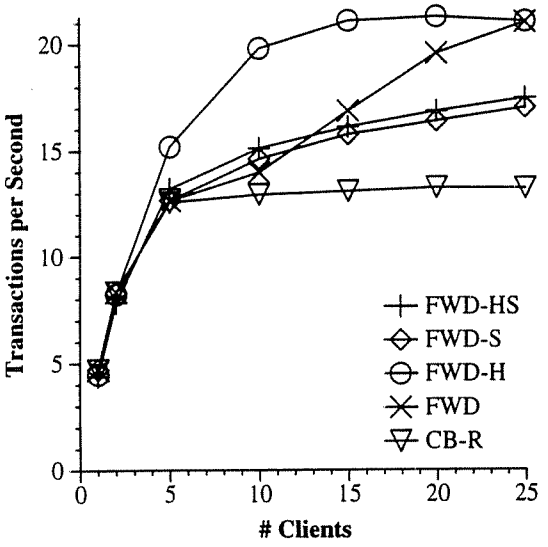


Figure 6.20: Throughput
(PRIVATE, 5% Cli Bufs, Slow Net)

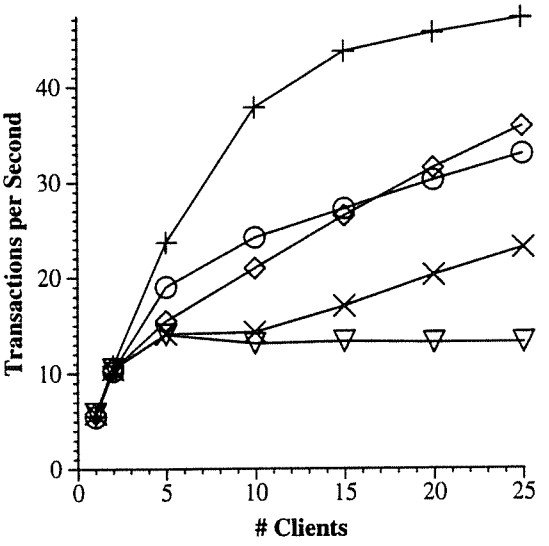


Figure 6.21: Throughput
(PRIVATE, 5% Cli Bufs, Fast Net)

with small client caches and the slow network. In this case, the relative performance of the algorithms is similar to what was seen for the RW-HOTCOLD workload (Figure 6.16). There is however, an important difference — the relative performance of the sending algorithms is worse here. This is due to higher message requirements resulting from sending more dropped pages to the server. Each client has a private hot region, and thus, when a client drops a hot region page, it will be sent back to the server unless the server already has a cached copy of the page; there is no chance that a copy of such a page will exist at another client. In the fast network case (Figure 6.21), the FWD-HS algorithm performs much better than the others because it is effective at keeping more of the database available in memory and thus has lower disk requirements.

Figure 6.22 shows the throughput results when the client cache size is increased to 15%. In this case, the client caches are large enough to contain the entire hot region for each client. As a result, any hot region pages that are kept in the server's buffer pool are detrimental to performance because they will not be accessed by any clients and thus, they waste space that could be used for cold region pages. In this situation, the hate hints will

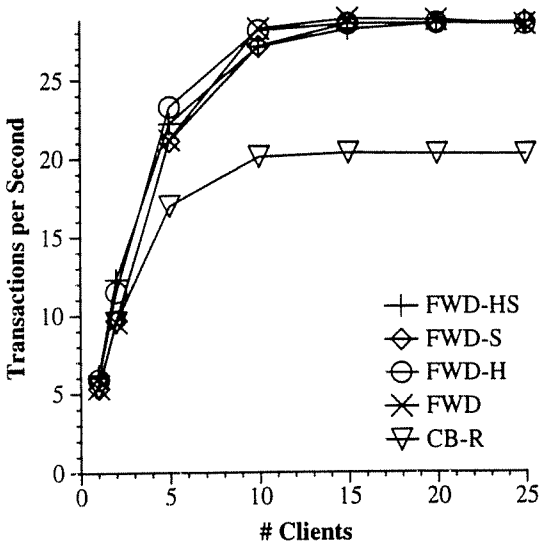


Figure 6.22: Throughput
(PRIVATE, 15% Cli Bufs, Slow Net)

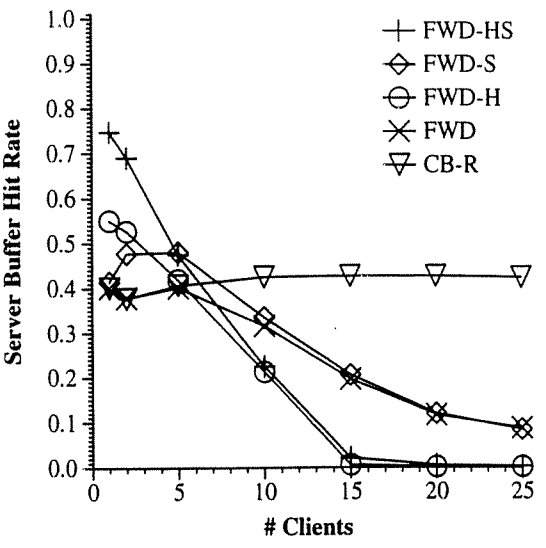


Figure 6.23: Throughput
(PRIVATE, 15% Cli Bufs, Slow Net)

tend to remove cold region pages from the server's buffer pool while retaining dirty hot region pages, resulting in the poor server hit rates shown in Figure 6.23. The other forwarding algorithms (FWD and FWD-S) also have poor server hit rates due to the presence of dirty hot region pages in the server's buffer pool. The sending technique gives FWD-S a slightly better server hit rate than FWD with small numbers of clients, as it causes some cold region pages to be sent back to the server. The CB-R algorithm has the best server hit rate at 10 clients and beyond due to the presence of cold region pages that are read in from disk. The other algorithms do not read as many cold region pages from disk since they are able to satisfy many server misses by forwarding requests.

While the interaction of the workload with the handling of dirty pages results in the interesting buffering effects just described, the net effects on performance are not as great as might be expected. In the slow network case (Figure 6.22), there are only small differences among the performance of the algorithms. In this case the performance results for the forwarding algorithms are primarily dependent on the message behavior of those algorithms, and they all eventually have similar message behavior. The CB-R algorithm becomes disk-bound despite its superior server hit rate, while the forwarding algorithms perform virtually no disk reads at 15 clients and beyond. In the fast network case (not shown), the forwarding algorithms become server CPU bound at 15 clients and converge shortly thereafter, at about 3.5 times the throughput of the disk-bound CB-R algorithm.

6.3.4 Experiment 4: UNIFORM Workload

As described in Table 5.2, the UNIFORM workload has no locality and has a write probability of 20%. With small client caches and the slow network (shown in Figure 6.24), FWD-H performs the best until it is matched by

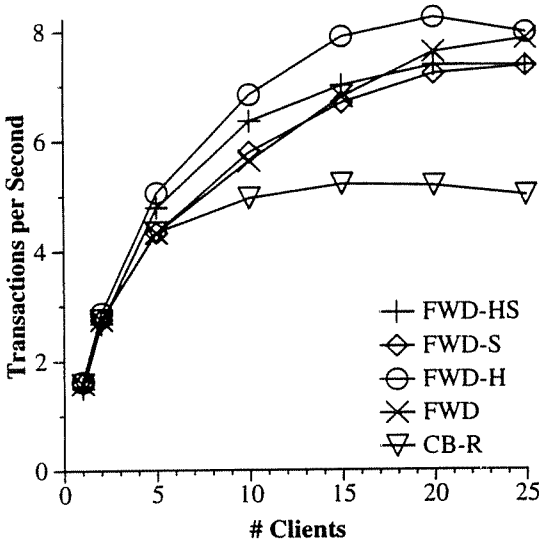


Figure 6.24: Throughput
(UNIFORM, 5% Cli Bufs, Slow Net)

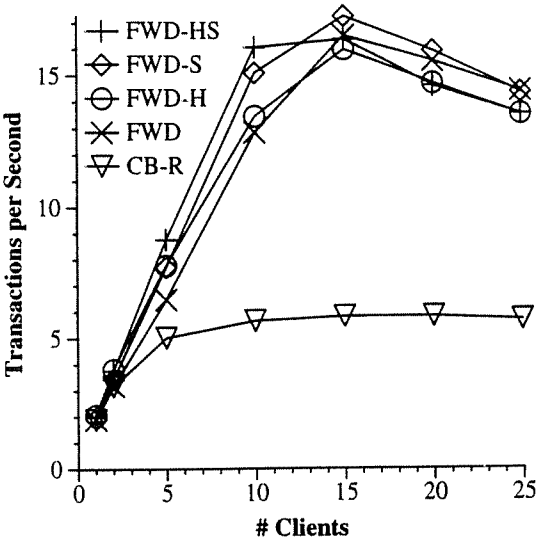


Figure 6.25: Throughput
(UNIFORM, 15% Cli Bufs, Fast Net)

the FWD algorithm at 25 clients. Initially, FWD-H's advantage is due to a high server buffer hit rate. In this case, the hate hints improve the server buffer hit rate for small client populations by reducing server-client correlation. In contrast to what was observed in the skewed workloads, decreasing server-client correlation actually improves the server buffer hit rate here, since all pages are equally likely to be accessed. As more clients are added, FWD-H's advantage is the result of it having lower message requirements than the sending algorithms. The lack of locality reduces the importance of the sending technique, so FWD-S keeps a smaller portion of the database in memory than FWD-H until 25 clients (at which point they are equal). FWD keeps the smallest portion of the database in memory among the forwarding algorithms in this case, but its lower message requirements allow it to perform relatively well with larger numbers of clients.

With larger client caches and the fast network (shown in Figure 6.25), the forwarding algorithms begin to approach the bifurcated state that they reached in the RW-HOTCOLD case (Figure 6.19). However, there are two noticeable differences in the trends compared to the RW-HOTCOLD case. First, prior to 20 clients, the FWD-HS algorithm performs better than FWD-H, and second, prior to 25 clients, the FWD-S algorithm performs better than FWD. At 25 clients, all of the forwarding algorithms have access to the entire database in memory. However, prior to this point, FWD-HS keeps a larger portion of the database available in memory than FWD-H,

and FWD-S keeps a larger portion of the database available in memory than FWD. In this case, the ability to keep more of the database available in memory results in noticeably better performance. Once all of the forwarding algorithms have the entire database available in memory, FWD-H and FWD-HS pay a slight penalty due to additional disk writes caused by hate hints.

6.4 Related Work

This section briefly discusses work related to global memory management in client-server DBMSs, data sharing DBMSs, and other distributed systems.

6.4.1 Workstation-Server Database Systems

Several recent papers have investigated issues of global memory management for DBMSs in a workstation-server environment. In [Leff91], the problem of replica management for efficient use of the global memory resources of a distributed system was addressed. In particular, an analytical model was used to investigate the tradeoffs between a "greedy" algorithm, where each site makes its own caching decisions to maximize its own performance, and two algorithms where caching decisions are made (statically) in order to maximize overall system performance. The latter two algorithms were intended to provide an upper bound on the potential performance improvement that could be expected due to global memory management, and not as practical algorithms for implementation. The key tradeoff found was that without coordination, all sites chose to cache the hottest objects, while the coordinated strategies were able to keep many different pages in memory, thereby replacing disk I/Os with (cheaper) messages. The study did not consider updates and all sites had the same database reference patterns. Our algorithms also attempt to keep more pages available in memory, but they focus on using the server buffer as a mechanism for achieving this rather than modifying client buffer replacement policies. Algorithms for using the memory of underutilized workstations (called *mutual-servers*) to keep more of the database in memory were proposed and studied in [Pu91a]. The algorithms included variations in which the sender and/or the receiver played active roles in initiating the caching of a page at a mutual-server. All of the algorithms were broadcast-based, and the study did not consider concurrency control or data contention issues.

6.4.2 Transaction Processing Systems

Issues related to some of the global memory management techniques discussed here are also addressed in a data-sharing context. In one paper [Dan91] an analytical model was used to study a two-level buffer hierarchy.

The paper investigated policies for placing pages in a shared buffer based on when pages are updated, read in from disk, and/or aged-out of a private buffer. The study identified a number of buffer correlation effects similar to those discussed in Section 4, especially those dealing with the relative sizes of the shared (server) and private (client) buffers. There are several important differences between the two studies, however. First, [Dan91] did not investigate global algorithms that allowed shared-buffer misses to be serviced by private buffers. Second, it compared the techniques based on their buffer hit rates but did not investigate the impact of the algorithms on actual transaction throughput. Many of the experiments discussed in Section 4 showed that the buffer hit rates often did not determine the performance of the algorithms. Third, due to the nature of the shared-disk environment, the [Dan91] study investigated workloads in which there was no difference among the access patterns at the various private buffers. Therefore, the complex interaction of server-client correlation and server hit rates that arose here in the HOTCOLD workloads were not identified.

A more recent paper [Dan92] studies callback-style shared-disk caching algorithms and investigates the performance gains that are available by avoiding disk writes for dirty pages when transferring a page between sites. Adding this optimization to a shared-disk system results in more complex recovery schemes, as described in [Moha91]. An algorithm which avoids replicating copies at multiple sites was studied and was found to have tradeoffs similar to some of those seen for our forwarding techniques. However, there are significant differences between the client-server environment and the data-sharing environment (e.g, the use of the server for logging and recovery [Fran92c], and the expense of messages in a client-server system). Thus, many of the tradeoffs that we have identified in a client-server context differ from those observed in data-sharing systems.

Another recent paper [Rahm92] studies the use of several types of extended memory to improve the performance of transaction processing systems. Extended memory adds a new memory hierarchy level between main memory and disk. An added dimension is the use of non-volatile memory to avoid data and log disk writes. The paper also investigates other types of extended memory such as solid-state disks and disk caches. Many of the correlation issues found in our study (and in [Dan91]) also arise in this environment. In particular, the correlation caused by forcing dirty pages to the extended memory (similar to our copying of dirty pages back to the server), was shown to reduce the effectiveness of the extended memory. This study differs from ours because of fundamental differences in the system architecture; these include the use a single main memory buffer (versus multiple clients), lower communication costs between the levels of the hierarchy, and differences in the types of workloads studied.

6.4.3 Non-DBMS Approaches

Issues related to global memory management have also been addressed in distributed object systems such as Emerald [Jul88], where methods for allowing objects to migrate among sites were addressed. Migration in this case was intended to improve performance by bringing objects to the sites where they were being accessed, and also to simplify the programming of distributed applications, rather than to avoid disk I/O. The idea of using the memory of idle workstations as a backing store for virtual memory was investigated in [Felt91]. This paper raised several policy issues and presented a simple queueing model study that indicated that the approach has potential for significant performance gains over swapping to disk, even with current networking and OS technology. Finally, work in Non-Uniform Memory Access architectures, in which the memories of nodes in a multiprocessor system are viewed as a single memory hierarchy, is relevant as well. This work includes [LaRo90], which studied the performance of a wide range of proposed memory management policies and [LaRo91], which investigated dynamic policies that can be adapted to particular page reference behaviors.

6.5 Conclusions

This chapter has examined performance tradeoffs for global memory management in page server database systems. Three different memory management techniques were presented. Each of the proposed techniques can be implemented within the context of avoidance-based cache consistency maintenance algorithms, and they require no information to be kept at the server or at the clients beyond what is already required by such algorithms. The primary technique, forwarding, attempts to avoid disk I/O at the server by forwarding page requests to remote clients that have a requested page in their cache. Forwarding allows the caches of remote clients to be treated as an additional level in a global memory hierarchy. Accesses to this additional memory level incur a cost of one control message and some additional client processing beyond the cost of a request made to the server, so it lies between the server memory and the disks in the hierarchy.

The two other techniques are intended to increase the utility of forwarding by more efficiently managing the global memory of the system. These techniques attempt to exploit the server's buffer pool in order to keep a larger portion of the database available in the global memory. One technique, called hate hints, is a simple heuristic which tries to reduce replication between the buffer pool contents of the server and its clients. In this technique, the server's buffer replacement policy is modified to "hate" a page when it sends a copy of that page to a client. The other technique, called sending dropped pages, attempts to retain pages in memory by keeping a client from simply dropping a valuable page from its cache. By piggybacking information on other messages, a

client informs the server of its intention to drop a page. If the server then determines that the page to be dropped is the only copy of that page in global memory, it asks the client to send the page back rather than dropping it.

These three techniques were compared under a range of workloads and system configurations using via simulations. The results of the performance study show that, as expected, forwarding can provide significant performance gains over a non-forwarding cache management algorithm. The study also showed that the hate hints and sending techniques were indeed effective in keeping a larger portion of the database in memory. However, while these techniques achieved their objectives, they did not always yield performance improvements and in some cases were even detrimental to performance. For example, the hate hints technique was successful at reducing replication, but in some situations it removed valuable pages from the server buffer pool — thereby increasing I/O demands. The sending technique was found to be expensive in network-constrained situations in which using messages to avoid disk I/Os is the wrong approach to take. However, in many situations the sending and hate hints techniques were both shown to provide substantial performance gains. The study also investigated the impact of updates on the buffering behavior and performance of the algorithms, and it identified issues in the interaction of global memory management and the management of dirty pages for supporting transaction durability.

A number of areas for future work were raised by this study. First, the experiments identified situations where the global techniques caused the system to become unbalanced or to perform extra work. For example, there were cases in which the disk became underutilized while the network became saturated due to the effectiveness of the forwarding technique. Another example was the interaction of hate hints and dirty pages in the server buffer, which caused an increase in the number of disk writes performed per transaction. These situations demonstrate the need for algorithms that can adapt to the resource and memory usage patterns of the system. Several extensions along these lines could be easily added to the existing techniques. Another important area for future study is the recovery and performance implications of techniques that would avoid having clients send dirty pages to the server prior to committing a transaction. A final avenue for future work is to investigate global memory management algorithms that take a more active role in determining where pages should reside in the system. Such algorithms could control issues such as the amount of replication allowed at various levels in the system and the placement of page copies in the memories of idle or under-utilized workstations.

Chapter 7

Local Disk Caching

7.1 Introduction

This chapter explores the use of client disks as an additional resource that can be used to improve system performance and scalability. As was stated in Section 2.1, most current OODBMSs exploit client *processor* resources through the use of a *data-shipping* architecture, which enables much of the work of data manipulation to be performed at clients. Furthermore, as described in the preceding chapters, systems can exploit client *memory* resources through the use of intra- and inter-transaction caching and global memory management techniques. In contrast, existing systems provide only limited support for exploiting client *disk* resources. This omission is potentially costly, as client disks represent a valuable addition to the storage hierarchy of a client-server OODBMS due to their capacity and non-volatility. This chapter addresses this issue by investigating the performance gains that can be realized by adding client disks to the storage hierarchy of page server DBMSs.

7.1.1 Alternatives for Integrating Client Disks

There are several ways in which existing OODBMS typically allow client disks to be used. The first, and most common, is to use local disks to make each client a *server* for part of the database. The data to be placed on client disks is determined statically by partitioning the database among the clients. Clients are given ownership of the data pages for which they act as server, meaning that they are responsible for maintaining the consistency of the data, and must always be capable of providing its most recent committed value. Another way that existing systems allow the use of client disks is indirectly, through virtual memory swapping. Most OODBMS keep their client caches in virtual memory. If the cache is larger than the allocated physical memory, the operating

system will swap parts of it to disk. The Versant system provides an additional way of using client disks, called the "Personal Database" [Vers91]. Users can check objects out from the shared database and place them in a personal database, which can reside on the client disk. Objects that are checked out cannot be accessed by other clients.

The first approach, making clients act as servers, has implications for data availability, as the crash of a client causes the data owned by that client to become unavailable. As was discussed in Section 2.3.2, allowing clients to own data is problematic due to inherent asymmetries in workstation-server environments, namely, that servers are expected to be more reliable and available than clients. Availability concerns limit the applicability of client ownership policies to data that is private and/or of limited value, or to systems in which substantial expense is incurred to make clients more reliable. The problems of the second approach, using operating system virtual memory for buffer management, are well known (e.g., [Ston81, Trai82]), and stem from (among other things) the operating system's lack of knowledge about database access patterns and differences in disk management policies. Relying on the operating system to manage the client disk places an important performance issue beyond the control of the database system. Furthermore, virtual memory swapping makes only transient use of client disks, and thus, the disks can cache data pages only while a caching process is active at a client.

In this chapter, a different approach is taken, namely, the idea is to extend the memory caching work that was described in the previous chapters to take advantage of client disks. As described in Section 2.3.3, caching enables the use of client resources (in this case, disks) without incurring the problems associated with giving ownership of data to clients. This approach to using client disks is referred to as the *extended cache* architecture. The use of client disks as an extended cache provides a *qualitative* change in the utility of caching at client workstations compared to memory-based caching strategies. The lower cost per byte of disk storage increases the amount of data that can be cached at a client, possibly enabling the caching of the entire portion of the database that is of interest at that site. This has the potential to affect the basic trade-offs in cache management (as compared to memory-only caching), and may change the role of the server from that of a data provider to that of an arbiter of data conflicts and a guarantor of transaction semantics.

In terms of related work, we are aware of only one other study of client disk caching for DBMSs [Deli92]. This work investigates a system in which relational query results are cached on client disks but all updates are performed at the server. Prior to executing a query, a client sends a message to the server requesting any updates that have been applied to tuples cached at the client. In response, the clients are sent logs containing relevant updates, which are then applied to the cached query results. As will be seen in Section 7.2, the extended cache architecture studied here is quite different; it allows updates to be performed at clients, and uses the disk as a

page cache that is largely an extension of the LRU-managed memory cache. Also related is work on distributed file systems. Unlike existing DBMSs, some distributed file systems do use client disks for caching (e.g., Andrew [Howa88] and its follow-on project CODA [Kist91]). As discussed in Section 2.4, however, distributed file systems, differ from client-server DBMS in significant ways, including: 1) they support caching at a coarse (e.g., file) granularity, rather than at a page or object granularity, 2) they do not support serializable transactions on the cached files, and 3) they are typically designed under the assumption that sharing is an infrequent occurrence.

7.1.2 Extended Cache Design Issues

This chapter addresses three specific issues in the design of an extended cache. First, algorithms for accessing data cached on client disks and ensuring its consistency are developed and analyzed. As will be shown in the study, the success of disk-caching in offloading the server for certain workloads can result in the server disk *writes* becoming the next bottleneck. Therefore, the second issue studied is techniques for reducing the demands on the server disks that result from transaction updates. Thirdly, several techniques for managing the transition of client disk caches from an off-line state to an on-line state are discussed. These techniques build on the methods that are developed in the first part of the chapter for ensuring the consistency of disk-cached data.

7.2 Extended Cache Implementation

In this section algorithms for managing and utilizing client disks in the extended cache architecture are proposed. Before describing these algorithms, however, we first elaborate on our model of how the client disk is employed by the database system.

7.2.1 Disk Cache Management Pragmatics

This work assumes that each client has a fixed amount of disk space that is allocated for use as a cache for database pages; this area is managed by the database system. This space is referred to as the *disk cache*, and likewise, the area of main memory that is used to cache database pages as the is referred to as the *memory cache*. It is also assumed that each page is tagged with a version number that uniquely identifies the state of the page with respect to the updates applied to it (as in the C2PL algorithm described in Section 4.3).

The memory cache is managed through the use of a data structure containing an entry for each resident page and a list of available memory cache slots. The LRU chain is threaded through this structure. The disk cache is managed as a FIFO queue based on when pages are placed in the disk cache (rather than LRU). To implement the

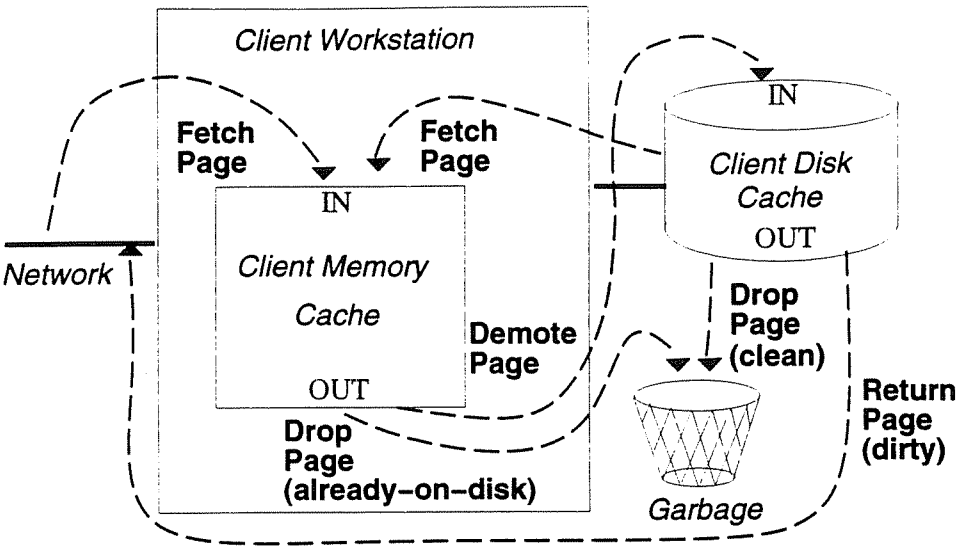


Figure 7.1: Client Page Flow

disk cache efficiently, a structure analogous to that used to manage the memory cache must also be maintained in memory. This structure contains an entry for every page resident in the disk cache and a list of available disk cache slots. At a minimum, the entry for each page contains the position of the page in the disk cache page queue, its location on disk, and its version number.

Pages flow between the memory and disk caches of a client as shown in Figure 7.1. A new page is first brought into the memory cache as the result of a cache miss. Pages can be faulted in from the server or from the local disk.[1] To bring a new page into the memory cache, a cache slot must be made available for the page (unless a free slot already exists). To open a memory cache slot, the least recently used page in the memory cache is chosen for replacement. When a page is replaced from the memory cache it is *demoted* to the local disk cache.

There are three cases to consider when demoting a page from the memory cache to the disk cache. The first two cases arise if a copy of the demoted page is already resident in the disk cache. If the copy in the memory cache has not been updated, then its disk-resident copy is simply made the most recently added page in the disk cache by adjusting the disk cache control information (which is memory-resident). Thus, for the first case, no disk access is required. The second case arises when an out-of-date copy of the demoted page is present in the disk cache. In this case, the disk cache copy of the page is overwritten and it becomes the most recently added page in the disk cache. The third case arises when no copy of the demoted page exists in the disk cache. In this case a slot in the disk cache must be made available for the demoted page. The process of opening a disk slot is

[1] Pages copied from the local disk also remain in the disk cache at their current position in the disk cache page queue.

analogous to the replacement process in the memory cache. If no slot is available, then the least recently added page is removed from the disk cache. The fate of the page chosen for replacement depends on the status of its contents. If the page contains updates that are not reflected in any other copies of the page (e.g., at the server), then a copy of the page is sent to the server. Otherwise, the chosen page is simply overwritten.

7.2.2 Extended Cache Management Algorithms

As with the global memory work of the previous chapter, the algorithms developed here are built on top of the Callback-Read (CB-R) algorithm for memory cache management. As stated earlier, due to its combination of avoidance-based and pessimistic cache consistency maintenance, CB-R provides good performance, while being relatively simple to implement and extend. This section describes how CB-R is modified to support the use of client disks in the extended cache architecture.

Two important aspects of memory management algorithms for any distributed system that supports replication or caching are *hierarchy search order* and *consistency maintenance*. The hierarchy search order dictates the process through which a particular data item is found in the storage hierarchy. Consistency maintenance ensures that the caching of data in the storage hierarchy does not cause a violation of transaction execution serializability. These two dimensions define a design space for cache management algorithms incorporating client disks. In the following, each of these dimensions is first addressed separately. Algorithms that integrate the dimensions are then described.

Hierarchy Search Order

The goal of the hierarchy search order is to allow transactions to obtain the *lowest cost* copy of a page among the copies present in the system. In this study, there are four potential locations from which a client can obtain a page copy: *1) local client memory, 2) local client disk, 3) server memory, and 4) server disk.* [2] The cost of obtaining a page from a location in the storage hierarchy is dependent on several factors:

1. The path length of accessing the location (e.g., the cost of sending and receiving messages, the cost of performing disk I/O, etc.)
2. Contention for those shared resources required to access the location (e.g., the network, server or client disk, server or client CPU, etc.)
3. The probability of finding the page at the location (e.g., server or client buffer hit rate).

[2] As described in Chapter 6, clients can also be allowed to obtain pages from other clients. The additional resources represented by remote clients can be fit into the framework used in this chapter; however, for simplicity this issue is not addressed here.

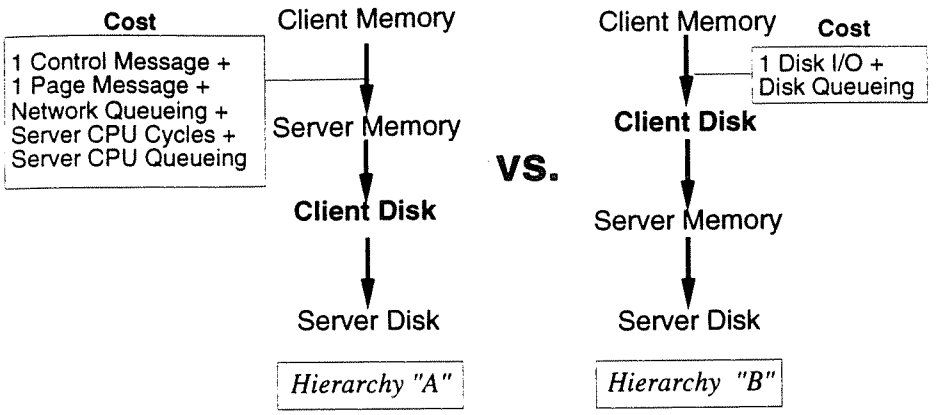


Figure 7.2: Two Possible Hierarchies

It is important to note that while the path length is relatively fixed for a given configuration, the other two components are dependent on dynamic aspects of the workload such as application mix and intensity. Therefore, in general it is not possible to determine a fixed cost hierarchy for the DBMS to traverse.

Figure 7.2 shows two possible cost hierarchies for a client-server DBMS if the issue of consistency is ignored (i.e., assuming all copies are valid). In hierarchy "A", the server's memory is assumed to be cheaper to access than the client's local disk cache, while in hierarchy "B", these two levels are inverted. As the client population changes, this inversion can (and does, as will be shown in Section 7.3) take place. In a system with few clients, there will be low contention for the server and network resources, so the tradeoff will be between the cost of a random disk I/O and the cost of a round-trip RPC to the server. With current technology, the access to the server memory will likely be less expensive than the disk I/O. However, as clients are added, contention for the network and the server will increase. Eventually, the cost of a local disk I/O will fall below the expected cost of a remote memory access.

Despite the dynamic nature of the hierarchy costs, there are some relationships that remain fixed for a given configuration. For example, the local client memory is always the least expensive level, and it is always checked first to determine if the page is already available to the transaction. Likewise, the server memory is always cheaper to access than the server disk. In addition, if the server disks and client disks have the same performance characteristics, then the server disk is always the most expensive location.

Consistency Maintenance

Consistency maintenance ensures that transactions execute in a serializable manner despite the presence of replicated copies of data pages. As described in Section 4.2, the two basic mechanisms for performing consistency

maintenance are *detection and avoidance*. Detection schemes allow stale copies of data to remain in client caches; the validity of a page copy must be confirmed before that copy can be accessed by a transaction, and thus, attempted access to invalid pages is detected and disallowed.

In contrast, avoidance-based schemes ensure that all accessible copies of a data page are valid so that access to invalid data is avoided. This can be done by *invalidating* other replicas of an updated item (as is done by CB-R for memory cache contents), or by *propagating* the new data value to the other replicas. The results of Chapter 5 showed that for memory-based caching, invalidation performs better than propagation under most workloads. Propagation preserves replication, thereby increasing the cost of updates. In contrast, invalidation destroys replication of read-write shared data so that subsequent updates incur less consistency overhead. This study considers relatively large disk caches and large client populations, which would exacerbate the problems of propagation. Furthermore, propagation to a page copy in the disk cache would require disk I/O, whereas invalidation can be performed by simply modifying memory-resident structures. For these reasons, the study of avoidance-based mechanisms for disk cache consistency maintenance is restricted to algorithms that use invalidation.

Integrated Algorithms

Efficient cache management algorithms must address the interaction of hierarchy search order and consistency maintenance considerations. For example, if the server must be contacted to determine the validity of a page copy, then the cost of accessing the server memory is reduced because the required consistency checking message can also serve as an implicit request for the page. As described in the previous two sections, two options are considered for each of the dimensions in the design space. The four resulting algorithms are shown in Figure 7.3. For the hierarchy search order dimension, we investigate the tradeoffs between algorithms that differ in whether they favor accessing the local disk cache (called LD algorithms) or the server memory (called SM algorithms). In terms of consistency maintenance, all of the algorithms that are developed use the avoidance-based CB-R to manage the client *memory* caches, ensuring that the memory cache contents are always valid. This is because the study of Chapter 5 showed that avoidance generally performs better than detection for memory caches. For *disk* caches, however, we re-examine the tradeoffs between avoidance and detection. This re-examination is undertaken because disk caches are much larger than memory caches and typically hold colder data, resulting in different tradeoffs than for memory caches.

The two avoidance-based algorithms that are studied are called Local Disk/Avoid (LD/A) and Server Memory/Avoid (SM/A). Both of these algorithms extend the CB-R algorithm to ensure that all pages resident in the

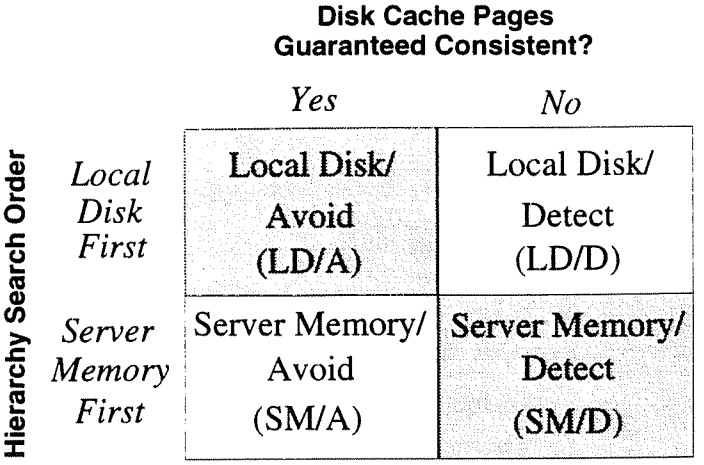


Figure 7.3: Simplified Algorithm Design Space

disk caches (in addition to those in the memory caches) are valid. Consequently, clients can read pages from their local disk cache without contacting the server. Likewise, clients must invalidate page copies from both their memory and disk caches in order to service a callback request. These algorithms extend CB-R to the disk cache contents, so the server must track the contents of both the memory and disk caches at the clients. Clients inform the server when they no longer have a copy of a page in either cache, rather than when the page is removed from the memory cache as in standard CB-R.

Using LD/A, clients request a page from the server only if that page is absent from both local caches. In contrast, SM/A assumes that the server memory is cheaper to access than the local disk; therefore, when an SM/A client fails to find a page in its memory cache, it sends a request for the page to the server. In the request, it includes an indication of whether or not it has a copy of the page in its local disk cache. If the page is resident in the server's memory, the server will send a copy of the page to the client. If the page is not in the server's memory, but a copy is resident in the client's disk cache, then the server simply tells the client to read the page from its local disk cache. LD/A and SM/A access the server's disk only as a last resort.

In contrast, the detection-based algorithms, Local Disk/Detect (LD/D) and Server Memory/Detect (SM/D), allow pages in the disk caches to be out-of-date. Therefore, clients must contact the server to determine the validity of page copies in their local disk cache. The client is allowed to fault a disk-cached page into its memory cache only if the server replies that the disk-cached page copy is valid.[3] For these algorithms, the server tracks only the contents of the memory caches at the clients. As with standard CB-R, clients inform the server when they drop a page from their memory cache, even if the a copy of the page resides in the client disk cache. As

[3]Recall that once a page is placed in the memory cache, it is guaranteed to be valid for the duration of its residency.

a result, a page in the client disk cache at a site is invalidated by the callback mechanism only if a copy of the page also resides in the memory cache at that site.

When LD/D incurs a buffer miss in a client memory cache, it checks the disk cache information to see if the disk cache holds a copy of the desired page. If so, it obtains the version number of the disk-resident copy (from the in-memory disk cache control table) and sends it to the server. The server checks to see if the client's copy is valid and if so, informs the client that it can access the cached page copy. If the client does not have a valid copy of the page, then the server obtains one (either from its memory or from its disk) and returns it to the client. SM/D works similarly, but takes advantage of its communication with the server to access the server memory first. When the server receives a validity check request for a page that is resident in the server memory cache it returns the page regardless of the status of the client's disk-resident copy. Therefore, the SM/D algorithm uses a larger return message to avoid performing a disk read at the client. As with the avoidance-based algorithms, the server's disk is the last place accessed.

7.2.3 Algorithm Tradeoffs

This section briefly summarizes the performance-related tradeoffs among the cache management algorithms. The most intricate tradeoffs that arise in this study are those between using detection or avoidance to maintain disk cache consistency. These tradeoffs are similar to those that arise in memory-only caching, but they differ qualitatively due to the higher capacity of disk caches and the potential shifting of bottlenecks when client disks are employed. There are three main tradeoffs to consider: 1) message requirements, 2) server memory requirements, and 3) effective disk cache size. In terms of messages, avoidance is most efficient when read-write data sharing is rare; consistency is maintained virtually for free in the absence of read-write sharing but requires communication when sharing arises. In contrast, detection incurs a constant overhead regardless of the amount of contention. This overhead is higher than that of avoidance under light read-write sharing, but can be lower than for avoidance when the sharing level is increased. The second tradeoff is the size of the page copy information that the server must maintain under each scheme. This information is kept memory resident, so it consumes space that could otherwise be used to buffer pages. For avoidance, the server maintains a record of each page copy residing in a client disk cache, while for detection, the server needs only to keep a list of LSNs for pages that may be in one or more disk caches.

The third tradeoff relates to a metric referred to as the *effective disk cache size*. In Chapter 5 it was observed that for memory caches, the amount of useful (valid) data that can be kept in client caches is lower for detection than for avoidance. This is because detection allows pages to remain in the cache after they become out-of-date.

Such pages are invalid, so they provide no benefit, but take up cache slots that could be used to store valid pages. Avoidance on the other hand, uses invalidation to remove out-of-date pages, thus opening more cache slots for valid pages. As a result, in the presence of read-write sharing we should expect that the detection based algorithms (LD/D and SM/D) will have a smaller effective disk cache size than the avoidance-based ones (LD/A and SM/A).

There are several other tradeoffs that affect the relative performance of the algorithms. A small tradeoff arises when choosing between using client disks for caching or not. Client memory space is required to store the information used to manage the local disk cache and this information reduces the size of the client memory cache. There are also the obvious tradeoffs between the hierarchy search orders. The SM algorithms both attempt to avoid accessing local disks. SM/D does this by using a larger response message to a validity check request when the requested page is in the server memory, while SM/A uses a separate round-trip communication with the server to obtain the page. In contrast, the LD algorithms will use a local disk first, even if there is low contention for shared resources. In particular, LD/D will ignore the presence of a page copy in the server's memory even though it has to contact the server to check the validity of the copy of the page in its disk cache.

7.3 Performance of the Extended Cache

7.3.1 Model and Workloads

The client-server caching model described in Chapter 3 has been extended to support client disk caching. This extension necessitated some changes in the model compared to the versions of the model used for for the caching and global memory management studies described previously. One change required was the addition of a mechanism to allow disk writes to be performed asynchronously at the clients and server. Asynchronous writes are necessary in order for client disk caching to be cost-effective, as the time required to synchronously write out pages that are dropped from the memory cache quickly dominates any benefit that is gained by the use of client disks. Thus, two memory pages are reserved for use as I/O write buffers at each client. Likewise, at the server, one memory page per active client is reserved for use as an I/O write buffer. Due to the large disk cache sizes that are used in this study, the client disk cache for each client must be preloaded in order to reduce the impact of a long warm-up phase on the statistics produced by a simulation run. The initial pages placed on a client's disk are chosen randomly from among the pages that may be accessed at that client according to the workload specification. The warm-up phase also affects the amount of memory that needs to be reserved for disk management information at clients. At the start of the simulation we initially reserve the maximum amount

required based on the disk cache size (assuming 20 bytes of descriptive data per disk page) and then reduce the allocation (if necessary) after running the simulation for a brief period. After the I/O write buffers and the disk information pages are subtracted, the remainder of the client's memory is available for use as a memory cache. As stated in Section 7.2.3, the server also uses memory for the structures that it uses to track page copy locations. The amount of memory reserved for this structure is changed dynamically during the simulation run, based on the number of outstanding page copies that the server must track.

In terms of the resource and overhead parameter settings used in the following experiments, they are largely unchanged from those used in the preceding chapters. The only differences are the following: first, the maximum number of clients was doubled (to 50) in order better test the scalability of the algorithms presented here. As a result, the database size was also doubled (to 2500 pages) and an additional disk was added to the server (for a total of three). The memory size of the server is set at 30% of the database size, as in the global memory management study. The client memory caches are set at 3% of the database size, so they are slightly larger than the "small setting" in the previous studies (75 pages versus 62 pages). Finally, the client disk caches are relatively large — each client disk cache can hold 50% of the active database. The ratio of the size of the memory and disk caches is on the order of what would be expected in an actual system, given the current relative prices of memory and disk.

In the remainder of this section, results are presented for the three workloads shown in Table 7.1. The characteristics of these workloads are slightly different than the ones that have been used in the previous two studies. It is clear from the previous studies that large client caches are only beneficial in workloads with low or moderate data contention. Thus, the three workloads used in this chapter have a lower write probability than the corresponding workloads described in Section 5.1.2. In addition, a somewhat different UNIFORM workload, is used here. UNIFORM-WH is a variant of the UNIFORM workload in which half of the database is read-write shared while the other half is shared in a read-only manner. Low per-client locality and the presence of read-write sharing both negatively impact the performance of caching at clients. However, large client caches should be advantageous for the large read-shared region of the database in this workload. As before, the HOTCOLD workload has a high degree of locality per client and a moderate amount of read-write sharing among clients. The high locality and read-write sharing of this workload provide a test of efficiency of the consistency maintenance mechanisms used by the algorithms. Finally, the PRIVATE workload has high per-client locality and no read-write sharing. As stated previously, this type of access is expected to be typical in applications such as large CAD systems or software development environments in which users access and modify their own portion of a design while reading from a library of shared components. As has been seen in the two previous studies, the

PRIVATE workload represents a very favorable environment for client caching.

Parameter	UNIFORM-WH	HOTCOLD	PRIVATE
TransSize	20 pages	20 pages	16 pages
HotBounds	1 to 1250	p to p+49, $p = 50(n\text{-}1)+1$	p to p+24 $p = 25(n\text{-}1)+1$
ColdBounds	1251 to 2500	rest of DB	1251 to 2500
HotAccProb	0.5	0.8	0.5
HotWrtProb	0.1	0.1	0.1
ColdWrtProb	0.0	0.1	0.0
PerPageInst (doubled on write)	30,000	30,000	30,000
ThinkTime	0	0	0

Table 7.1: Workload Parameter Meanings and Settings for Client n

Simulation experiments were run using all four algorithms described in Section 7.2.2. For simplicity however, the presentation is sometimes focused on the results for the Local Disk/Avoid (LD/A) and Server Memory/Detect (SM/D) algorithms, as these two algorithms are often sufficient to show the tradeoffs among the different approaches to consistency maintenance and hierarchy search order. In addition to the disk caching algorithms, results are also shown for the Callback-Read (CB-R) algorithm, which does not utilize the client disk caches. CB-R is used as a baseline to help gauge the magnitude of the performance gain (or in a few cases, loss) resulting from the use of client disk caches.

7.3.2 Experiment 1: UNIFORM-WH Workload

The first set of results were obtained using the UNIFORM-WH workload. In this workload (as shown in Table 7.1) all clients uniformly choose pages to access from the entire database. The pages in the first half of the database are accessed with a 10% write probability, while the pages in the other half are accessed read only. Figure 7.4 shows the distribution of page accesses among the four levels of the storage hierarchy for the LD/A and SM/D algorithms. Several trends can be seen in the figure. First, LD/A and SM/D obtain a similar (small) percentage of their pages from the client memory caches. Because the algorithms all use CB-R to manage memory caches and search the memory cache before looking elsewhere, they have similar memory cache hit rates in this experiment and in the ones that follow. Second, as would be expected, SM/D obtains more pages from the server memory and fewer pages from the local disk cache than LD/A throughout the range of client populations. Therefore, LD/A does more total (server and client) disk reads to get its pages than SM/D. Third, with one client, LD/A does not access any pages from the server memory, but then reaches a fairly stable level

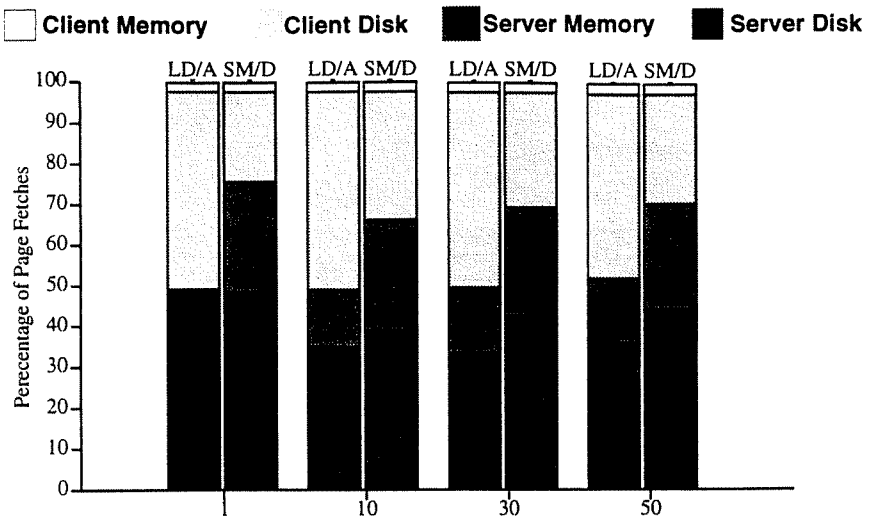


Figure 7.4: Page Access Distribution

(UNIFORM Workload)

of access. As seen in previous chapters, the initial low server memory hit rate is due to correlation between the contents of the client's caches and the server memory — with few clients, the server memory largely contains copies of the pages that are in client memories, so the server memory is less effective for serving client misses than would otherwise be expected given its size. This correlation is damped out as more clients are added to the system. Fourth, and most importantly for this experiment, both algorithms initially obtain a similar number of pages from the server disk, but as clients are added SM/D obtains more pages from the server disk than LD/A. This is due to the differences in effective disk cache size discussed in Section 7.2.3.

The performance impact of these page distributions can be seen in Figure 7.5, which shows the throughput results for this workload when run with the fast network, and in Figure 7.6, which shows the corresponding transaction response times up to 20 clients.[4] First, comparing the performance of CB-R to that of the disk caching algorithms shows the magnitude of the performance gains obtained by introducing client disk caches. As would be expected, CB-R is hurt by its relatively high server disk demands as clients are added.[5] For the disk caching algorithms, the performance results show how the dominant algorithm characteristic changes as the number of clients in the system varies. In general, at 20 clients and beyond, the avoidance-based algorithms (LD/A and

[4] Note that we use a closed system model, so throughput and response time are equivalent metrics.

[5] With one client, CB-R actually has a slight performance advantage because its client memory cache does not have to store disk cache information.

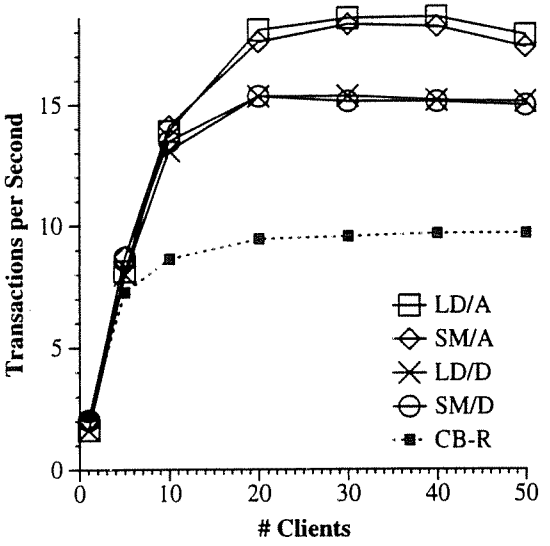


Figure 7.5: Throughput
(UNIFORM-WH, Fast Network)

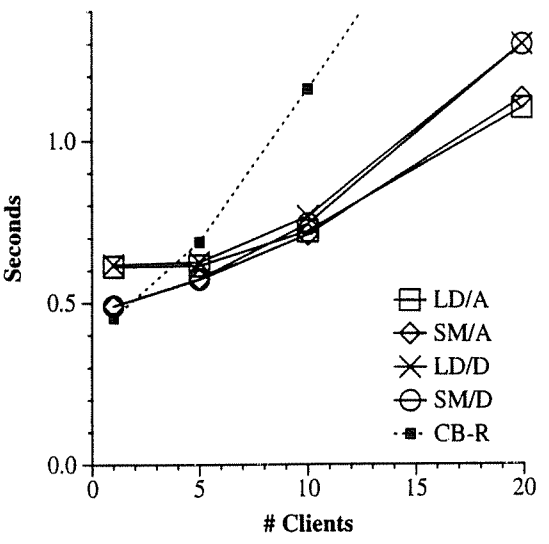


Figure 7.6: Trans. Response Time
(UNIFORM-WH, Fast Network)

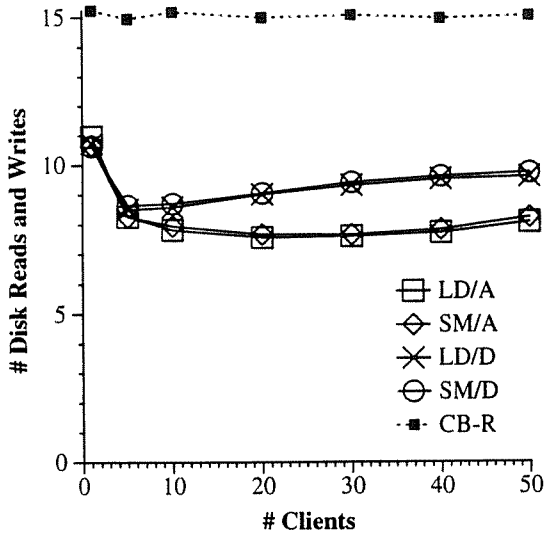


Figure 7.7: Server I/O per Commit
(UNIFORM-WH, Fast Network)

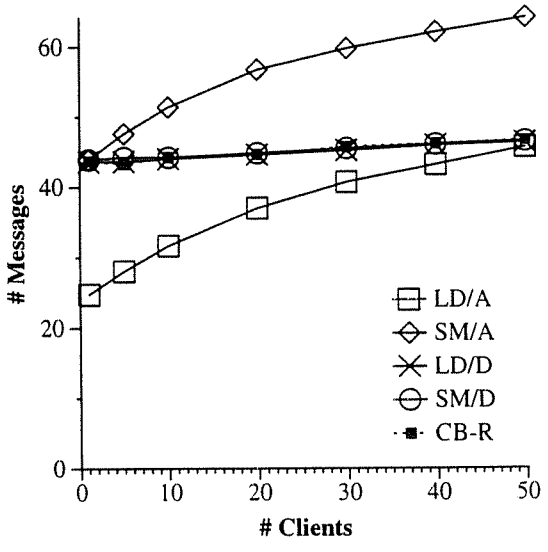


Figure 7.8: Messages Sent/Commit
(UNIFORM-WH, Fast Network)

SM/A) perform similarly, and they are significantly better than the detection-based SM/D and LD/D. However, with small client populations the dominant characteristic is the hierarchy search order, with the server memory algorithms (SM/D and SM/A) having slightly better performance than LD/A and LD/D.

The impact of the search order can be seen most clearly in Figure 7.6. In the range of 1 to 5 clients the Server Memory algorithms perform best. In this region, the server memory is less costly to access than a local disk cache because with small client populations, the (fast) network and the server are lightly loaded, and the fast network involves low on-the-wire costs for sending pages from the server to clients. The relative costs of accessing the server memory and accessing the local disk can be seen by comparing the performance of the two avoidance-based algorithms (LD/A and SM/A). The two algorithms perform the same amount of work to maintain the consistency of the disk caches and obtain the same number of pages from the client memory caches and the server disk. They differ only in the proportion of pages that they obtain from the local disk caches versus the server memory. With the fast network (Figures 7.5 and 7.6), the server memory is slightly less expensive up to 10 clients, beyond which the local disk caches are cheaper to access. When the slow network is used, (not shown) the network eventually becomes a bottleneck for the SM algorithms due to the volume of pages being sent from the server to the clients and thus, the server memory is even more expensive to access.

At 20 clients and beyond, the performance of the algorithms is dictated by the disk cache consistency approach — the avoidance-based algorithms dominate the detection-based ones. The reason for this can be seen in the number of server disk I/Os the algorithms perform per transaction (Figure 7.7). In this experiment, the server disk becomes the dominant resource, as all of the algorithms approach a disk bottleneck. As can be seen in the figure, the detection-based SM/D and LD/D algorithms lead to over 20% more disk I/Os than the avoidance-based algorithms beyond 20 clients. These additional I/Os are reads (as disk writes account for less than one I/O per transaction for all of the algorithms here). The extra server disk reads occur because the effective size of the client disk caches is substantially lower for the detection-based algorithms than for the avoidance-based ones. The reason that the effective disk cache size differences are so significant here is due to the uniform access pattern. With a uniform workload, all cache slots are equally valuable — there is no "working set" that can be kept cache-resident. Also, the uniformity of the workload results in a high degree of read-write sharing, so disk caches managed by the two detection-based algorithms will contain a large number of stale pages.

Finally, it is important to note the message behavior of the algorithms. Figure 7.8 shows the number of messages sent per committed transaction. The message counts of the detection-based algorithms (and CB-R) remain fairly constant as clients are added. They each need to send a round trip message for each page accessed by a transaction. Some of these messages are short control messages, while others (especially for CB-R) are large

messages containing page values. In contrast, LD/A initially requires fewer messages, as it contacts the server only for pages that are absent from both of the local caches on a client. However, LD/A also requires invalidation messages to be sent to remote sites when a page is updated. The number of invalidation messages that must be sent increases with the number of clients in the system for this workload. The increase in invalidations also results in an increase in the number of pages that LD/A must request from the server. Finally, SM/A incurs the combined message costs of contacting the server on each page access and of invalidating remote pages; thus, it has the highest message costs among the algorithms studied.

7.3.3 Experiment 2: HOTCOLD Workload

The next workload to be investigated is the HOTCOLD workload, in which (as shown in Table 7.1) each client has a distinct 50-page read/write "hot range" of the database that it prefers, and the hot range of each client is accessed by all other clients as part of their cold range. This workload exhibits a high degree of locality, but also has a significant amount of read-write sharing as the client population is increased. As shown in Figure 7.9, the high locality of this workload allows the majority of pages to be obtained from each client's memory cache.

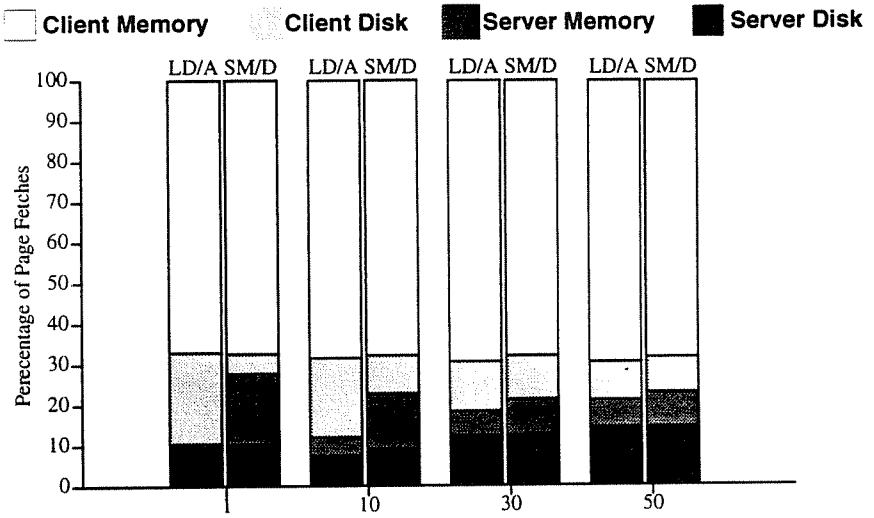


Figure 7.9: Page Access Distribution

(HOTCOLD Workload)

The figure also shows that LD/A and SM/D obtain a similar, but small, percentage of their pages from the server disk. The differences between the two algorithms are evident in the way that they split the remaining accesses.

With one client, LD/A obtains the remainder of its pages from client disk caches. As clients are added, the proportion of pages that LD/A obtains from the server memory increases somewhat. As discussed below, this occurs because adding clients increases read-write data sharing for this workload, and the effectiveness of the client disk caches decreases in the presence of such data sharing. SM/D shows different trends, with the portion of its pages coming from the server memory decreasing as clients are added. This decrease occurs because, in this type of workload, the server memory can hold fewer of the active hot set pages as the client population increases. Note that at a population of fifty clients, both algorithms have very similar page access distributions.

These distributions show the effects of the high level of read-write sharing in this workload, which increases as clients are added to the system. Regardless of the algorithm used, the update of a page at one site causes all copies of the page at other sites to become unusable. Thus, as the client population increases for this workload, the utility of the clients' disk caches decreases. For example, at 50 clients the LD/A algorithm has on average, over 1000 disk cache slots (out of 1250) empty as a result of invalidations — a disk cache larger than 250 pages will simply not be used by the avoidance-based algorithms in this experiment.

Turning to the throughput results for the fast network (shown in Figure 7.10) it can be seen that the advantages of using disk caches are lower in this experiment than in the previous one, particularly with large client populations. As in the UNIFORM-WH experiments, the Local Disk algorithms have the lowest performance for small client populations. At 20 clients, all four of the disk caching algorithms have roughly the same throughput, which is about 50% higher than that of CB-R. Beyond this point, however, the disk caching algorithms separate into two classes — and the detection-based algorithms out-perform the avoidance-based ones. This is the opposite of the ordering that was seen in the UNIFORM-WH case, and occurs despite the fact that at 20 clients and beyond, all four of the algorithms perform a similar amount of server disk I/O (Figure 7.11).[6] The reason for this behavior is due to the extra work that LD/A and SM/A perform for invalidations. As shown in Figure 7.12, the avoidance-based algorithms send significantly more messages per transaction than the other algorithms. These additional messages are largely due to the invalidation of remote disk cache pages, the impact of which can be seen, for example, in the difference between the SM/A and SM/D lines in Figure 7.12. The additional invalidation activity results in increased transaction path length. Consequently, while the detection-based algorithms both eventually become server disk-bound, the avoidance-based algorithms never reach the disk bottleneck and their performance falls off at a faster rate. The results for the slow network (not shown) are similar to these, although the Server Memory algorithms perform below the level of the Local Disk algorithms (but still above CB-R) in the range of 10 to 30 clients, due to the network cost of sending pages from the server to clients.

[6] Note that, unlike the previous case, server disk writes are an important component of the overall server I/O here.

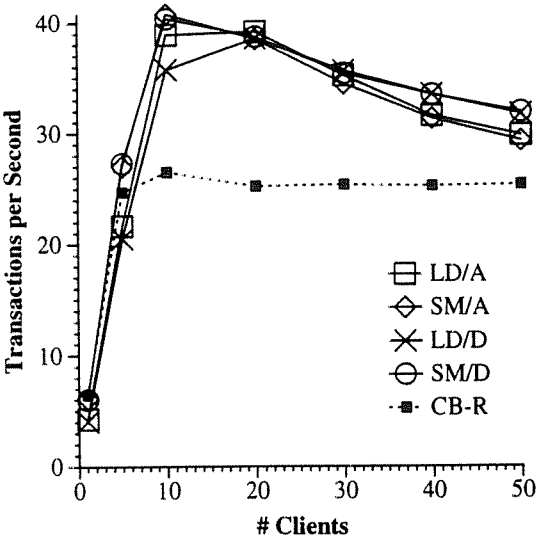


Figure 7.10: Throughput
(HOTCOLD, Fast Network)

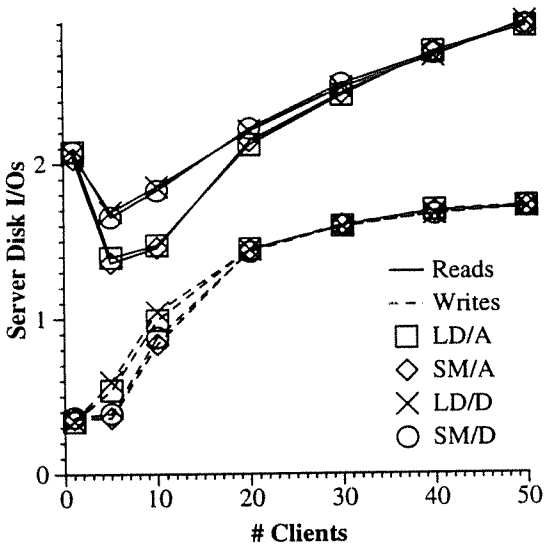


Figure 7.11: Server I/O per Commit
(HOTCOLD, Fast Network)

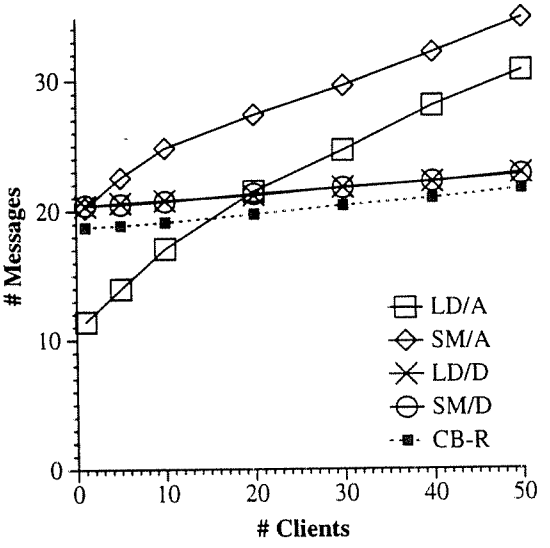


Figure 7.12: Messages per Commit
(HOTCOLD, Fast Network)

This experiment demonstrates the effect of a high degree of read-write sharing on the performance of the disk cache management algorithms and on the usefulness of client disk caching in general. This is demonstrated by the fact that the throughput for all four disk caching algorithms has a downward slope beyond 10 clients. As the level of read-write sharing is increased the disk caches become less effective. If the sharing level were high enough, then the client disk caches could actually harm performance — because they would not contribute any useful pages yet they require maintenance overhead. Therefore, it is clear that client disk caching will be most appropriate for environments in which there is a substantial amount of data that is not subject to a high degree of read-write sharing.

7.3.4 Experiment 3: PRIVATE Workload

It is anticipated that many application environments for OODBMS will have a substantial amount of data that is private or has low data contention. In this section, we investigate the performance of the alternative disk caching algorithms using the PRIVATE workload. In this workload (as shown in Table 7.1), each client has exclusive read-write access to a 25-page region of the database, and all clients share the other half of the database in a read-only fashion; thus, none of the database is read-write shared.

As can be seen in Figure 7.13, this workload presents an excellent environment for client disk caching. The

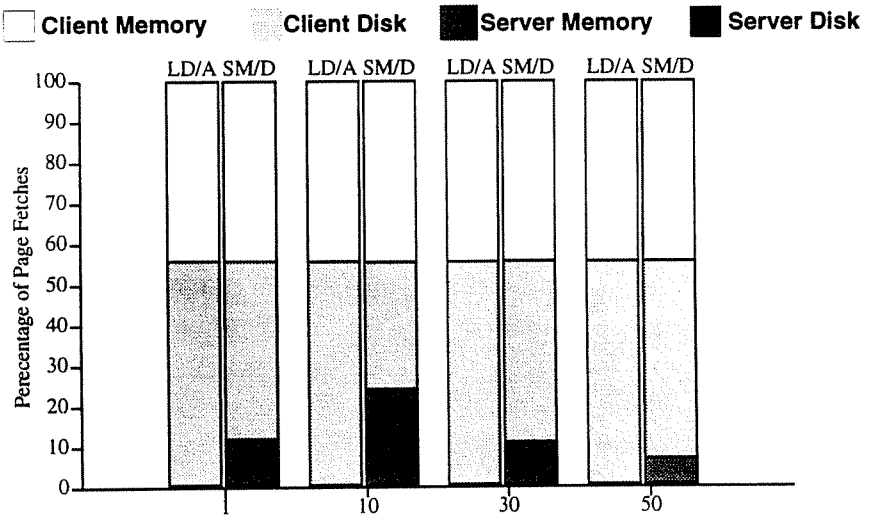


Figure 7.13: Page Access Distribution

(PRIVATE Workload)

most notable aspect of this graph is that (after system start-up) no server disk reads are required. Moreover, LD/A accesses all of its pages locally at the clients. As would be expected, SM/D behaves differently. It has an initial decrease in its locally-accessed portion; but, beyond 10 clients the locally-accessed portion increases — at 50 clients over 90% of the SM/D page accesses are satisfied locally. This is due to server-client memory correlation effects as discussed for the UNIFORM-WH workload in Section 7.3.2. As clients are added to the system, the correlation dissipates and the server memory hit rate improves. Due to the skewed access pattern of the PRIVATE workload, however, when enough clients are added that their hot ranges no longer fit in the server memory, the hit rate at the server once again decreases.

The PRIVATE workload throughput results (Figure 7.14) show that the LD/A algorithm has substantial

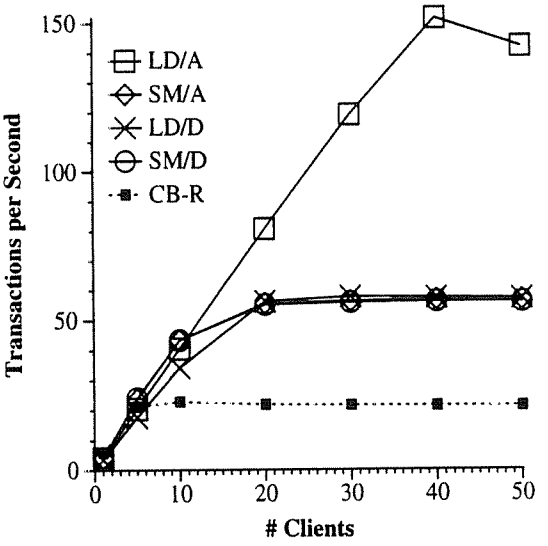


Figure 7.14: Throughput
(PRIVATE, Fast Network)

performance benefits over the other algorithms in this case. This is due to the effectiveness of the local disk caches, as described above, and because of LD/A's use of avoidance-based consistency management. In this workload, there is no read-write sharing, so no invalidation requests are required. Consequently, the avoidance-based algorithms get consistency virtually for free, while the detection-based algorithms must still check with the server on every initial access. As a result, LD/A sends only 6.5 messages per committed transaction, while the other algorithms send over 23 messages per transaction. The combination of local access and cheap consistency maintenance allows LD/A to scale almost linearly up to 40 clients here, while the other disk caching algorithms all bottleneck at 20 clients. Also, note that for this workload, CB-R flattens out at only 5 clients, at a throughput level that is less than 40% of the peak LD/D, SM/A, and SM/D throughput and less than 15% of the peak

throughput of LD/A.

In this experiment, all of the disk caching algorithms except for LD/A become bottlenecked at the server CPU due to messages, while LD/A is ultimately bottlenecked at the server disk. Recall that in this experiment, LD/A does not perform any reads from the server disk; the disk bottleneck is caused by *write* I/Os. The server write I/Os occur because pages that are dirtied by transactions are copied back to the server at commit time. These pages must eventually be written to the server disk when they are aged out of the server's memory. When the slow network is used (not shown), the copying of dirty pages back to the server also hurts the scalability of LD/A. In this case, however, the dirty pages cause the network (rather than the server disk) to become the bottleneck. Section 7.4.1 investigates ways to reduce this cost.

7.3.5 Result Summary

This section reviews the main results of the performance study of client disk caching algorithms. First, it should be noted that in all but a few cases, client disk caching provided performance benefits over the memory-only caching CB-R algorithm. The three workloads used in the study brought out different tradeoffs among the algorithms for managing client disks. In the workloads with read-write sharing (UNIFORM-WH and HOTCOLD), it was shown that with small client populations, the dominant algorithm characteristic was the hierarchy search order, with the server memory first algorithms having a slight advantage over the local disk first algorithms. With larger populations, however, the disk cache consistency maintenance approach was dominant for these workloads. For UNIFORM-WH, the avoidance-based algorithms performed best because they resulted in a larger effective disk cache size. With the fast network, there was little difference between the two avoidance-based algorithms (LD/A and SM/A), but when the slower network was used, LD/A outperformed SM/A because the local disk caches were less expensive to access than the server memory. For the HOTCOLD workload, the high level of read-write sharing in the presence of large client populations reduced the usefulness of the client disk caches, and it caused the avoidance-based algorithms to perform slightly worse than the detection-based ones because of higher message requirements. The PRIVATE workload, which has high per-client locality and no read-write sharing was seen to be an excellent workload for client disk caching. For this workload, the LD/A algorithm performed far better than the others when 20 or more clients were present in the system. This is because its bias towards using the local disk first allowed it to scale and its use of avoidance for disk cache consistency maintenance allowed it to ensure consistency virtually for free (due to the absence of read-write sharing). In fact, LD/A scaled nearly linearly with the number of clients until the server disk became a bottleneck due to writes.

7.4 Algorithm Extensions

This section addresses two additional performance enhancements for client disk caching: 1) reducing the overhead caused by copying updated pages to the server at commit-time, and 2) ways to reduce the expense of maintaining large caches that are not currently in use.

7.4.1 Reducing Server Overhead

As stated in Section 7.1.1, the server is the natural location at which to guarantee transaction semantics. The policy of copying dirty pages to the server at commit time simplifies the implementation of the server's ownership responsibilities. For example, it allows the server to easily produce the most recent committed copy of a page when it is requested by a client. However, the results of the previous experiments (particularly for the PRIVATE workload) show that for certain workloads this simplicity comes at a cost in performance and scalability. In this section, we examine the complexity and the potential performance gains that result from relaxing the commit-time page send policy. While we are unaware of any work in that has addressed this issue for client-server DBMS, it should be noted that similar issues can arise when transferring pages among the processing nodes of shared-disk transaction processing systems [Moha91, Dan92].

The following discussion assumes a system that uses a write-ahead-logging (WAL) protocol [Gray93] between clients and the server. Therefore the server is always guaranteed to have all of the log records required to reconstruct the most recently committed state of all database pages. A description of the implementation of such a protocol (i.e., ARIES [Moha92]) for a client-server DBMS can be found in [Fran92c].

Consequences of Retaining Dirty Pages

Relaxing the commit-time page send policy places certain constraints on the operation of clients. If clients are allowed to commit transactions without copying updated pages to the server, then a client may have the only copy of the most recent committed value of a page. Clients can not freely dispose of such pages. In contrast, under the commit-time page send policy clients are free to retain or drop updated pages after a transaction commits. Furthermore, allowing a client to overwrite the only valid copy of a page complicates the implementation of transaction abort; either clients will have to perform undo (which implies that they need sophisticated log management), or affected pages will have to be sent to the server to be undone.

Relaxing the policy also has implications for the operation of the server. The server must keep track of which client (if any) has the current copy of each page, and to satisfy a request for a page the server may have

to obtain the most recent copy of a page from a client. This facility can be added to the existing algorithms by using the Callback-All (CB-A) algorithm as a basis rather than Callback-Read. As described in Section 4.3, CB-A is a callback locking algorithm that allows clients to retain write intentions on pages across transaction boundaries. When the server receives a request for a page that is cached with a write intention at another client, the server sends a downgrade request to the client, asking it to revoke its write intention on the page. If the commit-time page send policy is enforced, the client needs only to acknowledge the downgrade to the server, the server can then send its own copy of the page to the requester. If the policy is relaxed, CB-A must be changed so that in response to a downgrade request, a client will send a copy of the page to the server along with the acknowledgement. When the server receives the new page copy it installs it in its buffer pool and sends a copy to the requester.

This scheme works, of course, only if clients are always available to respond to downgrade requests from the server. Under the commit-time page send policy, the server has the option of unilaterally deciding that a client's cache contents are invalid and deciding to abort outstanding transactions from a non-responsive client. Without the commit-time page send policy, however, this could result in the loss of committed updates. In a system that uses write-ahead-logging this problem can be solved by taking advantage of the server's log. To do so, however, requires an efficient way of performing redo on individual pages while normal processing is underway. One possible implementation would be to link all of the log records pertaining to a particular page backwards through the log, and to have the server keep track of the most recent log record for each outstanding page. If a page is needed from an unresponsive client, the server can then perform redo processing on its copy of the page by scanning backwards through the linked list of log records for the page to find the first missing update. It can then process those records in the forward direction, redoing the missed updates.

A related problem has to do with log space management. The log is typically implemented as a circular queue in which new records are appended to the tail and records are removed from the head when they are no longer needed for recovery. A log record can be removed from the head of the log if the transaction that wrote the record completed (committed or aborted) and the copy of the corresponding page in stable storage is correct with respect to the logged update and the outcome of the transaction.[7] When executing transactions, there is a minimum amount of free log space that is required in case recovery needs to be performed. If the required free space is not available, then transactions must be aborted. If clients are allowed to retain dirty pages indefinitely, then the server may be unable to garbage collect its log, resulting in transaction aborts. Therefore, dirty pages

[7] For a committed transaction the stable copy of the page must reflect the logged update. For an aborted one, the stable copy must *not* reflect the logged update.

must still be copied back to the server periodically. This can be done by having clients send back updated copies of pages that exceed a certain age threshold, or by having the server send requests for copies of pages that will soon impact its ability to garbage collect the log. Regardless of which method is employed, as a last resort the server can always apply selective redo to particular pages in order to free up log space.

Potential Performance Gains

It should be clear that substantial complexity must be incurred in order to relax the commit-time page send policy. This section examines the performance improvements that could result from relaxing this policy. The Local Disk/Avoid algorithm was extended to allow pages updated by transactions to remain dirty at clients. Two extensions have been considered: *LD/A-KeepMem*, which allows updated pages to remain dirty in a client's memory cache, but sends copies to the server when the page is demoted to its disk cache, and *LD/A-KeepDisk*, which allows dirty pages to reside in the disk cache as well as in memory. In both extended algorithms, a copy of a page is sent to the server in response to a downgrade request for the page, after which the client's copy of the page is no longer considered dirty.

Figures 7.15- 7.17 show the throughput of the original and extended LD/A algorithms for the three workloads studied in Section 7.3. In these experiments the periodic copying of dirty pages to the server for log space reclamation is not modeled, and no attempt was made to study the impact of client failures on the performance of the server. As a result, the performance gains shown in the figures are upper bounds for what could be expected (this is particularly true for the LD/A-KeepDisk algorithm). The throughput results for the PRIVATE workload show that, as expected, relaxing the commit-time page send policy can avoid the performance bottleneck that LD/A hits beyond 40 clients. LD/A-KeepMem performs about 33% fewer disk writes with 50 clients than does the original LD/A algorithm. The LD/A-KeepDisk algorithm performs no disk writes in this experiment, and thus, it scales linearly within the range of client populations studied here. Unfortunately, the results for LD/A-KeepDisk are unrealistic, because as mentioned above, pages do eventually have to be sent back to the server to allow log space to be reclaimed (and to minimize the performance impact of a client failure). However, a reasonable implementation of LD/A-KeepDisk should perform fewer server disk writes than LD/A-KeepMem does here, and thus, could still approach linear scale-up within this range of client populations.

The throughput results for the HOTCOLD workload (Figure 7.16) also show a performance gain from relaxing the commit-time page send policy. LD/A-KeepMem and LD/A-KeepDisk each benefit from a reduction in both server disk writes and server disk reads in this case. The reduction in server disk reads is due to an increase in the server memory hit rate. As it turns out, the commit-time page send policy hurts the server memory

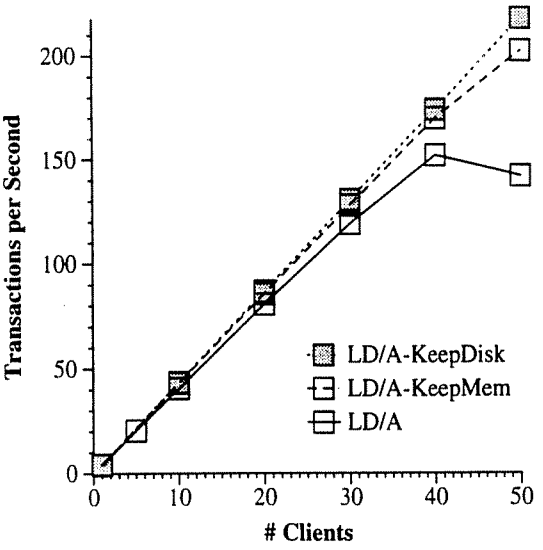


Figure 7.15: Throughput
(PRIVATE, Fast Network)

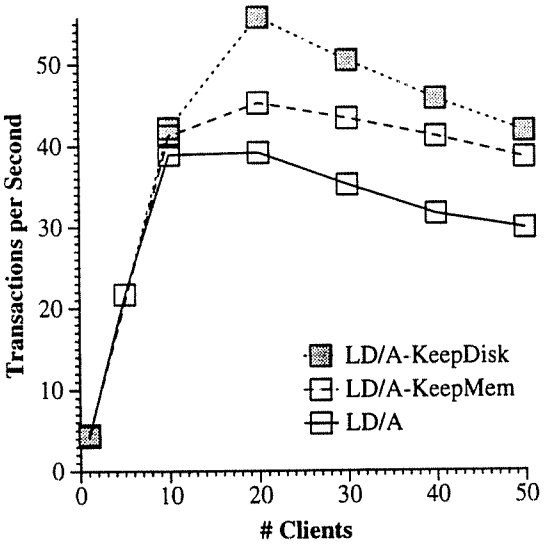


Figure 7.16: Throughput
(HOTCOLD, Fast Network)

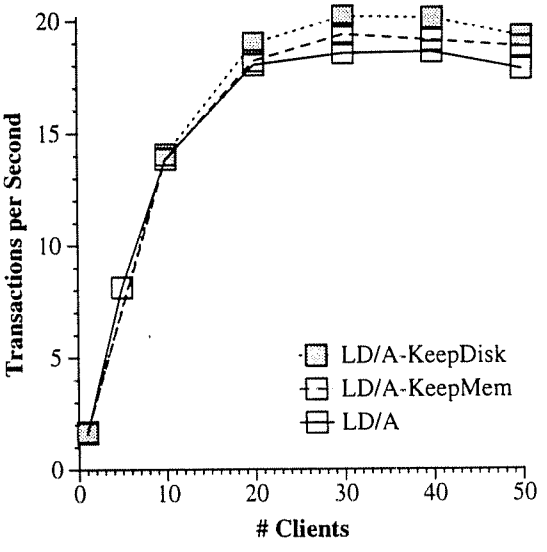


Figure 7.17: Throughput
(UNIFORM-WH, Fast Network)

hit rate because the server memory becomes filled with copies of dirty pages that are also cached at clients (this phenomenon was also observed in Chapter 6). In this experiment, LD/A-KeepDisk reaches a bottleneck at the server CPU due to messages sent for invalidations, and thus, its performance falls steeply beyond 20 clients. LD/A-KeepMem has similar message requirements, but it also has somewhat higher server disk I/O requirements, and thus it performs at a lower level than LD/A-KeepMem. Similar effects on server disk reads and writes also occur for the UNIFORM-WH workload (Figure 7.17). In this case, however, there is a smaller performance benefit to relaxing the policy. In this workload, the probability of amortizing writes due to relaxing the policy is low. This is due to the lack of per-client locality, which makes it likely that a page will be accessed at another client or dropped from memory (in the case of the KeepMem algorithm) before it is rewritten at a client. As a result, the LD/A-KeepMem and LD/A-KeepDisk algorithms do not provide a reduction in disk writes for this workload; the gains are the result of a slight reduction in server disk reads.

7.4.2 On-line vs. Off-line Caches

Up to now, this study has focused on systems in which all of the clients actively use the database. In an actual environment, however, it is likely that clients would go through periods of activity and inactivity with respect to the database. In other words, clients may be "on-line" or "off-line". One aspect of client disk caching that has not yet been addressed is how to treat the data cached at a client when the client is off-line. If the cache contents are retained across an off-line period, then when a client comes back on-line, the cache will already be in a "warm" state. The retention of client cache contents across off-line periods is particularly important for disk caches, as large disk cache contents would be very expensive to re-establish from scratch when reactivating the database system. Also, due to the low cost and non-volatility of disk storage, it is inexpensive and simple to allow the disk cache contents to persist through an off-line period. One problem that must be addressed, however, is to ensure that upon re-activation the client will not access cached data that has gone stale during the off-line period. If a detection-based approach to cache consistency maintenance is used, then the retained cache can be used as is. But, as shown in the performance results of Section 7.3, avoidance-based protocols are the recommended approach, and under such protocols, steps must be taken to ensure the validity of the cache contents after an off-line period.

The simplest approach is to have each client field invalidation requests, even during "off-line" periods. This solution requires, of course, that clients stay connected to the server. The cost of this approach is that a client process with access to the cache management data structures must be active during "off-line" periods, and invalidation messages for the pages in the inactive caches will still have to be sent by the server and processed by

the off-line clients. Such overhead is likely to be acceptable in many environments. If, however, disconnection is likely during the off-line period, or if database system overhead during off-line periods is undesirable, then there are several ways to extend the consistency maintenance techniques of Section 7.2.2 to allow cache consistency to be re-established when a client re-activates its local database system.

When a client wishes to go off-line with respect to the database, it must first return any dirty page copies to the server, save its disk cache control information on disk, and then inform the server that it is going off-line. An *incremental* approach to re-establishing cache consistency can be easily implemented by combining detection and avoidance techniques. As part of the process of going off-line, a client marks all of its disk cache pages as "unprotected". When a client is on-line, it must treat any unprotected cached page as if a detection-based consistency maintenance algorithm was being used, checking validity of the page with the server. Once the validity of a page is established, the page is marked as "protected" and thereafter, can be accessed using the normal avoidance-based protocol.

An alternative approach to re-establishing cache consistency is to have the server keep track of updates to the pages that are resident in off-line caches. When an off-line client wishes to come back on-line, it sends a message to the server, and the server responds with information that allows the client to re-establish the validity of its cache contents. This information can be: 1) a list indicating the pages to invalidate, 2) actual log records for the updates that were applied to the off-line pages by other clients, or 3) copies of the pages that changed during the off-line period. The tradeoffs between option 1 and the others are similar to those that arise when deciding between invalidation or propagation of updates for on-line caches (as discussed in Section 7.2.2). Therefore, we expect that sending a list of invalidations will typically have the best performance.

The incremental approach has the advantage that clients can begin processing immediately after coming on-line; however, it has the disadvantage of having to contact the server for validity checks (assuming the use of the LD/A algorithm, for example), so performance may be reduced during the initial period after coming back on-line. In contrast, the all-at-once approach has the advantage that the disk caches are quickly cleared of invalid data, resulting in a larger effective disk cache. Also, less communication with the server is required to re-establish to cache consistency than with the incremental approach. The disadvantages of the all-at-once approach are that the server needs to perform bookkeeping for off-line clients, and that there will be a delay between the time that a client comes back on-line and the time that it can begin processing database transactions.

Several projects have performed work on deferred consistency maintenance that is related to the work here. The ADMS± system [Rous86] uses an incremental method to update query results that are cached at clients. Updates are performed at clients prior to executing a query at a client. As discussed in Section 7.1.1, the

performance of a similar scheme is studied in [Deli92]. Techniques to reduce update overhead for caches in information retrieval systems are addressed in [Alon90]. These techniques are based on *quasi-copies*; i.e., copies whose values are allowed to diverge from the value of the primary copy. Finally, distributed file systems that support disconnected operation, such as CODA [Kist91] and Ficus [Guy91], must address issues of validating local caches after a disconnected period. Such systems do not support transaction semantics, however, and require user-interaction to resolve some classes of conflicts.

7.5 Conclusions

This chapter has demonstrated that client disks can provide substantial performance benefits when employed in an extended cache architecture. Four alternative algorithms (all based on CB-R) were described and studied. The performance study showed that for small client populations, the server memory was typically less expensive to access than the local client disk caches, but that this order inverted as clients were added to the system. In terms of disk cache consistency maintenance, algorithms that avoid access to stale data through the use of invalidations were found, in most cases, to perform better than algorithms that detect access to stale pages. This was due primarily to a larger effective disk cache size for avoidance-based algorithms. As expected, however, under high levels of read-write sharing, the larger number of messages due to consistency maintenance caused the performance of the avoidance-based algorithms to suffer somewhat. However, in the cases where avoidance was seen to perform worse than detection, the relative advantage of disk caching in general was low for all of the algorithms. It is expected that disk caching will have the greatest benefits in configurations with large client populations and low levels of read-write sharing. For such environments, Local Disk/Avoid (LD/A) appears to be the most promising of the four algorithms. Dynamic extensions of LD/A can be developed to better handle cases with small client populations.

The effectiveness of client disk caching in reducing the demand for server disk reads resulted in an increase in the relative impact of server disk writes on performance. To address this, ways of relaxing the policy of sending copies of updated pages to the server at commit-time were investigated. Although relaxing the policy has complexity implications for database system implementation, it appears that many of the problems can be solved by extending standard write-ahead-logging techniques. Using simple extensions of the LD/A algorithm, it was seen that the potential performance and scalability benefits of allowing updated pages to remain dirty in client memory caches and/or disk caches can be quite high. Another issue raised by the large capacity of client disk caches is the importance of preserving the cache contents across periods of database system inactivity.

Several approaches that allow clients to re-establish the validity of their cache contents after an "off-line" period were described.

The use of client disk caches is related to the preceding chapter's work on global memory management, as both techniques utilize client resources to offload the server disk. Client disk caching is likely to be easier to add to an existing system because the disk is treated as an extension of the memory cache, while global memory management requires new communication paths. Also, client disk caching is likely to be more effective than global memory in situations where clients access primarily private data. In contrast, global memory may be more useful for situations in which much data is shared among the clients — particularly if clients often read data that is written by other clients. The two techniques are complementary and could be integrated in a single system. To do so, however, several interesting performance issues will need to be addressed. For example, it is not obvious where to place the various remote client resources in the hierarchy search order. This is one potential avenue for future work.

Longer-term future work is to investigate the opportunities raised by the non-volatility of client disks. In particular, disconnected operation is one such area, as the off-line/on-line issues raised in Section 7.4.2 will be of prime importance in an environment where clients can disconnect from the rest of the database system. Another interesting avenue for future work is an investigation into the issues raised by the implementation of logging and crash recovery algorithms that enable the relaxation of the commit-time page send policy.

Chapter 8

Conclusions

8.1 Summary of Results

The confluence of two trends has raised a new set of performance opportunities and challenges for the design of workstation-based database systems. First, the rapid improvement in the price/performance characteristics of workstations, servers, and local-area networks has enabled the migration of sophisticated database function from machine rooms to desktops. As a result, networks of high-performance workstations and servers have become an important target environment for the current generation of commercial and prototype database systems. At the same time, the demands of non-traditional application environments have resulted in the development of a new class of object-oriented database systems. This thesis has investigated several of the fundamental architectural and algorithmic issues that must be understood before high performance, scalable object-oriented database systems can be constructed in a client-server environment.

The work in this thesis has been based on the following premises:

1. The key to performance and scalability in client-server database systems is the exploitation of the plentiful and relatively inexpensive resources provided by client machines.

2. The use of *client caching* is the most effective way to exploit client resources without sacrificing availability in a client-server DBMS.

The initial chapters of the thesis motivated and justified the above premises. Architectural alternatives for client-server database systems were described and the arguments for adopting a page server approach were presented. Chapters 3 and 4 were dedicated to an investigation of client cache consistency algorithms. Such algorithms serve as the foundation on which much of the design of a client-server DBMS rests. A taxonomy

was developed, which categorized the algorithms that have been developed as part of the research for this thesis as well as algorithms that have appeared in the literature. The taxonomy provides insight into the tradeoffs inherent in cache consistency maintenance. At its coarsest level, the taxonomy divides the design space into detection-based and avoidance-based algorithms. The decision made along this dimension was shown to have wide-ranging implications on the other tradeoffs, such as optimism vs. pessimism and propagation of updates vs. invalidation.

The bulk of the thesis consists of three performance analyses of algorithms related to caching and memory management. Cache consistency algorithms were the subject of the first study presented. Seven algorithms from three families were analyzed using a range of workloads and configurations. The algorithms were examined under different types and amounts of data sharing, locality, and information flow. The performance study enabled the quantification of the design tradeoffs that were identified in the taxonomy. The results of the study show that: 1) client caching can provide significant performance and scalability benefits, 2) invalidation of remote copies typically outperforms propagation-based approaches to consistency maintenance, 3) propagation performs well in certain environments, but it is extremely sensitive to configuration and workload characteristics, such as client cache size, 4) a simple heuristic is able to provide the benefits of invalidation where appropriate, while also performing well in situations where propagation is beneficial, 5) avoidance-based techniques combined with pessimism, as in the callback locking algorithms, provide good performance, high robustness in the presence of data contention, and lower abort rates than more optimistic algorithms. The remaining two performance studies were based on callback locking as a result of the conclusions drawn from this first study.

The second performance study examined the gains that can be obtained through the further use of client memory resources in order to avoid disk access at the server. A technique called "forwarding" allows client page requests to be satisfied by copies that are cached at other clients. Forwarding exploits copy information that is maintained by the server for cache consistency maintenance. The use of forwarding creates a global memory hierarchy which raises new issues of efficiency for memory management. Two additional techniques were proposed to address these global issues in a manner that does not impact the cache management policies of individual clients. The performance results showed that, as expected, forwarding can provide significant performance gains over a non-forwarding cache management algorithm. The study also showed that the additional techniques were indeed effective in keeping a larger portion of the database in memory. However, while these techniques achieved their objectives, they did not always yield performance improvements and in some cases they were even detrimental to performance. Thus, the extended techniques should be used only in limited situations, or else they must used in an adaptive fashion.

The third performance study investigated the use of client disks for caching. Four algorithms combining different approaches to consistency maintenance and hierarchy search order were proposed. The study showed that for small client populations, the server memory was typically less expensive to access than the local client disk caches, but that this order inverts as clients were added to the system. Therefore, for scalability reasons, local disk caches should typically be accessed first. In terms of disk cache consistency maintenance, algorithms that avoid access to stale data through the use of invalidations were seen, in most cases, to perform better than algorithms that detect access to stale pages in the disk cache. The effectiveness of client disk caching in offloading the server disk for reads can result in server writes becoming the next bottleneck. Solving this next problem raises some interesting issues in the design of the logging and recovery systems for a page server DBMS. Finally, the chapter discussed the potential raised by the non-volatility of client disks, such as the ability to support long-lived client caches.

8.2 Future Work

The factors that motivated the initiation of this work nearly four years ago are still gaining momentum. Ongoing trends in commercial computing are accelerating the move from centralized data management to network-based or even mobile environments. The current state-of-the-art in database systems provides neither the performance nor the flexibility required by applications in such environments. Consequently, efficient techniques for managing data in dynamic distributed systems will be of central importance in the coming years.

Client-server database systems that employ data-shipping, such as those studied in this thesis, represent a step towards providing high-performance database support for distributed environments. There are several remaining opportunities for interesting research and improved performance within the context of data-shipping. One issue that is directly related to the material presented in the thesis is a merger of client disk caching and global memory management into a unified framework for utilizing client resources. Another important problem to be addressed for data-shipping systems is efficient index maintenance. Indexes, which require highly concurrent access, are amenable to special consistency protocols because they are encapsulated by the database system. Finally, there is potential for further performance improvements through the use of asynchronous information flow for propagating hints about copy locations and status among clients and servers.

The next step in the evolution of client-server databases is the development of adaptive, hybrid systems that combine data-shipping for navigational data access with query-shipping for efficient processing of associative queries. Such hybrid systems would combine the benefits of utilizing client resources with the communication

savings resulting from performing selections at the server. There are many interesting performance and complexity tradeoffs to be studied for hybrid systems. Another direction that is becoming increasingly important is the exploitation of idle resources on the network. The techniques developed in the thesis can serve as a starting point for this work. For example, the global memory management techniques could be modified to exploit the memory of idle workstations, thus expanding the aggregate memory available to active client DBMS processes. A more ambitious approach is to combine client-server systems with techniques from shared-nothing parallel database systems. Extensions of the local disk caching methods proposed in the thesis, particularly the ability to support large, off-line caches, could enable computationally intensive queries to be run in a distributed fashion on the network.

On a broader scale, it is likely that the performance and utility of distributed database systems will ultimately be limited by the constraints imposed by the desire to support serializable transactions. Progress in distributed operating systems and file systems has shown that some application environments can tolerate reduced consistency or the need for user intervention in resolving conflicts. Much work remains to be done in the area of making database mechanisms more lightweight. Promising areas of research include flexible transaction and cache consistency mechanisms that provide multiple levels of consistency guarantees, new data access paradigms, such as digital libraries, and support for new modes of data manipulation, such as disconnected operation.

Bibliography

[Adve90] Adve, S., Hill, M., "Weak Ordering — A New Definition", *Proceedings of the 17th International Symposium on Computer Architecture*, May, 1990.

[Agar88] Agarawal, A., Simoni, R., Hennessy, J., Horowitz, M., "An Evaluation of Directory Schemes for Cache Coherence", *Proceedings of the 15th International Symposium on Computer Architecture*, Honolulu, June, 1988.

[Agra87] Agrawal, R., Carey, M., Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Transactions on Database Systems*, 12(4), December, 1987.

[Alon90] Alonso, R., Barbara, D., Garcia-Molina, H., "Data Caching Issues in an Information Retrieval System", *ACM Transactions on Database Systems*, 15(3), September, 1990.

[Alsb76] Alsberg, P., Day, J., "Principles for Resilient Sharing of Distributed Resources", *Proceedings of the 2nd International Conference on Software Engineering*, IEEE, San Francisco, 1976.

[Arch86] Archibald, J., Baer, J., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, 4(4), November, 1986.

[Atki89] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S., "The Object-Oriented Database System Manifesto", *1st International Conference on Deductive and Object-oriented Databases*, Kim, W., Nicolas, J-M., Nishio, S., eds., Elsever Science Publishers, Amsterdam, 1990.

[Bake91] Baker, M., Hartman, J., Kuper, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System", *Proceedings of the 13th International Symposium on Operating System Principles*, Pacific Grove, CA, October, 1991.

[Bern87] Bernstein, P., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bhid88] Bhide, A., Stonebraker, M., "An Analysis of Three Transaction Processing Architectures," *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, August, 1988.

[Bhid92] Bhide, A., Goyal, A., Hsiao, H., Jhingran, A., "An Efficient Scheme for Providing High Availability", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, San Diego, CA, June, 1992.

[Bora90] Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., Valduriez, P., "Prototyping Bubba - A Highly Parallel Database System", *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March, 1990.

[Bell90] Bellew, M., Hsu, M., Tam, V., "Update Propagation in Distributed Memory Hierarchy," *Proceedings of the 6th IEEE Data Engineering Conference*, Los Angeles, February, 1990.

[Care91a] Carey, M., Franklin, M., Livny, M., Shekita, E., "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, June, 1991.

[Care91b] Carey, M. and Livny, M., "Conflict Detection Tradeoffs for Replicated Data", *ACM Transactions on Database Systems*, 16(4), December, 1991.

[Care93a] Carey, M., DeWitt, D., Naughton, J., "The 007 Benchmark", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Washington, D.C., May, 1993.

[Care93b] Carey, M., Zaharioudakis, M., Franklin, M., "Fine-grained Locking in Page Server Database Systems", *In preparation*, 1993.

[Cart91] Carter, J., Bennett, J., Zwaenepoel, W., "Implementation and Performance of Munin", *Proceedings of the 13th International Symposium on Operating System Principles*, Pacific Grove, CA, Oct. 1991.

[Catt91a] Cattell, R., "An Engineering Database Benchmark", in [Gray91].

[Catt91b] Cattell, R., *Object Data Management*, Addison Wesley, Reading, MA, 1991.

[Ceri84] Ceri, S., Pelagatti, G., *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.

[Chen84] Cheng, J., Loosley, C., Shibamiya, A., Worthington, P., "IBM Database 2 Performance: Design, Implementation, and Tuning", *IBM Systems Journal*, 23(2), 1984.

[Comm90] The Committee for Advanced DBMS Function, "Third Generation Data Base System Manifesto", *SIGMOD Record*, 19(3), September, 1990.

[Cope89] Copeland, G., Keller, T., "A Comparison of High-Availability Media Recovery Techniques", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Portland, OR, June, 1989.

[Cope90] Copeland, G., Franklin, M., Weikum, G., "Uniform Object Management", *Proceedings of the International Conference on Extending DataBase Technology (EDBT)*, Venice, Italy, March, 1990, *Lecture Notes in Computer Science #416,* Springer-Verlag, New York, 1990.

[Dan90a] Dan, A., Dias, D., Yu, P., "The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment", *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[Dan90b] Dan, A., Yu, P., "Performance Comparisons of Buffer Coherency Policies", *IBM Research Report RC16361*, November, 1990.

[Dan91] Dan, A., Dias, D., Yu, P., "Analytical Modelling of a Hierarchical Buffer for a Data Sharing Environment", *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, San Diego, May, 1991.

[Dan92] Dan, A., Yu, P., "Performance Analysis of Coherency Control Policies through Lock Retention", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, San Diego, June, 1992.

[Davi85] Davidson, S., Garcia-Molina, H., Skeen, D., "Consistency in Partitioned Networks", *ACM Computing Surveys*, 17(3), September, 1985.

[Deli92] Delis, A., Roussopoulos, N., "Performance and Scalability of Client-Server Database Architectures", *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, Canada, August 1992.

[Deux91] O. Deux *et al.*, "The O2 System", *Communications of the ACM*, 34(10), October, 1991.

[DeWi90a] DeWitt, D., Futtersack, P., Maier, D., Velez, F., "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems", *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August, 1990.

[DeWi90b] DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H., Rasmussen, R., "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March, 1990.

[Dias87] Dias, D., Iyer, B., Robinson, J., Yu., P., "Design and Analysis of Integrated Concurrency-Controls", *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, England, 1987.

[Exod93] EXODUS Project Group, "EXODUS Storage Manager Architectural Overview", *EXODUS Project Document*, Computer Sciences Department, University of Wisconsin-Madison, (available by ftp from ftp.cs.wisc.edu), 1993.

[Felt91] Felten, E., Zahorjan, J., "Issues in the Implementation of a Remote Memory Paging System", *Technical Report 91-03-09*, Computer Science Department, University of Washington, March, 1991.

[Fiel88] Field, A., Harrison, P., *Functional Programming*, Addison-Wesley, Wokingham, England, 1988.

[Fran92a] Franklin, M. Carey, M.,"Client-Server Caching Revisited", *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, August, 1992, (published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds., Morgan Kaufmann, San Mateo, CA, to appear, 1993).

[Fran92b] Franklin, M. Carey, M., and Livny, M., "Global Memory Management in Client-Server DBMS Architectures", *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, B.C., Canada, August, 1992.

[Fran92c] Franklin, M., Zwilling, M., Tan, C., Carey, M., DeWitt, D., "Crash Recovery in Client-Server EXODUS", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, San Diego, June, 1992.

[Fran93] Franklin, M. Carey, M., and Livny, M., "Local Disk Caching in Client-Server Database Systems", *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, August, 1993.

[Giff88] Gifford, D., Needham, R., Schroeder, M., "The Cedar File System", *Communications of the ACM*, 31(3), March 1988.

[Good83] Goodman, J., "Using Cache Memory to Reduce Processor Memory Traffic", *Proceedings of the 10th International Symposium on Computer Architecture*, Stockholm, Sweden, June, 1983.

[Gray89] Gray, C., Cheriton, D., "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency", *Proceedings of the 12th International Symposium on Operating System Principles*,1989.

[Gray91] Gray, J., ed., *The Benchmark Handbook: For Database and Transaction Processing Systems*, Morgan Kaufmann, San Mateo, CA, 1991.

[Gray93] Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.

[Guy91] Guy, R., "Ficus: A Very Large Scale Reliable Distributed File System", *Ph.D. Dissertation, UCLA Technical Report CSD-910018*, June, 1991.

[Hagm86] Hagmann, R., Ferrari, D., "Performance Analysis of Several Back-End Database Architectures", *ACM Transactions on Database Systems*, 11(1), March, 1986.

[Henn90] Hennesy, J., Patterson, D., *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA., 1990.

[Howa88] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M., "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems*, 6(1), February, 1988.

[Hsia90] Hsiao, H., DeWitt, D., "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines", *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles, February, 1990.

[Hsu88] Hsu, M., Tam, V., "Managing Databases in Distributed Virtual Memory", *TR-07-88*, Aiken Computation Laboratory, Harvard University, March, 1988.

[Judg92] Judge, A., "A Survey of Distributed Shared Memory Systems", *Distributed Systems Group Report TCD-INT-0025 (in preparation)*, Trinity College, Dublin, Ireland, July, 1992.

[Jul88] Jul, E., Levy, H., Hutchinson, N., Black, A., "Fine Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems*, 6(1), February, 1988.

[Khos92] Khoshafian, S., Chan, A., Wong, A., Wong, H., *A Guide to Developing Client-Server SQL Applications*, Morgan Kaufmann, San Mateo, CA, 1992.

[Kim90] Kim, W., Garza, J., Ballou, N., Woelk, D., "The Architecture of the ORION Next-Generation Database System," *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March, 1990.

[Kist91] Kistler, J., Satyanarayanan, M., "Disconnected Operation in the Coda File System", *Proceedings of the 13th International Symposium on Operating System Principles*, Pacific Grove, CA, October 1991.

[Lamb91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D., "The ObjectStore Database System", *Communications of the ACM*, 34(10), October, 1991.

[LaRo90] LaRowe, P., Ellis, C., "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors", *Technical Report CS-1990-10*, Duke University, April, 1990.

[LaRo91] LaRowe, P., Ellis, C., Kaplan, L, "The Robustness of NUMA Memory Management", *Proceedings of the 13th International Symposium on Operating System Principles*, Pacific Grove, CA, October, 1991.

[Leff91] Leff, A., Yu, P., Wolf, J., "Policies for Efficient Memory Utilization in a Remote Caching Architecture", *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, 1991.

[Levy90] Levy, E., Silbershatz, A., "Distributed File Systems: Concepts and Examples", *ACM Computing Surveys*, 22(4), December, 1990.

[Li89] Li, K., Hudak, P., "Memory Coherence in Shared Virtual Memory Systems", *ACM Transactions on Computer Systems*,7(4) November, 1989.

[Lisk92] Liskov, B., Day, M., Shrira, L., "Distributed Object Management in Thor", *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, August, 1992, (published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds., Morgan Kaufmann, San Mateo, CA, to appear, 1993).

[Litz88] Litzkow, M., Livny, M., Mutka, M., "Condor - A Hunter of Idle Workstations", *Proceedings of the 8th International Conference on Distributed Computing Systems*, June, 1988.

[Livn90] Livny, M., *DeNet User's Guide*, Version 1.5, Computer Sciences Dept., University of Wisconsin-Madison, 1990.

[Lome90] Lomet, D., "Recovery for Shared Disk Systems Using Multiple Redo Logs", *Technical Report CRL 90/4*, DEC Cambridge Research Lab, Cambridge, MA, October, 1990.

[Moha91] Mohan, C., Narang, I., "Recovery and Coherency Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment", *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, September, 1991.

[Moha92] Mohan, C., Haderle, D. Lindsay, B., Pirahesh, H., Schwarz, P., "ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, 17(1), March, 1992.

[Nels88] Nelson, M., Welch, B., Ousterhout, J., "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems*, 6(1), February, 1988.

[Nitz91] Nitzberg, B., Lo, V., "Distributed Shared Memory: A Survey of Issues and Algorithms", *IEEE Computer*, 24(8), August, 1991.

[Obje91] Objectivity Inc., *Objectivity/DB Documentation Vol. 1*, 1991.

[Onto92] ONTOS Inc., *ONTOS DB 2.2 Reference Manual*, 1992.

[Oust85] Ousterhout, J., Da Kosta, H., Harrison, D., Kunze, J., Kuper, M., Thompson, J., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Proceedings of the 10th International Symposium on Operating System Principles*, Orcas Island, WA, December, 1985

[Pope90] Popek, G., Guy, R., Page, T., Heidemann, J., "Replication in Ficus Distributed File Systems", *Proceedings of the Workshop on Management of Replicated Data*, IEEE, Houston, November, 1990.

[Pu91a] Pu, C., Florissi, D., Soares, P., Yu, P., Wu, K., "Performance Comparison of Sender-Active and Receiver-Active Mutual Data Serving", *Technical Report CUCS-014-090*, Columbia University, 1991.

[Pu91b] Pu, C., Leff, A., "Replica Control in Distributed Systems: An Asynchronous Approach", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, June, 1991.

[Rahm91] Rahm, E., "Concurrency and Coherency Control in Database Sharing Systems", *Technical Report 3/91*, Computer Science Dept., University of Kaiserslautern, Germany, November 1991.

[Rahm92] Rahm, E., "Performance Evaluation of Extended Storage Architectures for Transaction Processing", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, San Diego, CA, June, 1992.

[Rama92] Ramakrishnan, K., Biswas, P., Karedla, R., "Analysis of File I/O Traces in Commercial Computing Environments", *Proceedings of the ACM SIGMETRICS and Performance '92 Conference*, May, 1992.

[Roth80] Rothnie, J., Bernstien, P., Fox, S., Goodman, N., Hammer, M., Landers, T., Reeve, C., Shipman, D., and Wong, E., "Introduction to a System for Distributed Databases (SDD-1), *ACM Transactions on Database Systems*, 5(1), March, 1980.

[Rous86] Roussopoulos, N., Kang, H., "Principles and Techniques in the Design of ADMS+-", *IEEE Computer*, December, 1986.

[Sand92] Sandhu, H., Zhou, S., "Cluster-Based File Replication in Large-Scale Distributed Systems", *Proceedings of the ACM SIGMETRICS and Performance '92 Conference*, May, 1992.

[Shek90] Shekita, E., Zwilling, M., "Cricket: A Mapped, Persistent Object Store", *Proceedings of the 4th International Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Morgan Kauffman, 1990.

[Sten90] Stenstrom, P., "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, 23(6), June, 1990.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Transactions on Software Engineering*, SE-5(3), May, 1979.

[Ston81] Stonebraker, M., "Operating System Support for Database Management", *Communications of the ACM*, 24(7), 1981.

[Ston90] Stonebraker, M., "Architecture of Future Data Base Systems", *IEEE Data Engineering*, 13(4), December, 1990.

[Stur80] Sturgis, H., Mitchell, J., Israel, J., "Issues in the Design and use of a Distributed File System", *Operating Systems Review*, 14(3), July, 1980.

[Trai82] Traiger, I., "Virtual Memory Management for Database Systems", *Operating Systems Review*, 16(4), October, 1982.

[Vers91] Versant Object Technology, *VERSANT System Reference Manual, Release 1.6*, Menlo Park, CA, 1991.

[Wang91] Wang, Y., Rowe, L., "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, June, 1991

[Whit92] White,S., DeWitt, D., "A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies", *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, Canada, August, 1992.

[Wilk90] Wilkinson, W., Neimat, M., "Maintaining Consistency of Client Cached Data", *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August, 1990.

[Will81] Williams, R., Daniels, D., Haas, L., Lapis, G., Lindsay, B, Ng., P., Obermarck, R., Selinger, P., Walker., A., Wilms, P., Yost, R., "R* an Overview of the Architecture", 1981, (Reprinted in Readings on Database Systems, Stonebraker, M., ed., Morgan Kauffman, San Mateo, CA, 1988).

[Wolf92] Wolfson, O., Jajodia, S., "Distributed Algorithms for Dynamic Replication of Data", *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, San Diego, June 1992.

[Zdon90] Zdonick, S., Maier, D., "Fundamentals of Object-Oriented Databases", *in Readings in Object-Oriented Database Systems*, Zdonick and Maier, eds., Morgan Kaufmann, 1990.