The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases

by

Sanjay Ghemawat

S.B., Cornell University (1987)
S.M. Massachusetts Institute of Technology (1990)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1995

Author ....
Department of Electrical Engineering and Computer Science
September 7, 1995

Certified by ....
Barbara H. Liskov
NEC Professor of Software Science and Engineering
Thesis Supervisor

Certified by ....
M. Frans Kaashoek
Jamieson Career Development Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases

by
Sanjay Ghemawat

Submitted to the Department of Electrical Engineering and Computer Science
on September 7, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Object-oriented databases store many small objects on disks. Disks perform poorly when reading and writing individual small objects. This thesis presents a new storage management architecture that substantially improves disk performance of a distributed object-oriented database system. The storage architecture is built around a large *modified object buffer* (MOB) that is stored in primary memory.

The MOB provides volatile storage for modified objects. Modified objects are placed in the MOB instead of being immediately written out to disk. Modifications are written to disk lazily as the MOB fills up and space is required for new modifications. The MOB improves performance because even if an object is modified many times in a short period of time, the object has to be written out to disk only once. Furthermore, by the time an object modification has to be flushed from the MOB, many modifications to other objects on the same page may have accumulated. All of these modifications can be written to disk with a single disk write.

This thesis evaluates the modified object buffer in combination with a number of disk layout policies that make different tradeoffs between read performance and write performance. The MOB has been implemented as part of the Thor object-oriented database. Results from simulations and from this implementation show that the MOB improves the performance of a read-optimized disk layout that preserves clustering by over 200%. As a result, the read-optimized disk layout consistently out-performs a write-optimized disk layout that does not preserve clustering. The read-optimized disk layout provides better overall performance on a wide range of workloads, including workloads that write a lot of data. Performance results also show that for typical object-oriented database access patterns, the modified object buffer architecture out-performs the traditional page based organization of server memory used in many databases and file systems.

Keywords: modified object buffer, object-oriented databases, disks, clustering, write buffers, write absorption, read-optimized, write-optimized, disk layout, installation reads

Thesis Supervisor: Barbara H. Liskov
Title: NEC Professor of Software Science and Engineering

Thesis Supervisor: M. Frans Kaashoek
Title: Jamieson Career Development Assistant Professor of Computer Science and Engineering

Acknowledgments

I thank my research supervisors Barbara Liskov and Frans Kaashoek for supporting and guiding this thesis. I am also grateful to Butler Lampson for being a careful and meticulous reader and for pointing out the importance of simple analytical models. Barbara, Frans, and Butler all greatly strengthened my thesis by providing constructive criticism and feedback.

Bob Gruber has been a good friend and wonderful colleague. Frequent discussions with him have had a significant impact on this thesis. Liuba Shrira and Jim O'Toole deserve a lot of credit for pointing out the performance benefits of delaying installation reads. Atul Adya's hard work on automated testing for Thor saved me a lot of time during the last few critical months of my work on this thesis. I am also indebted to the other members of the programming methodology group (Eui-Suk, Debbie, Quinton, Phil, Tony, Umesh, Miguel, Andrew, Mark, Dorothy, and Paul among others) for providing an excellent research environment and a friendly and relaxed work atmosphere.

Other members of the lab have also contributed to make the lab a pleasant environment. Wilson Hsieh, Anthony Joseph, and Carl Waldspurger have been here since almost from the very start. Wilson in particular has been good for countless hours of thesis avoidance. Parry Husbands has been a storehouse of mathematical knowledge. Discussions with Eric Brewer over morning capuccinos got me in the right frame of mind to start many work days. I also appreciate the can of soup from a donor who shall remain nameless.

I thank all of my volleyball teammates from the Vile Servers and the M.I.T. club team. In particular Gunter, Koji, Amy, Phil, Darcy, and Parry have been good friends and wonderful dinner companions.

Finally, I thank all of the members of my family for their love and support, and for not asking me when I was going to graduate too often.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136, in part by the National Science Foundation under Grant CCR-8822158, and by an equipment grant from Digital Equipment Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Object-oriented databases provide persistent storage for a large number of predominantly small objects. Disks perform poorly when reading and writing individual small objects. Therefore it is challenging to provide good I/O performance for such systems. This thesis presents a new storage management architecture for a distributed object-oriented database system. Simulation and performance measurements show that the new architecture out-performs other approaches.

The thesis focuses on systems that are organized as a collection of clients and servers. The servers provide persistent storage for objects. Applications run on client machines and interact with the servers to access and modify persistent objects. Each server supports many clients and therefore overall system performance is heavily dependent on the efficiency of storage management at the servers.

The new storage architecture allows servers to manage their disks more efficiently, thus improving system throughput and response time, and allowing each server to support more clients. Read performance is improved by clustering related objects on the disk and then reading an entire cluster of objects from the disk into memory when any of the objects in the cluster is accessed. Write performance is improved by maintaining a large *modified object buffer* (MOB for short) in server memory. The MOB acts as a write buffer for modified objects. Modifications are stored in the MOB and not installed on disk right away. As the MOB fills up, these modifications are moved lazily from the MOB to the disk while preserving clustering. Delaying the writing of modified objects has the benefit that multiple modifications to the same cluster of objects can be written out to the disk with one disk operation. The use of a MOB does not compromise the reliability of the system: the system provides transaction semantics by streaming modifications to a stable transaction log at transaction commit. The contents of the MOB are shadowed in the log and therefore if the system crashes, the MOB can be reconstructed on recovery by scanning the log.

The MOB improves write performance because of write absorption. If an object is modified many times in a short period of time, it is only written out to disk once. Furthermore, by the time an object modifications has to be flushed from the MOB, many modifications to other objects on the same page may have accumulated. One disk write will move all of these modifications to the disk. This thesis describes a simple analytical model of the MOB that shows that the MOB provides significant write absorption even under a workload that has no locality of access. For example, a MOB that is 10% of the database size can reduce the number of disk writes to one-third of the disk writes that would be required without a MOB. Workloads with high locality of access allow even more write absorption.

A number of experiments were run to determine the effectiveness of the MOB, both under simulation, and as part of an implementation of a distributed object database. The experiments studied a number of disk layout policies to evaluate the effectiveness of the MOB at improving write performance. The *read-optimized policy* preserves the clustering of objects on disk by writing modifications back in place. The *write-optimized policy* does not preserve clustering, but uses a disk layout that allows modifications to be written efficiently to disk. A few variants of these policies were also studied. The experiments investigated the performance of these disk layout policies in the presence of a MOB, and also compared the performance of a system with a MOB to the performance that could be achieved without a MOB.

The results show that the MOB increases the throughput of the read-optimized policy by over 200%. The results also show that the read-optimized policy in combination with a MOB significantly outperforms the write-optimized policy under a wide range of workloads, including workloads that modify large amounts of data. The write-optimized policy provides better performance only under a very limited set of conditions that are unlikely to arise in practice.

The experiments also compare the performance of an object database server organized around a MOB to the traditional page cache organization used in many databases and file systems. This thesis demonstrates that the MOB organization is more flexible than the page cache organization, and also that a system built around a MOB provides better performance under typical object database workloads. Under certain workloads a system built around a MOB has more than twice the throughput of a page cache system.

These results suggest that servers for object databases should use a read-optimized layout for good read performance, and a large modified object buffer for good write performance.

The rest of this chapter is organized as follows: Section 1.1 explains the characteristics of object databases that make efficient storage management challenging. Section 1.2 explains the operation of the modified object buffer and its interaction with other parts of the system. Section 1.3 summarizes the important results of this thesis. Section 1.4 describes the structure of this thesis and gives an overview of the contents of the rest of the chapters.

1.1 Background

Object-oriented databases have a number of characteristics that make efficient storage management difficult. The database may be very large, *e.g.*, many gigabytes, or even a few terabytes of data. Main-memory is much more expensive than disks, and therefore for many application domains, it will not be feasible to store the entire database in memory. Instead, the database must reside on disk, and pieces of the database will be brought into memory as necessary. Since disks are much slower than main-memory, efficient use of disk storage will be a crucial issue in achieving high performance.

Disk management for object-oriented databases poses a hard problem because these systems store data as objects, most of which are quite small. For example, in the OO7 benchmark database [9], most objects are smaller than 100 bytes. Reading and writing individual objects from the disk is slow because of the performance characteristics of typical disk accesses. Each disk access has two parts. In the first phase, the disk head is moved to the right position on the disk surface. This movement involves a seek latency while the disk arm is positioned at the appropriate distance from the center of the disk, and a rotational latency during which the data transfer is blocked waiting for the right data to rotate under the disk head. In the second phase, the information is transferred between the disk surface and memory. The seek and rotational latencies are independent of the amount of data being transferred, whereas the time taken to transfer the data is proportional to the amount of data. If the server reads and writes individual small objects, most of the time will be spent waiting for the disk arm to move to the right place on the disk. Only a very small amount of time will be spent actually transferring the data. As a consequence, the data transfer rate between disk and memory will be very low.

Object accesses usually follow certain patterns that can be exploited to improve storage management for persistent objects. Often, certain sets of objects are accessed together. For example, records of employees belonging to the same department may often be accessed together. Or, in a system that manages calendar information, the entries for a particular date are likely to be accessed together. Another important characteristic of such databases is that objects are read much more often than they are modified. Therefore storage management techniques should not sacrifice read performance in an attempt to improve write performance.

1.1.1 Transactions

Object-oriented databases may store complex data structures with many integrity constraints between different pieces of the database. These constraints have to be preserved in the presence of unexpected failures, as well as in the face of concurrent access by many applications. Therefore accesses to an object-oriented database are constrained to occur inside transactions [23]. Each application groups a set of object reads and writes within a transaction. The system provides atomic execution of these transactions: either the entire transaction succeeds as a unit (it *commits*), or the transaction fails (it *aborts*). Transactions may abort for two reasons: one or more of the servers, clients, or the interconnection network could fail before the transaction completes. Transactions may also be aborted if the system cannot execute the transaction without having it interfere with other concurrently executing transactions. If a transaction aborts, it has no impact on the state of the database. Furthermore, each transaction appears to execute sequentially without any interference from transactions initiated by other applications.

Transactions are usually implemented by recording the modifications performed by each committing transaction in a transaction log that is kept on stable storage at the server. This stable storage is in addition to the disk space used to store the actual contents of the database. The log is used to guarantee transaction atomicity in the presence of machine and network failures.

1.1.2 Distributed Operation

Networks can have high latencies, and may be shared by many clients and servers. Therefore it is important to reduce the amount of network communication. We assume that the system

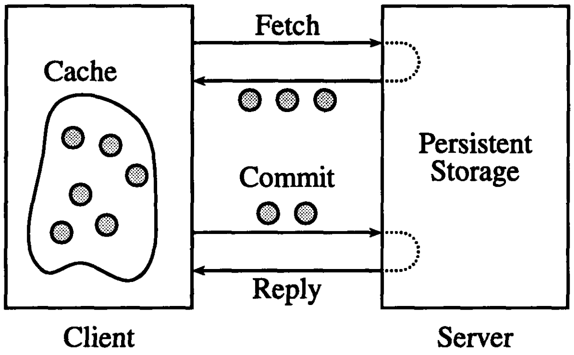


Figure 1-1: Client–server Protocol. The client has a large cache of objects. Transactions read and write cached copies of objects. At commit time all modified objects are shipped back to the server.

is organized with large client caches that hold copies of objects that the client is currently accessing. (See Figure 1-1.) Applications are organized as a sequence of transactions. Each transaction runs on a client machine and reads and modifies cached copies of objects. If the transaction needs an object that is not present in the client cache, a fetch request is sent to the server that manages the object, and the server responds with a copy of the object. At transaction commit, all of the objects modified by the transaction are shipped back to the appropriate servers. The servers determine whether the transaction should be allowed to commit, and if so, the modified objects are written to the stable log and the transaction is declared committed.

1.2 Storage Management Options

A challenge in building an object-oriented database is to provide efficient access to small objects that are not found in the client cache, as well as to allow efficient processing of modifications to these objects as transactions commit. As mentioned earlier, objects are frequently accessed in detectable patterns. Many systems exploit this fact to group objects together on the disk in ways that match these access patterns reasonably well [5, 11, 18, 53, 54, 55]. When the server has to read an object from the disk, it can read an entire cluster of related objects from the disk into memory, under the expectation that other objects in this cluster will be needed soon. This large transfer of data is less expensive per byte of transferred data because the seek and rotational latencies are amortized over a larger data transfer; therefore clustering can greatly improve the read performance of an object-oriented database system.

However, preserving the on-disk clustering of objects can be expensive: clustering requires that modified objects be written back "in place." Writing objects back in place may involve expensive disk arm movement before the modified object can be written back to the disk. We solve this write performance problem by reserving a large fraction of server memory for a *modified object buffer*. The MOB stores all committed modifications that have not been installed into the persistent copy of the database. As each transaction commits, its modifications are recorded both in the stable log, and the MOB. The modifications are not written out to the on-disk copy of the database immediately. Instead, the modifications are buffered in the MOB until the MOB starts to fill up with modifications. At that point multiple modifications are streamed from the MOB to the disk. The MOB allows multiple modifications to the same object or set of objects to be combined into a single disk write. The reliability of the system is not compromised because all of the modifications that are buffered in the MOB are also present in the stable log.

The MOB is different from the page caches used in many databases and file systems. These systems dedicate all of server memory to a page cache. (We call this server organization a PAGE organization.) Pages in this cache can contain modified objects just like the MOB, but the unit of storage is larger: the MOB stores just the modified objects while PAGE stores entire dirty pages in memory. PAGE is not well-suited for workloads where only small portions of each page are modified together. Under such workloads the MOB may perform better because it stores just the modified portion of the page whereas PAGE has to store the entire dirty page. Furthermore, in many systems clients ship back just the objects modified by a transaction, not entire dirty pages [16, 31, 34, 38]. The focus of this thesis is on such object-shipping systems. In such systems, a PAGE organization requires expensive disk reads at transaction commit to fetch the old contents of the modified pages into server memory before the modified objects can be installed into the server cache. Modified objects can be installed directly into the MOB without requiring immediate disk reads. O'Toole and Shrira report on the performance benefits of avoiding these immediate disk reads in [45].

The write-performance of a system that manages small objects can also be improved by changing the disk layout policy: instead of preserving the clustering of objects on disk, the server can efficiently stream a large number of modified objects to a previously empty space on the disk. This technique was proposed as part of the log-structured file system by Rosenblum, Ousterhout, and Douglis [46, 48]. The MOB can be used in conjunction with the *write-optimized* disk layout policy to provide even better write performance than that offered by the write-optimized layout alone. However, the MOB will have only a small impact on the performance of a system that uses a write-optimized disk layout. The main problem with a write-optimized disk layout is more likely to be its read performance (because it does not preserve the on-disk clustering of related objects).

Intermediate positions between the read-optimized and write-optimized disk layout policies are also possible. A read-optimized policy preserves the clustering of related objects on disk, say by storing related objects on the same page, and by storing each page contiguously at a fixed location on disk. A write-optimized policy does not preserve the clustering of objects because it streams out modified objects without regard to clustering. A *hybrid* disk layout policy preserves the clustering of related objects by storing the corresponding page contiguously on disk, but allowing the on-disk location of the page to change freely. This freedom in allowing pages to move on the disk can be exploited to achieve better write performance than under the read-optimized policy by streaming out a number of modified pages to a large empty region of the disk.

The write-optimized and hybrid policies use up a large empty region of the disk whenever any modifications are written out. Therefore these policies require a background *cleaner* that rearranges the contents of the database to create more free regions. Implementation of the cleaner can be complex, and the cleaner can often use a lot of available disk bandwidth and may therefore slow down client transactions [7, 48, 52].

1.3 Results

This thesis describes and evaluates a new storage management architecture built around a modified object buffer. The performance of the MOB and the effectiveness of different disk layout policies were evaluated using analytical models and by running two sets of experiments: simulations, and measurements of an implementation of the MOB in Thor, an object-oriented database system [38].

An simple analytical model shows that the MOB provides significant write absorption even under a workload that has no locality of access. For example, if the workload is completely uniform, client transactions modify 10% of a page at a time, and if the MOB size is 10% of the overall database size, the MOB reduces the number of disk writes to one-third the number of disk writes that would be required in a system without a MOB. The MOB provides even better write absorption under workloads with high locality of access.

The simulation results show that the MOB substantially improves the write performance of the read-optimized policy. The simulations also show that the write-optimized policy provides poor read performance and therefore its overall performance is much worse than the performance of a read-optimized policy in combination with a MOB. The simulations also compare the MOB to the page cache organization used is many databases and file systems. This comparison shows that under typical access patterns the MOB provides much better performance than the traditional organization.

The implementation experiments evaluate the impact of the MOB and the differences between the performance of the read-optimized and write-optimized disk layout policies. (Simulation results show that the performance of the hybrid disk layout policy is similar to the performance provided by the read-optimized policy even when cleaning costs are ignored for the hybrid policy. Therefore because of the implementation complexity of the cleaner, and potential performance problems created by running a background cleaner, the hybrid policy was not implemented as part of Thor.) The implementation experiments validate the important simulation results reported in this thesis.

The main result of this thesis is that the MOB architecture is well-suited for object-oriented databases. A server organization that uses a MOB for improving write performance and a read-optimized disk layout policy for improving read performance provides good performance under a wide range of workloads. The rest of this section gives a brief overview of the other important results described in this thesis.

1.3.1 Object-Shipping Systems

Client-server communication in distributed object-oriented databases can be organized along two general lines: clients may ship back just the objects modified by the transaction when sending a commit request to the server. Or the clients may ship back entire pages that contain modified objects. Both object-shipping and page-shipping systems are examined in this thesis, but the focus is on object-shipping systems.

The experiments show that in an object-shipping system, dedicating a large fraction of available server memory (approximately 80% for the workloads studied in this paper) to the modified object buffer significantly improves write performance under the read-optimized policy. Overall system throughput increases by more than 200% as the MOB size is increased. The performance benefits provided by the MOB are primarily because of write absorption: many modifications to the same object or group of related objects can be buffered in the MOB before the modifications are written out to the disk with a single disk write.

The MOB also improves performance under the write-optimized policy, but in that case its impact is smaller. The write-optimized policy has excellent write performance even with a very small modified object buffer. The benefit provided by a larger MOB is that more modifications are buffered in main memory and the system can therefore do a better job of sorting the stream of modifications sent to the disk in an attempt to preserve as much clustering as possible without sacrificing write performance. Therefore a large MOB can provide small improvements in the read performance of the write-optimized policy.

The performance of the hybrid policy is similar to that of the read-optimized policy. Both policies provide good read performance because they preserve the on-disk clustering of related objects. The hybrid policy is able to write out modified pages faster than the read-optimized policy: the pages are written back in place under the read-optimized policy, but under the hybrid policy many unrelated pages can be streamed out to the disk with one write operation. However, the MOB provides enough write absorption that the cost of writing out these pages is a small fraction of the total disk utilization. Therefore the overall performance gains exhibited by the hybrid policy over the read-optimized policy are minor. Furthermore, cleaning costs were ignored for the hybrid policy. These costs will degrade the performance of the hybrid policy in a real system.

The experiments also show that the write-optimized policy provides poor read performance under many workloads because it does not preserve the clustering of related objects on disk. Cleaning costs that are part of the write-optimized policy were ignored in these experiments, and therefore these experiments represent a best-case performance for the write-optimized policy. Even so, the experiments show that a read-optimized policy in combination with a large MOB out-performs the write-optimized policy on a wide range of workloads: the read-optimized policy has good read performance because of clustering, and good write performance because of the MOB. The write-optimized policy has excellent write performance but poor read performance.

The relative performance differences between the two disk layout policies diminish under certain workloads: when most of the objects in a cluster of related objects are modified together, this collection of modified objects is written out in a clustered form to the disk under both disk layout policies and therefore the read performance of the write-optimized policy improves. Also, when individual objects are themselves big, cross-object clustering becomes less important and the performance gap between the two disk layout policies narrows. This is part of the reason write-optimized disk layouts provide good read performance for file systems [48]. File contents are typically read and written sequentially in their entirety, and therefore each file behaves like one big object.

In certain configurations the write-optimized disk layout policy out-performs the read-optimized policy: if the client caches are large enough so that the vast majority of object fetches hit in the client cache, no read requests show up at the server and therefore the poor clustering provided by a write-optimized policy becomes irrelevant. Write performance becomes the dominating factor and therefore the write-optimized policy provides better performance. In the simulations described in Chapter 4, this situation arose only when the client cache size was increased to within 90% of the database size.

1.3.2 Page-Shipping Systems

The PAGE architecture that dedicates the entire server memory to a page cache does not work well in object-shipping systems [45, 56]. This thesis also examines page-shipping systems in which clients ship back entire pages at transaction commit. In such systems, the PAGE architecture can provide reasonable performance and therefore this thesis compares the MOB architecture to the PAGE architecture in the context of a page-shipping system. The results of this comparison are workload dependent.

Research on clustering and application access patterns show that writes are less common than reads, and that clustering is often sub-optimal because of limitations in clustering algorithms and variances in application access patterns [11, 55]. Therefore, we expect typical applications to only modify small portions of a page a time. Simulation results show that under such workloads the MOB provides much better performance than PAGE. PAGE has to hold entire pages in memory even if only a small portion of the page is dirty. The MOB architecture holds just the modified objects in memory. Therefore the MOB architecture can make more efficient use of a limited amount of memory and provide better write performance than PAGE. Under certain workloads, MOB provides more than twice the throughput of PAGE.

PAGE has better performance than the MOB architecture if transactions modify large portions of a page at once. Under such workloads PAGE does not waste as much memory by caching entire dirty pages, and therefore its write performance is comparable to that of the MOB architecture. PAGE has the benefit that it never has to perform any disk reads when it is lazily writing out modifications to the disk: entire pages are already in memory. The MOB architecture on the other hand has to perform extra disk reads so that individual modified objects can be installed into appropriate locations on the corresponding pages before the pages are written out to disk. These disk reads reduce the performance of the MOB architecture below that of PAGE. However, if we allow the server to fetch the page contents from the caches of other clients instead of reading these pages from the disk, the performance of MOB and PAGE is similar even in workloads where transactions modify large portions of a page at once. Therefore MOB provides good performance in general, and is better than PAGE for the typical access patterns that arise under imperfect clustering and low modification rates.

1.4 Overview

This thesis focuses on an object-shipping system in which clients ship back modified objects to servers at transaction commit. Chapter 2 contains a description of the underlying system architecture. The chapter describes the model presented to application programmers by the object-oriented database system. The chapter also describes the operations of the clients, and gives a brief overview of the implementation of the server.

Chapter 3 examines the storage management options for an object database server. The operation of the modified object buffer is described in detail. The disk layout polices studied in this thesis are also described in this chapter.

Chapter 4 describes the simulation study used to evaluate the effectiveness of the modified object buffer and its interaction with different disk layout policies. The chapter describes the simulated database, the client workload, and the simulator. Results of the simulation study are also presented in this chapter.

The preceding chapters all focus on an object-shipping system. Chapter 5 presents a simulation study of a page-shipping system. The simulations described in this chapter compare the MOB to the traditional PAGE organization that can be used in a page-shipping system.

Chapter 6 describes the implementation of the MOB as part of the the Thor object-oriented database. This chapter also contains a description of the read-optimized and write-optimized disk layout policies that have been implemented in Thor. The chapter concludes with performance measurements of Thor that validate some of the results of the simulation study.

Chapter 7 describes related research on file systems, object-oriented databases, and other more traditional databases.

Chapter 8 concludes the thesis with a synopsis of the results and suggestions for future work.

Chapter 2

System Architecture

A distributed object-oriented database system provides a model of a universe of persistent objects to application programmers. This thesis presents a storage architecture for such a system. This chapter describes the underlying assumptions about the system architecture. Our research has been done within the context of Thor [38]. Details about the implementation of the modified object buffer and the disk layout policies in Thor are deferred until Chapter 6.

2.1 System Interface

We assume that the system maps the universe of persistent objects into a programming language's heap, and therefore allows application programs to access the persistent objects in the same way that non-persistent objects provided by the programming language are accessed: by following pointers from one object to another, and by reading and writing the contents of individual objects. In Thor, each object provides a set of methods, and object accesses are encapsulated inside the methods: application code invokes methods on objects, and the implementation of the methods performs the actual object manipulation.

This thesis does not concern itself with the implementation details of providing a seamless mapping of persistent objects into the heaps of non-persistent programming languages. That work has been covered elsewhere [16, 34, 38, 40]. The storage management policies described in this thesis are useful for building the lower levels of such a database system. At this low level, objects consist of a sequence of slots. Each slot holds either uninterpreted data, such as integers and characters, or a pointer to another object. Furthermore, each object has a header that contains the size of the object as well as a structural descriptor that can be used to find the location of all references in the object. The reference information in the header is used by different parts of the run-time system: by the garbage collector to determine whether the space used by any particular individual can be reclaimed, by transaction management code to find newly persistent objects, and by server code that attempts to discover structural relationships between objects so that the server can send an appropriate set of objects in response to a fetch request. (See Section 6.2.1 for more details.)

Higher levels of the system generate an access pattern that consists of pointer traversals of the object graph as well as reads and writes of the contents of object slots. The rest of

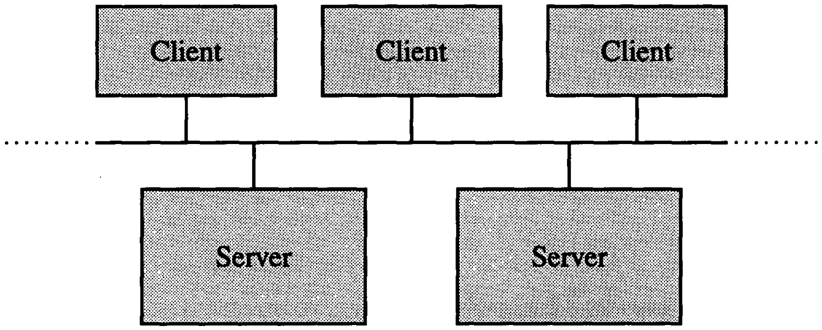


Figure 2-1: Client–server architecture

this chapter contains a description of the overall system architecture with emphasis on the lower levels of the system that provide persistent storage for objects, support pointer traversals between objects, and handle reads and writes of objects. The actual storage management and disk layout policies that are the focus of this thesis are described in Chapter 3.

2.2 Distributed Operation

The system is organized as a collection of clients and servers connected together by a network as shown in Figure 2-1. The servers provide persistent storage for objects. The applications run on the client machines and interact with the servers to access persistent objects.

2.3 Transactions

Interaction with the object database is organized using atomic transactions [23]. Each client application groups a set of object reads and writes within a boundary of a transaction. The system provides atomic execution of these transactions: either the entire transaction succeeds as a unit (it *commits*), or the transaction fails (it *aborts*). If a transaction aborts, the system guarantees that none of the object modifications performed inside the transaction have any effect on the state of the database. If a transaction commits, all of the object modifications performed within the transaction are guaranteed to survive subsequent failures. Therefore transactions provide atomicity with respect to failures of parts of the distributed system. Failure atomicity is essential for maintaining integrity constraints between different objects in the database. For example, if the system fails while processing a transaction that transfers money from one bank account to another, either both accounts are updated, or neither account is updated.

Transactions also prevent concurrently running applications from interfering with each other. Transactions appear to execute atomically with respect to other transactions. For example, in the bank account transfer described above, if the transaction subtracted an amount of 100 dollars from account *a* and then added the same amount to account *b*, no other transaction in the system would be able to observe the state where the 100 dollars had been subtracted from *a*, but not yet added to *b*. From the point of view of the client applications, each transaction appears to execute sequentially, *i.e.*, it looks as if at most one transaction is active in the entire system at any given time. However executing transactions serially in a large collection of clients and servers would be very wasteful of resources. Therefore the system allows multiple transactions to execute concurrently, but adds extra concurrency control checks that prevent different transactions from interfering with each other. Techniques for performing these checks efficiently are well-known [1, 2, 19, 24, 27].

Since transactions can use objects that belong to multiple servers, special coordination is required at the servers to ensure the atomicity properties guaranteed by the transaction model. The well-known two-phase commit protocol can be used to allow a transaction to commit its modifications atomically at more than one server [23]. The storage management issues discussed in this thesis are independent of the implementation details required to manage multi-site transactions. Therefore the experiments described in this thesis all use single-site transactions that read and modify objects from just one server. The system architecture described in this chapter can be extended to handle multi-site transactions, and in fact the Thor implementation of this architecture (described in Chapter 6) has been extended to support multi-site transactions.

2.4 Client Operation

We assume that the system uses an object-shipping architecture, *i.e.*, objects are the unit of transfer between the clients and the servers. An evaluation of the modified object buffer in the context of an architecture that ships entire pages instead of individual objects is presented in Chapter 5.

We assume that the code running at the client machine can be divided into two layers: the application layer and the run-time system. The application layer is implemented by the application programmer and invokes methods on persistent objects. The run-time system is supplied as part of the database system. The run-time system sits below the application layer and handles the mechanics of shipping objects between the client and the server as the application invokes methods on persistent objects.

Each client has a cache that is used to store recently accessed objects. Clients cache objects across transaction boundaries and read and write the cached copies of these objects. If the application code invokes a method on an object that is not present in the client cache, the run-time system detects the absence of the required object and sends a *fetch* request to the server to obtain a copy of the missing object. When the server responds with a copy of the requested object, this object is placed in the client cache and the application code is allowed to continue. If the application code uses this fetched object again, it will be found in the cache, and no fetch request will be necessary for further accesses to this object.

In addition to the cached copies of persistent objects, clients may also have volatile objects that have been created by the application, but have not been made persistent. In Thor such volatile objects automatically become persistent when the objects become reachable from a persistent object. Section 6.1.4 describes how these newly persistent objects are discovered in Thor by the client run-time system at transaction commit. In some other systems, objects are

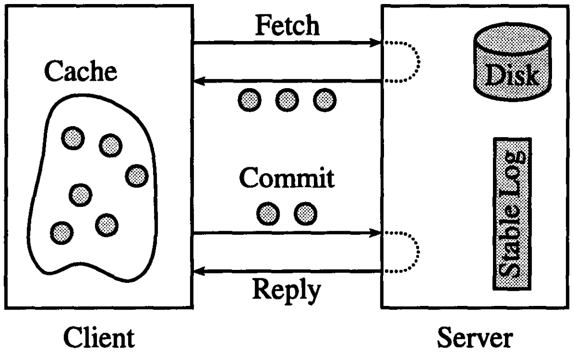


Figure 2-2: Client–server protocol

made persistent by type: certain types are marked as persistent types, and all objects of these types are automatically made persistent when the transaction that created the objects commits.

When the application requests a transaction commit, the run-time system sends the new states of all modified objects as well as any newly persistent objects to the server. If the transaction commits successfully, the client can start executing another transaction. If the transaction aborts, the client run-time system cleans up by restoring the cached copies of modified objects to the state they had before the transaction started.

Objects are discarded from the client cache either to make space for new objects being fetched into the cache, or for concurrency control reasons the server may ask the client to discard a particular object from its cache. Eviction of objects to make space in the client cache is discussed in [13]. Concurrency control issues for Thor are described in [1, 2, 27].

2.5 Client–Server Protocol

The clients send fetch requests and commit requests to the server as described in the previous section. The client–server communication protocol is illustrated in Figure 2-2. The server responds to a fetch request with a copy of the requested object. Since a transaction can access many objects, requiring a network round-trip to the server for each object that has to be loaded into the client cache will be very expensive. Therefore we assume that the server also sends back other objects that are related to the fetched object. However, since disk operations are slow and utilize limited disk bandwidth, we assume that the server only sends objects that are already present in server memory: the server never performs additional disk I/O for sending objects that the client has not explicitly requested. A description and analysis of policies for determining the related objects to be sent to the client can be found in [13].

A commit request to the server includes newly persistent objects, and the new values of all objects modified by the transaction. The server has to atomically write all of these objects values to stable storage: *i.e.*, failures in the middle of writing out these modifications should not leave the database in an inconsistent state. We assume that the system uses a stable transaction log to implement atomic updates to stable storage. In particular, we assume that the log is managed as write-ahead log: modifications are written out to the log before they are written out the persistent database itself [6].

The server appends new objects and modified objects to the end of the the stable log. The transaction commits successfully if and only if all of the modified objects reach the log. If the server crashes in the middle of writing out the modified objects, the transaction is aborted. The server replies to the client as soon as all of the modified objects are recorded in the log. The modifications do not have to be applied to the stable copy of the database before replying to the client. Instead, the modifications are written lazily to the stable copy of the database. If the server crashes before some of these modifications have been applied to the database, the database is not damaged because the modifications are still present in the stable log. When the server recovers from the failure, the log is scanned, and the most up-to-date versions of these modified objects are recovered from the log.

Transaction commits are more complicated when the committing transaction has read and written objects at more than one server. The well-known two-phase commit protocol can be used for such transactions [23]. Further details about the implementation of the two-phase commit protocol in Thor can be found in [2].

2.6 Stable Transaction Log

The storage management schemes discussed in this thesis use a stable write-ahead log to implement atomic transactions. Several techniques for implementing such a stable log are described in this section. Most database systems use an on-disk log [25]. This log can be stored either on the same disk as the rest of the database, or on a dedicated logging disk. A dedicated logging disk helps performance because it removes interference between logging and other system activity. The logging disk's head does not get repositioned by other activity and therefore whenever new data has to be logged the disk head is likely to be very close to the location at which this data has to be written out. Therefore the logging of data can occur without requiring slow disk arm movement. However, commit latency can be high even with a dedicated logging disk because of rotational latencies: up to a full disk revolution may have to occur before the transfer of data to the disk surface can start. The logging disk may also be a bottleneck to system throughput if modifications cannot be streamed out quickly enough to the disk. A throughput problem is likely to arise because most database systems write entire disk pages to the log at a shot and if an individual transaction modifies a very small amount of data, each transaction commit will require the disk write of a mostly empty page. DeWitt and others propose using *group commit* to solve this problem [17]: transaction commits are delayed for a small period of time and then modifications from different transactions are grouped together on the same page and that page is written out to the stable transaction log.

The stable transaction log can be stored in non-volatile memory (NVRAM) instead of on a disk. NVRAM is much faster than disk storage, but is also much more expensive. Baker quotes NVRAM board costs ranging from $130–$630 dollars per megabyte [3]. (The higher

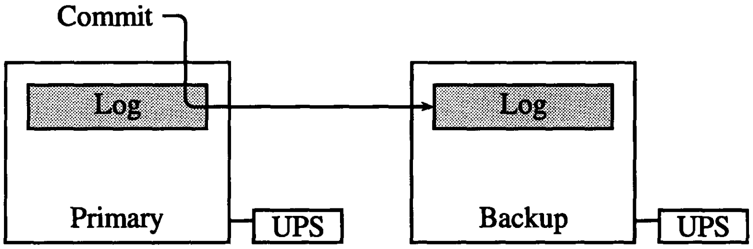


Figure 2-3: A stable transaction log implemented by replication in the memories of a primary and a backup. Un-interruptible power supplies are used to guard against power failures.

costs are for small NVRAM boards where the fixed packaging cost is amortized over a small amount of memory.) Current disk prices on the other hand are as low as 20–30 cents per megabyte. Because of the high speed of NVRAM, a system that uses an NVRAM log will provide much smaller commit latencies as compared to a system that uses an on-disk log. In fact, even a small amount of NVRAM that serves as the tail of a larger on-disk log can be used to reduce commit latencies. Modifications can be recorded in the NVRAM, and dumped out to the on-disk transaction log as the NVRAM fills up. This approach is similar to the group commit approach, but it has the benefit of not delaying transaction commits: a transaction can be considered committed as soon as its modifications are stored in NVRAM.

An alternative approach is to use replication to implement a stable transaction log (see Figure 2-3). The Harp file system uses the main-memories of multiple replicas to implement a stable log [39]. As modifications reach the primary replica, they are recorded in the volatile memory of the primary and also sent over the network to a backup replica. The transaction is considered committed as soon as the modifications are stored in the memories of both the primary and the backup. Under the assumption that the primary and the backup do not crash simultaneously, this approach provides a stable transaction log without requiring NVRAM or a logging disk. The primary and the backup have to be equipped with un-interruptible power supplies (UPS's) to guarantee against power failures that would otherwise crash both replicas at the same time. This scheme is well suited for a system in which the servers are already replicated to provide highly available storage, and each server replica has a MOB that holds recently modified objects. A stable transaction log falls out for "free" in such an organization: the combined contents of the memory at the primary and the backup can serve as the stable transaction log. Thor servers are replicated for high availability, and therefore this implementation of the stable transaction log is used in Thor. (See Chapter 6.)

2.7 Pages

We assume that the database has been partitioned into a set of *pages*. Each page is a logical storage entity that provides storage for a number of related objects. Unlike operating system pages, the pages used in our organization may have different sizes. Furthermore the size of a page may change as objects that belong to that page grow or shrink in size, or as objects are added to or removed from the page.

The partitioning of objects into pages is done by some *clustering process* that uses information about the object graph and typical access patterns to find groups of objects that are likely to be accessed together. We do not propose a particular clustering process. Instead, we assume that some unspecified clustering process has already partitioned the objects into various pages. The literature on object-oriented database systems describes a number of clustering policies that would fit into the organization described here [5, 11, 18, 53, 54, 55].

The page sizes used in our system are on the order of 32 to 64 KB. These sizes are much larger than the page sizes traditionally used in operating systems and file systems (usually around 4 or 8 KB). There are several reasons why we use larger pages than used in operating systems and file systems. Disk revolution times are decreasing and magnetic densities are increasing faster than seek latencies are decreasing. Therefore, the relative cost of large page transfers is falling.

Another reason file systems use small pages is to avoid space wastage due to fragmentation. Most file system implementations allocate an integral number of pages to each file. Therefore on average, each file wastes half a page of disk space. Systems tend to have many small files and with large pages this wastage of space will be prohibitive. The 4.3 BSD file system partially solves this problem by dividing each page into 2, 4, 8, or 16 fragments and then allocating individual fragments to different files to reduce the amount of wasted space [36]. This optimization allowed the BSD fast file system implementation to increase the page size to 4 or 8 kilobytes without significantly increasing the amount of wasted space.

Large pages do not cause fragmentation problems in our system because we treat small and large objects differently. Large objects do not waste space because our system allows different pages to have different sizes and each large object is stored on a separate page that has exactly the size required for that page. Small objects are stored by packing many such objects into a large page. Therefore the maximum amount of space wasted in a page full of small objects will be at most the size of a small object. Therefore neither large objects nor small objects create fragmentation problems in our system.

2.8 Object Location

The partitioning of related objects into pages is also exploited for efficient object location. Each object identifier has two parts: a page identifier, and an identifier that is unique within that page. The server maintains a two-level object-location map [14]. The first level of the map translates a page identifier into the location of the second level map entry for that page. The first level of the object location map is called the *page map* and each second level entry is called a *page header.* Each page header has enough information to allow the translation of the page specific portion of an object identifier into the location of that object. The page map and individual page headers are stored on disk along with all of the persistent objects. A volatile copy of the page map is also maintained in main-memory. Individual page headers are read

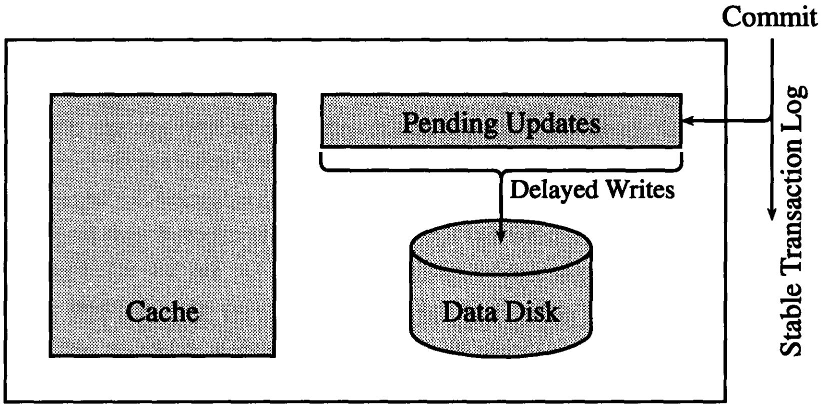


Figure 2-4: Server organization

in from the disk as necessary to handle object location requests. Chapter 3 contains a more detailed description of the object location process.

Similar two-level object-location policies have been used in other object-oriented database systems [10, 16, 31]. The benefit of this two-level location scheme is that under most disk-layout policies each page header is stored along with the objects named by that page header and therefore a single disk read suffices to fetch a number of related objects as well as the corresponding page header into memory. Furthermore, the indirection through the page header allows the system to rearrange the contents of a page as objects change in size or objects are inserted into or removed from the page.

2.9 Server Organization

The server contains some memory, disk storage, and a stable transaction log as shown in Figure 2-4. The data disk provides persistent storage for objects. The stable log holds modifications for committed transactions. The server memory is partitioned into a cache and a modified object buffer. The cache holds copies of objects that have been recently accessed by clients. The unit of storage in the cache is not an object: entire pages under the read-optimized policy, and page fragments under the write-optimized policy are stored in the cache.

The MOB holds recently modified objects that have not yet been written to the data disk. As transactions commit, modifications are streamed to the log and also inserted into the MOB. When the modifications stored in the MOB have reached the data disk, these modifications are removed from both the MOB and the log. If the server crashes while its memory contains some modifications that have not reached the data disk, then the in-memory copies of these modifications are reconstructed at recovery by scanning the log.

The server memory is partitioned into a cache and a MOB because the cache is managed differently from the MOB. The cache holds either entire pages or page fragments, whereas the MOB holds individual modified objects. Furthermore, cache contents are evicted using an LRU policy whereas MOB contents are evicted in FIFO order to allow the stable transaction log to be truncated in a straightforward manner. Details of the organization of the server memory and the different disk layout policies used for persistent storage of objects are described in Chapter 3.

Chapter 3

Storage Management Policies

This chapter describes the operation of a storage management architecture built around a modified object buffer. We focus on how objects modified by committing transactions are recorded in the MOB, how disk storage for the database is organized, and how modifications are written from the MOB to the disk. Performance measurements and analyses of the MOB and the disk layout policies are presented in Chapters 4, 5, and 6.

3.1 Modified Object Buffer

As described in Chapter 2, as modifications arrive at the server, these modifications are streamed to the stable transaction log and also buffered in server memory. The MOB storage organization reserves a portion of server memory to hold copies of objects that are also present in the stable transaction log. The rest of the server memory is used as a cache that holds pages (or page fragments in the case of the write-optimized disk layout described later in this chapter). Modifications are inserted into the MOB and the stable transaction log at commit time. Figure 3-1 illustrates this organization.

This organization contrasts with the page cache organization used in most databases and file systems. In a page cache organization, all of server memory forms a cache of database pages. At transaction commit modifications are streamed to the stable transaction log and are also installed into corresponding pages in the server cache. Therefore the MOB organization differs from a traditional organization in that a MOB holds just dirty objects whereas a traditional page cache holds entire dirty pages even when just parts of these pages have been modified. This section contains a more complete description of the operation of the MOB.

3.1.1 Object Modifications

New values of modified persistent objects and any newly persistent objects are inserted into the MOB at transaction commit time. A background *flusher* thread lazily moves modified objects from the MOB to the data disk. Once modifications have been written to the data disk, they are removed from the MOB and the transaction log, thus freeing up space for future transactions. The flusher thread runs in the background independent of any client activity and therefore

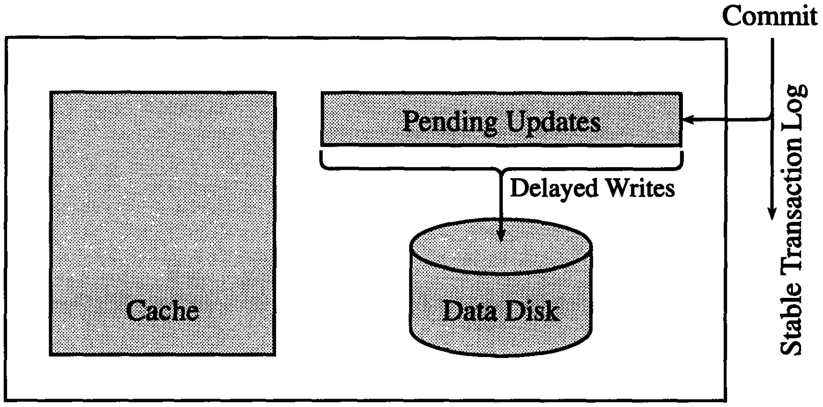


Figure 3-1: The MOB storage architecture

it does not delay commits unless the MOB fills up completely with pending modifications. The operation of the flusher thread is intimately tied to the disk layout policy and is therefore described in more detail along with the different disk layout policies later on in this chapter.

As described in Section 2.6, the transaction log may be implemented in NVRAM, or in the memory of two replicas. NVRAM and memory are expensive and therefore the log may be quite small. Objects should be flushed out from the MOB in a timely fashion to allow truncation of the contents of the log. Hence we organize the contents of the MOB itself as a log of objects. The flusher thread always writes out old objects from the MOB and as these objects are written out from the MOB, the corresponding log entries from the stable log are discarded in log order.

3.1.2 Object Fetches

As mentioned earlier, the server memory is partitioned into a cache and a MOB because the contents of the cache are managed differently from the contents of the MOB. (Cache contents are evicted using an LRU policy whereas MOB contents are evicted by the flusher thread in a FIFO manner.) An object may be present in either the MOB, or in the server cache. To satisfy a fetch request, the server first looks in the MOB, since if an object has been modified recently, its newest version is stored there. If the object is not found in the MOB, the server looks in the cache. If the object is not in the cache, the server reads the object from disk, stores it in the cache, and replies to the client when the disk read completes. As described in Chapter 6, the server may also send other related objects to the client if these objects are found in server memory (*i.e.*, in the cache or the MOB.)

3.1.3 MOB Interface

This section describes the interface provided by the modified object buffer. A simple implementation of this interface is presented in Section 3.1.4. The interface and implementation are specified in pseudo-C++. The following auxiliary types are used: Oid and Pid are object identifiers and page identifiers respectively. Log_Index is an integral type that identifies a log record. We assume that log indices increase as new log records are created, *i.e.*, newer log records have larger indices than older log records. An Object is the state of a persistent object; it includes the object identifier as well. A Modification is a structure that contains an Object, and some auxiliary indexing information that is private to the MOB. Array<T> is a generic array of elements of type T.

The MOB supports several kinds of accesses. The server processes fetch requests by searching in the MOB for the requested object. Transaction commits insert modified objects into the MOB. The flusher thread periodically scans the contents of the MOB and writes out modifications that belong to a certain set of pages. This section describes the interface used to support all of these accesses. The actual interface to the MOB used in Thor has extra complications created by the presence of multiple threads of execution and the desire to avoid unnecessary copying. These complications are mostly ignored in this "cleaned up" interface.

```
Object* fetch(Oid x)
```

If the MOB contains an object named x, then return a pointer to the object. Otherwise return nil. This routine is typically used in response to client fetch requests.

```
void insert(Log_Index l, Array<Object*> list);
```

Insert all of the objects contained in list into the MOB and associate these modified objects with the log index l. If any of the entries in list supersedes an older modification to the same object, the older modification is discarded from the MOB.

This routine is typically called when a transaction has committed.

```
Log_Index find_pages(Array<Pid>* list);
```

Find the set of pages that should be written out to disk next. This set of pages is picked so that a prefix of the transaction log can be deleted when the pages have all been written out. This routine stores the identifiers of these pages in list and returns the largest log index that can be discarded when all of these pages have been written out. All log entries with indices less than or equal to the returned index can be discarded when the writes have completed.

This routine is typically called by the flusher thread when it is ready to write some modifications to disk.

```
Array<Modification*> find_modifications(Pid page);
```

Find all pending modifications for page and return a list of pointers to these modifications.

This routine is typically called by the flusher thread to find the set of modifications that have to be installed into a page.

```
void remove(Array<Modification*> list);
```

Remove all modifications named by list from the MOB.

This routine is typically used by the flusher thread after all of the modifications in list have been written to disk. It is necessary to supply the actual list of modifications to remove instead of just the appropriate page identifiers because some other thread may have recently installed a new modification to one of these pages. This new modification will not have been written to disk if it was inserted after the call to find_modifications. Therefore we cannot blindly remove all modifications to a page once the page has been written out. Only the modifications that were actually installed into the page before it was written out can be removed from the MOB.

3.1.4 MOB Implementation

Since the MOB is scanned in log order by the flusher thread to find the oldest modifications that should be written out to memory, it is tempting to represent the contents of the MOB as an in-memory log of modified objects. However, this representation is not optimal: even though pages to be written out are identified by a scan starting with the oldest object in the MOB, modifications to these pages may be scattered throughout the MOB. All of these modifications are written out in one shot. Therefore if the MOB is represented with an in-memory log, after the page write has completed the log based representation may have holes in it. Therefore we use a more complicated data structure to implement the MOB. There are three distinct parts to this data structure: memory for modified objects, a *scan list* that is used to find the set of pages that need to be written out so that MOB contents can be discarded in log order, and an *object table* that is used to find objects given an object identifier or a page identifier.

Storage for modified objects is managed by a standard memory allocator such as malloc. As modified objects arrive at the server, space for these objects is allocated by calling the memory allocator. As objects are written out, the space used by the objects is de-allocated.

The scan list records for each log index the set of pages with pending modifications from that log index. This list is kept sorted in log index order, and can be stored in a wrap-around buffer so that new log indices can be added efficiently to one end, and old log indices can be removed efficiently from the other end. Each entry in the scan list has three fields:

```
Log_Index   index;
Pid         page;
int         count;
```

This entry says that the log record identified by index has count pending modifications to page. If a log record contains modifications to many pages, then there will be multiple scan list entries for this log record: one per page. Each Modification stored in the MOB contains an index into the corresponding entry in the scan list. As modifications are removed from the MOB, either because the modification has been written out to disk, or because a new modification has superseded an old modification, the count field of the corresponding entry in the scan list is decremented.

The object table consists of a two-level map. The first level maps from a Pid to the Modifications for that page. The second level maps from the page-specific portion of an Oid to an individual Modification. As mentioned above, each Modification contains the index of the corresponding entry in the scan list.

The fetch operation is implemented by indexing through the two levels of the object table to find the corresponding modification.

The insert operation creates appropriate entries in the scan list, and enters the supplied modifications into the object table. If any old modifications are superseded by these new modifications, those old modifications are discarded from the object table and the corresponding count fields in the scan list is decremented.

The find_pages operation searches through the scan list in log index order for any pages with a non-zero count field. Once a certain number of such pages have been found, this set of pages is returned to the flusher thread. Section 3.2.1 contains a discussion of the number of pages that are written out to disk together.

The find_modifications operation looks up the page identifier in the first level of the object table and returns the list of all of the pending modifications in the corresponding entry in the second level of the object table.

The remove operation removes the modifications from the object table and also decrements the count fields of the appropriate entries in the scan list. After all of the modifications have been processed, any entries at the beginning of the scan list that have a zero count are discarded from the scan list.

3.1.5 Discussion

Under the MOB organization, part of server memory is treated as a pool for dirty objects. There are other alternatives for the use of this memory: server memory can be used to cache database pages, and this cache may store either just clean pages, or a combination of clean and dirty pages. However, effective use of a limited amount of server memory is highly dependent on the management of memory at the client. First, because of the presence of client caches, the stream of fetch requests coming into the server cache will not exhibit much locality. The server cache is a second level cache, and in general it will not be very effective unless it is bigger than the combined cache size of all active clients [44]. Therefore, it is not worthwhile to devote a large fraction of the server memory to cache space. (Experiments described in Chapter 4 support this claim.) Instead, most of the available server memory should be used to buffer modifications so that multiple modifications to the same object or page can be handled with just one disk write.

Now the question is whether the server memory should hold entire dirty pages, or just dirty objects. The MOB architecture stores just the dirty objects in server memory. Many databases and file systems use a different memory organization that stores entire dirty pages, not just the modified portions of these pages. Let us call an architecture with this organization of server memory a PAGE architecture. The correct solution for server memory management again depends on the interaction between clients and servers. If the clients ship back just the objects modified by the transaction, a PAGE architecture will require a disk read of the modified page if the page is not already present in the server cache. We use O'Toole and Shrira's terminology and call this disk read for installing modifications into a page an *installation read* [45]. In a MOB architecture these installation reads can be delayed until the contents of the MOB have to be flushed to disk. As experiments described in Chapter 4 show, and as O'Toole and Shrira

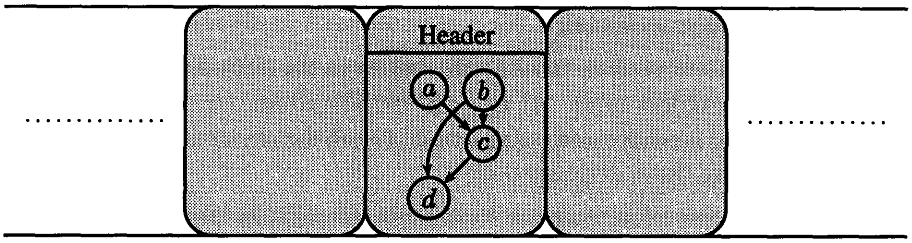


Figure 3-2: Read-optimized disk layout

demonstrated in [45], delaying these installation reads can reduce disk utilization and improve system performance.

Therefore, if the clients ship modified objects as opposed to modified pages at transaction commit, the MOB architecture will outperform the PAGE architecture. There are other differences between object-shipping systems and page-shipping systems such as the effectiveness of client caching, and the use of network bandwidth. However these issues are not directly related to this thesis. The reader can find detailed comparisons of object-shipping and page-shipping in [13, 56],

The focus of this thesis is on object-shipping systems in which clients ship just the objects modified by the transaction, and not entire pages. In Chapter 5 we relax this assumption and compare the performance difference between the MOB architecture and the PAGE architecture in the context of a page-shipping system. The rest of this chapter, and the rest of this thesis except for Chapter 5, focuses on object-shipping systems.

3.2 Disk Layout Policies

This section contains a description of the different disk layout policies examined in this thesis. These policies differ in the tradeoff they make between read performance and write performance. This section describes the overall operation of these layout policies. A qualitative comparison of the policies is presented in Section 3.2.4. Simulations and implementation results comparing these policies are presented in Chapter 4 and Chapter 6.

3.2.1 Read-Optimized Disk Layout

The read-optimized disk layout policy keeps related objects clustered together on the disk in an attempt to improve the overall read performance of the database. Pages serve as the unit of disk storage. A page and its header are stored contiguously on disk at some fixed location (see Figure 3-2). The server cache also stores entire pages with their headers. When a fetch request for an object comes into the server, the object's entire page (including the header) is read into the server cache if necessary. The on-disk location of the page is determined by consulting a volatile copy of the page map. The object is located in the cached copy of the page by looking up its page-specific identifier in the page header. If the client fetches several objects from the

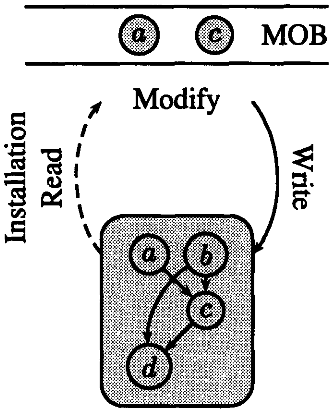


Figure 3-3: Writing modifications to the disk under the read-optimized disk layout policy

same page within a fairly short period of time, a single disk read is sufficient to satisfy all of these fetch requests.

The server cache is organized as a page cache. An object cache can provide potentially better hit ratios by discarding unused portions of a page from server memory. However, as will be shown later in Chapter 4, large server caches do not help performance and therefore server caches should be fairly small. The server cache mostly serves just as a holding area for objects until the client can fetch all relevant objects into its own cache. Since in the read-optimized policy entire pages are read from the disk, and the contents of the page do not have to stay in the server cache very long, there is no point in splitting up the page into individual objects. It is simpler to manage the server cache as a cache of pages.

The flusher thread introduced in Section 3.1.1 preserves the on-disk clustering by writing objects back in place. The flusher thread uses a read-modify-write cycle for flushing MOB contents to the disk: a page with pending modifications is read into the server cache, any pending modifications are installed into the page, and the new value of the page is immediately written back to disk. This read-modify-write process disk is illustrated in Figure 3-3, where the MOB contains objects *a* and *c* from the same page. The flusher thread processes many pages at a time. It uses the `find_pages` operation of the MOB to identify a set of pages with pending modifications. The flusher thread sorts this set of pages by disk address and then processes any pending modifications for each page. As a page is processed, it is read into the cache if necessary. All pending modifications to the page are installed into the cached copy, and the page is written back in place to the disk. When a page write has completed, the installed modifications are removed from both the MOB and the stable log.

This read-modify-write cycle for flushing modifications is different from the disk scheduling used by O'Toole and Shrira [45]. In their system, a large number of pending modifications is treated as a source for disk operations for the disk scheduler. However treating the entire MOB as input to the disk scheduler has several drawbacks. First, the scheduler may tie down a large amount of server memory by performing a large number of installation reads before performing any disk writes. (An installation read allocates storage for a page, and the corresponding disk write releases this storage.) Second, scheduling out of the entire MOB will reduce the amount of write absorption: some modifications may be written out of the MOB very soon after their insertion into the MOB. Third, fetch requests from clients may generate disk reads that disrupt a long pre-computed schedule for installation reads and writes. Furthermore, the analysis at the end of this section shows that the scheduling benefits of the MOB are minor even without this interference.

Therefore, the flusher thread uses a read-modify-write cycle and we start the flusher thread only when the MOB is 99% full. On each invocation, the flusher thread scans through the oldest 1% of the MOB, executes the read-modify-write cycles for any pending modifications from the scanned region, and then discards the scanned region. Many of the modifications in the scanned portion of the MOB will have been overwritten by later modifications. Any such overwritten modifications are ignored by the flusher thread and do not generate any reads or writes as explained in Section 3.1.4.

Scheduling Considerations

This section contains an analysis of the potential scheduling benefits provided by the MOB. The analysis shows that the scheduling benefits will be minor. Furthermore, scheduling reduces the amount of write absorption provided by the MOB and therefore, it is better to use a read-modify-write cycle rather than a disk scheduler for removing modifications from the MOB.

The basic problem with scheduling out of the MOB is that it is hard to build an efficient and portable disk scheduler because modern disk drives are complex and hard to model. Ruemmler and Wilkes point out a number of complications in modeling the behavior of a modern disk drive [49]: seek times are hard to predict because each disk seek ends with a *settle phase* that can range from 1–3 milliseconds. Delays due to short seeks are different for disk reads and disk writes because disk reads can be started optimistically while the disk arm is settling on to the correct track whereas writes cannot be started until the settle phase is complete. The disk controller can periodically lock out all requests while internal tables are re-calibrated to account for changes in temperature, disk arm pivot stickiness and other variable factors. Such re-calibrations can take from 500–800 milliseconds. Track lengths near the edge of a disk platter are typically twice the lengths of tracks near the center of the platter. Therefore data transfer rates are higher near the edge of the disk. For example, on the HP C2240 disk, transfer rates can vary from 3.1 megabytes/second to 5.3 megabytes/second [29]. Disks include spare sectors to allow defective physical sectors to be dropped from the disk address space. Disk reads may have to be retried several times to cope with transient errors encountered while reading a disk block. A smart scheduling policy will have to take all of these factors into account and therefore will be complex and not portable.

An alternative to building such a smart scheduler is to use a simple-minded disk scheduling policy that just sorts disk operations by the starting address. However, a simple analysis will show that this simple disk scheduler will not reduce seek and rotational latencies significantly and will have other performance drawbacks. Let us use the disk parameters of the SEAGATE drive used in our simulation study (see Section 4.2.3). Some of the important characteristics of

Disk Type	Seagate ST18771FC (Barracuda 8)
Rotational Speed	7200 RPM (≈ 8.3 milliseconds)
Track Size	89 kilobytes
Seek Times	
Average	8.75 milliseconds
Minimum	0.85 milliseconds
Maximum	19.5 milliseconds

Table 3.1: Disk characteristics

this disk are listed in Table 3.1. We assume that a simple disk scheduler can schedule a large number of disk operations so that each disk operation experiences minimal seek latency and half a disk revolution as its average rotation latency. Therefore the time required to transfer a b byte page will be

$$bt + s_{min} + rev/2$$

where t is the transfer time per byte, s_{min} is the minimum seek latency and rev is the disk's revolution time. Plugging the disk parameters into this formula, we find that a good disk scheduler can transfer a 64 kilobyte page in approximately 11 milliseconds.

If on the other hand instead of scheduling a large number of installation reads and disk writes at the same time, we schedule each installation read by itself immediately followed by the corresponding disk write, then the time required per page transfer will change. The actual cost here will depend on whether we lose an entire disk revolution because the delay between the completion of the installation read and the initiation of the following disk write is too large. The cost will also depend on whether the page fits entirely on one disk track or not. If the page fits entirely on a single track, then the read-modify-write cycle will require an average length seek before the read can start, but there will be no seek necessary between the read and the write. If the page spans more than one track, then a minimal disk seek back to the first track will be required between the read and the write. If we assume a uniform distribution of the start of page extents relative to the start of disk tracks, then with a page size of b and track size of x, the probability that the page fits exactly on one track is equal to the probability that the start of the page extent is located within the first $x - b$ bytes of the disk track. *I.e.*, a minimal disk seek will be required with probability b/x, and no disk seek will be required otherwise. This seek time has to be added to the server turn-around time when computing whether or not we lose an entire disk revolution between the read and the write.

With the default page size of 64 kilobyte, a track size of 89 kilobyte, and a rotational speed of 8.3 milliseconds per revolution, we have a leeway of 2.3 milliseconds before we lose a disk revolution. Given that the minimum seek time of the disk is 0.85 milliseconds, and modern CPUs are very fast, it will be relatively easy for the server CPU to initiate the disk write in the available time slot of 2.3 milliseconds. Therefore, the cost of the read-modify-write cycle will consist of:

- an average length seek before the read

- an average rotational latency of half a revolution before the read
- one rotation delay in which the read completes, the server turns around and initiates the disk write, and the disk controller waits while the disk head rotates back to the beginning of the page
- the time required to write the page back to the disk surface

Therefore the time required for the read-modify-write cycle will be

$$bt + s_{avg} + 1.5rev$$

This translates into 27 milliseconds for a page read and a page write. A disk scheduler on the other hand, as described earlier, will require 11 milliseconds per transfer and therefore 22 milliseconds combined for the read and the write.

Therefore the overhead of using a read-modify-cycle as opposed to a disk scheduler is 23% in elapsed time. As mentioned earlier, a disk scheduler will have the drawback that it will tie down a large portion of server memory from which it will schedule a large pool of page reads and writes. The disk schedule will also take a long time to complete because a large number of pages will have to be read and written. Therefore, the disk schedule will have to be generated and initiated long before the MOB is completely full to avoid massive delays to incoming transactions. Each read-modify-write cycle on the other hand can be delayed until the MOB is almost full. Therefore using a read-modify-write policy will effectively increase the usable MOB size.

3.2.2 Write-Optimized Disk Layout

The write-optimized disk layout policy is modeled after the implementation of the log-structured file system [46, 48]. The entire disk is divided up into large fixed-size regions (half a megabyte in the implementation described in this paper). When a large number of modifications have accumulated in the MOB, entire groups of modified objects are moved from the MOB to an empty region with a single disk write as shown in Figure 3-4. This writing process is efficient, but it does not preserve the on-disk clustering of objects. For example, if the four objects *a*, *b*, *c*, and *d* are in the same page, and *b* and *d* were modified at about the same time, the contents of the page are allowed to fragment on the disk as shown in Figure 3-5.

In determining what to place in the next region, the flusher thread preserves clustering by gathering together all modifications related to a page into a *page fragment*. Each fragment is written contiguously within the region, together with its updated page header. The entire header is written even if the fragment is incomplete in order to make fetching perform reasonably as discussed below. The header may be located in the server cache; otherwise it will need to be read from disk before the region can be written. The flusher thread orders these installation reads to minimize seek distances, reads in the needed headers, and then writes the region to disk. (The format of a region is illustrated in Figure 3-6.) Finally, the page map is updated so that for each page with a fragment in the region, the map points to the location of the corresponding header within the region. Special techniques for maintaining a stable copy of

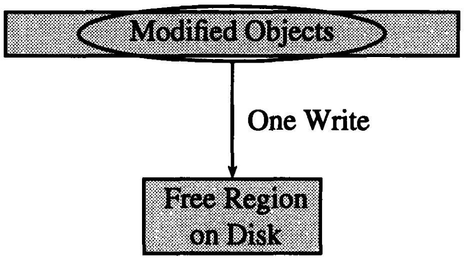


Figure 3-4: Writing modifications to the disk under the write-optimized disk layout policy

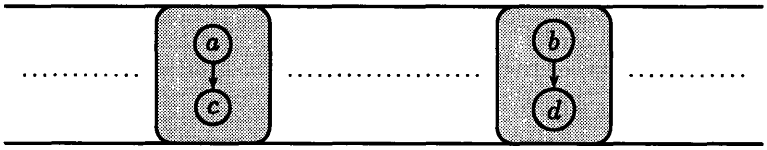


Figure 3-5: Fragmentation under a write-optimized disk layout

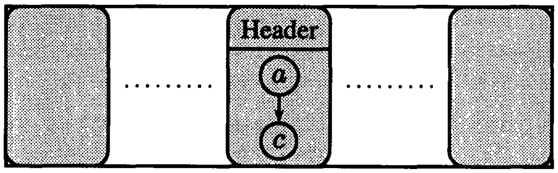


Figure 3-6: Disk region full of page fragments. Each fragment stores some objects and has an associated page header that was written out at the same time as the fragment.

the page map are essential because the page map is updated very frequently. A solution similar to the one used for the inode map in LFS [48] can be used by check-pointing the page map periodically. At recovery time the server can read the last check-pointed state of the page map and combine this check-pointed state with the object location information derived by scanning the regions updated since the checkpoint.

Fetching occurs as follows. If the page header is not in the server cache, it is located using the page map, and read into the cache along with its associated fragment. Then the object is looked up in the header; if it is not in memory, *i.e.*, if its newest version is not in the fragment just read, the header will indicate the location of the fragment holding the newest version, and that fragment will be read. Thus at most two disk reads are needed to fetch the object.

Prefetching occurs as before, except that it is less effective than in the read-optimized policy: related objects are less likely to be in the cache because some of the fragments that belong to the same page as the fetched object may not be present in the server cache. Thus future accesses of the client are more likely to miss in the client cache, and the fetches generated by those misses are more likely to cause an additional disk read than in the read-optimized policy.

Since a page header is more likely to be accessed than a fragment from the corresponding page, the server cache provides separate storage for page fragments and headers. Therefore individual fragments can be discarded from the server cache without having to discard the corresponding page header. Furthermore, the cache replacement policy favors page headers: headers are evicted from the cache only to make room for other headers, whereas fragments can be evicted to make room for either fragments or headers.

The MOB improves performance for the write-optimized scheme for a number reasons: the installation reads for the page headers can be scheduled efficiently, multiple modifications to the same object can be combined into one disk write of the newest version of the object, and modifications to objects in the same page can be grouped into the same fragment. This grouping can recover some of the benefits of clustering that are lost by fragmenting pages.

The write-optimized scheme writes more efficiently than the in-place scheme because it writes entire regions at once and because it only writes the modified objects and not the remainder of their pages. However, page headers are written out in their entirety whenever a new fragment for a page is written out the disk. Therefore page headers take up space in each region and also generate installation reads because the old value of the page header has to be present in the cache for the new value to be constructed. An organization that fragments page headers to avoid header installation reads would perform very poorly because locating a single object would require multiple disk reads to find the page fragment that contains the required object. Another reason to not allow page headers to fragment is that whenever an object is moved (because it has been modified and is being written out to an empty region), the old location of the object has to be available so that the server can keep track of the free space available per region.

Installation reads for headers are not often necessary. As described earlier, our cache replacement policy favors headers and therefore the server cache hit rates for headers are high.

Whenever some modified objects are written to an empty region of the disk, the old on-disk versions of these objects become obsolete. Therefore as the system runs, the amount of live data in each region decreases. In certain cases, the entire contents of a disk region may become obsolete, and the region can be re-used for new modifications. However in general only some fraction of each region will become empty, and the system will have to explicitly compact the data present on several regions to create new empty regions. This compaction of data is called *disk cleaning* and various algorithms for performing such cleaning efficiently have been proposed in the literature [48, 7]. We assume that the disk cleaner runs as a background thread. The thread is activated when the number of free regions on disk falls below a certain threshold. It collects live data from a set of non-empty regions, combines the live data, and writes it to free regions. The regions picked by the cleaner almost always have many holes that correspond to obsolete versions of objects. The cleaner eliminates these holes by rewriting just the live objects and therefore creates free disk regions. The cleaner can potentially also improve the on-disk clustering of related objects because when the cleaner finds multiple fragments that belong to the same page, it merges these fragments together into a larger fragment.

The cleaner can have a significant impact on the performance of the system depending on how often it runs. The frequency of cleaning depends on the amount of free space on disk. If the database size is small compared to the disk size, the cleaner will run less often and therefore cause a smaller performance degradation. If the amount of extra disk space is small, the cleaner will run often and slow down the system significantly. In all of the experiments described in this paper the system was configured so that cleaning cost was negligible. Factoring in the cleaning cost would degrade the performance of the write-optimized layout policy as compared to the read-optimized policy.

3.2.3 Hybrid Disk Layout

The hybrid disk layout policy combines the clustering characteristics of the read-optimized policy with the writing characteristics of the write-optimized policy. The contents of a page, including the header are stored contiguously on disk, and entire pages are read into the cache in response to fetch requests. However each page is allowed to move on the disk whenever it is written out: the disk space is divided up into fixed size disk regions and whenever some pages have to be written out, they are written back with a single disk operation to a previously empty region.

This scheme exhibits read performance similar to that of the read-optimized scheme, but has better write performance because only one disk seek and rotational delay is required to write out a disk region full of pages. However, the gain in write performance is not very significant: none of the installation reads are avoided, and the total saving gained by avoiding some of the disk seeks and rotational delays is a small fraction of the total write cost for reasonably sized pages. (See Section 4.3.13 for an analysis of the differences in write performance between the read-optimized and the hybrid schemes.) The drawback of the hybrid scheme is that it uses up disk regions very fast because entire pages are written out instead of just the modified objects. Therefore the disk cleaner runs frequently. (Even a small cleaning cost is a killer for this scheme because its performance advantages over the read-optimized are quite small to begin with.) Furthermore, since the pages keep moving around on the disk, the page map has to be updated frequently and implementing an efficient mechanism for making these updates persistent adds complexity to the server.

3.2.4 Discussion

The three disk layout policies all have their advantages and disadvantages. The read-optimized policy and the hybrid policy both preserve the clustering of objects in pages and can potentially provide very good read performance for small objects. The write-optimized policy allows pages to fragment over the disk and destroys the clustering of small objects. Therefore its read performance will be poor under many workloads. The write-optimized policy will provide good read performance for small objects only when an application's read access pattern exactly matches its write access pattern, because under this kind of workload the write pattern will automatically preserve the clustering required by the read pattern. Such workloads are unlikely in object database systems, but occur often in file systems because files are typically read and written sequentially in their entirety. The good performance of the write-optimized log-structured file system reported in [48] arises in part from this predominance of sequential reads and writes in typical file system workloads.

The write-optimized policy handles all object modifications very efficiently because it transfers large amounts of data to the disk with one disk operation; it writes out only the portions of the pages that have been modified; no installation reads (except some page header reads) are necessary before modifications to a page are written out. Both the hybrid and the read-optimized policy have much poorer write performance. In a main-memory database where the performance of reading data from the disk is not an issue, a write-optimized disk layout policy may very well be the best option.

Object location is an important problem for a storage system that holds many small objects. Both the read-optimized and the hybrid layout policies can use a simple two-level naming scheme that allows any object to be fetched into the server memory with at most one disk read. Under the write-optimized policy described here, two reads may be necessary: one to fetch the location map for the object's page into memory, and another to fetch the appropriate page fragment into memory. This problem may be alleviated, but at the cost of extra memory, by storing the entire object-location map in server memory. The object-location map changes frequently under both the hybrid and the write-optimized policies because neither of these policies write objects back in place. Efficient handling of these changes to the location map requires periodic check-pointing, as well as some amount of scanning at recovery time to reconstruct the contents of the map. This check-pointing and scanning add extra complexity to the implementation, and also increase the system recovery time.

The hybrid and the write-optimized policies have certain advantages because they do not write modified objects or pages back in place. Changes to object sizes and page sizes can be handled easily because no special code in the implementation is required to find a large enough empty slot on the disk. Instead, the disk cleaner generates big empty regions that will hold pages (and page fragments in the case of the write-optimized policy), as the pages are resized. A server with a read-optimized disk layout policy requires special code that allows the location of a page to change when necessary to allow the page to grow or shrink in size.

The hybrid and write-optimized policies also provide better reliability in some situations because modifications are not written back in place. If a server with a read-optimized policy crashes in the middle of a disk write, the contents of the page being written out may be in an inconsistent state. Preventing such inconsistencies may require the logging of extra information to stable storage whenever the contents of a page are rearranged.

The simulations and experiments described in this thesis compare these disk layout policies on the basis of overall system throughput. System implementors must also consider these other factors when deciding on a disk layout policy. For example, the hybrid policy does not provide any significant performance benefit over a read-optimized policy, and it requires a potentially expensive disk cleaner to run in the background. However, it may still be preferred because of its higher reliability guarantees and the ability to easily handle changes in page size.

Chapter 4

Disk Layout Policies

This chapter describes the simulation study used to evaluate the effectiveness of the MOB, and to compare the relative performances of a read-optimized and write-optimized disk layout policies in the MOB architecture. The simulator allows us to measure the impact of the modified object buffer under a wide variety of synthetic workloads, and to experiment with a number of different disk layout policies. The simulator results indicate that the on-disk clustering of related objects provided by a read-optimized disk layout is important for getting good read performance from an object-oriented database. Furthermore, a large MOB allows a read-optimized disk layout policy to achieve good write performance. The write-optimized disk layout policy does not perform as well under most workloads because it does not preserve the on-disk clustering of related objects.

The results presented in this chapter are for a disk-bound system in which disk bandwidth is the limiting factor to overall system performance. In a system that is not disk bound, the performance benefits provided by both the MOB and the read-optimized policy will become even more significant than predicted by these simulations. First, the MOB generates background disk activity and therefore in a system that is not disk bound, this activity will happen for "free" in the background. Second, the relative performance difference between the read-optimized and the write-optimized policy hinges on the write performance of the read-optimized policy and the read-performance of the write-optimized policy. Writes happen in the background and will be free when the system is not disk bound. Reads on the other hand happen in the foreground: applications have to wait for the results of a fetch request before they are allowed to continue. Therefore the relative performance of the read-optimized policy will improve as the disk load decreases.

Even though there is no formal validation of the simulator, the results of our simulation study are very dependable for a number of reasons. The server component of the simulator models the operation of a database server precisely. In fact, large portions of the simulated server were re-used in the implementation of the Thor server. The disk drive model is taken from the Berkeley Raid Simulator [35] and accurately models disk contention, seek times, rotational delays, and transfer delays. Finally, Chapter 6 backs up the simulation results with experiments on a real implementation running the widely accepted OO7 benchmark [9, 8].

Parameter	Default Value	Range Studied
Object size	128 bytes	128 bytes to 2 kilobytes
Database size	100 megabytes	100 to 900 megabytes

Table 4.1: Database parameters

Parameter	Default Value	Range Studied
Number of Clients	12	1 to 64
Objects fetched per transaction	128	128 or 512
Write Probability	20%	0 to 100

Table 4.2: Client parameters

Section 4.1 contains a description of the database and client workload used to drive the simulations. Section 4.2 contains a description of the simulator and all of the parameters of the simulator. Section 4.3 contains the results of the simulation study.

4.1 Workload

The database consists of a fixed number of equal-sized objects. Objects are partitioned into a number of equal-sized pages. Objects do not move from one page to another and pages do not grow or shrink. Under the default configuration objects are 128 bytes each and the database occupies 100 megabytes. The study looked at a range of object sizes and database sizes as illustrated in Table 4.1. The impact of varying these parameters is described in Sections 4.3.11 and 4.3.12.

A 100 MB database may appear small, but the way to understand this parameter is to consider a system with a much larger amount of data and a very large number of clients. During any given interval of time, only some of these clients will be active. Furthermore, most clients, at least over a short period of time, will limit their accesses to only a portion of the entire database. Hence even if the database is very large, only small parts of it will be in use at any given time. Therefore a simulation of a 100 MB database with a small number of clients can serve as an accurate predictor of the performance of a much larger database that is being used by many clients.

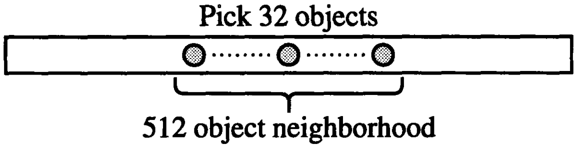
The simulated system consists of some clients connected to a server. Each client executes a sequence of transactions. A transaction reads some number of objects. Certain percentage of the objects read by a transaction is modified and at transaction commit the new versions of these objects are sent to the server. Default values for the number of clients, transaction length, and write probability are listed in Table 4.2 along with the range over which these parameters were varied in the simulation study. The number of objects fetched by each transaction was not varied in most experiments because changing this parameter has an identical effect on the performance of all studied configurations: it just scales the throughput as measured in transactions per second. (In a few of the experiments the transaction length was increased to 512 to allow a large number of related objects to be accessed together within the same transaction, but in the rest of the experiments the transaction length was held fixed at 128 accesses.) Part of the reason for the insensitivity of the simulation results to the transaction length is that the simulator does not model concurrency control. Increasing the transaction length in a real system will increase contention between clients and therefore reduce the overall system throughput.

The write probability of the client workload has a significant impact on the performance of the various disk management policies being studied. Section 4.3.6 describes the impact of changing the write probability on the performance of the policies under study. Even though the write probability is varied from 0 to 100 percent in this simulation study, typically write probabilities are fairly small and do not exceed 15 to 20 percent[11].

4.1.1 Clustering

The simulator models a server that provides clustering of related objects. The database is modeled as an ordered list; objects close to each other on the ordered list are related to each other. The server partitions this list of objects into equal sized pages by allocating the first n objects to the first page, the next n objects to the second page, and so on. The default page size for our experiments was 64 kilobytes, (*i.e.*, 512 objects.)

The client access pattern is designed to simulate pointer traversal among related objects. Different clients can have different access patterns and these patterns can vary over time. Therefore the static partitioning of objects into pages does not capture any particular client's access pattern exactly. Each client's access pattern is instead simulated by having it first pick a random object somewhere in the ordered list of objects and then pick other objects "nearby" in the object list.



In the default configuration of the simulator an object is considered "near" another object if they are separated by at most 256 positions in the ordered list. After the client has picked an initial object it picks 31 more objects that are near the initial object (*i.e.*, within 256 positions of the object on either side). Another way to state this is that the client randomly picks 31 more objects to visit in the 512 object subsequence that is centered around the initial object. This group of 32 object accesses is called to a *clustered access*.

We can formalize this clustering behavior with two parameters: *cluster region size* and *cluster length*. Cluster region size is the size of the neighborhood of objects from which related objects are picked by the client workload. Cluster length is the number of objects picked from each region.

Parameter	Default Value	Range Studied
Cluster region size	512 objects	32 to 2048
Cluster length	32 objects	1 to 512

Table 4.3: Clustering parameters for client workload

Larger values of cluster region size indicate that the clustering process was not able to find small groups of highly related objects. For example, a large object graph may have a small subset of objects that are always accessed together. A smart clustering process may detect this group of related objects and store them tightly in a small cluster. A simple clustering process, for example one based on a depth-first traversal of the object graph, would instead cluster the whole object graph into one large cluster. This large cluster may be spread out over several pages and therefore each clustered access will require touching a large number of pages.

The cluster length parameter captures the synergy between the clustering process and the application access patterns. A larger cluster length parameter indicates that more objects that are considered related by the application were placed together by the clustering process. The impact of a larger cluster length parameter is that each clustered access will touch more objects and therefore each transaction will require fewer clustered accesses.

Not all objects used by a clustered access require explicit fetch requests to the server. Some of these objects may already be in the client cache. Also, whenever the server receives a fetch request for a particular object x, it also sends back some other objects that it expects the client to read soon. This speculative shipping of objects from the server to the client reduces the number of network round-trips required to fetch a set of objects into the client cache. Day describes and compares a number of policies that can be used to control this prefetching of objects into the client cache [13]. Prefetching policies affect the number of network round trips and network utilization. Since the simulations described in this chapter are for a disk-bound system, the choice of prefetching policy will not have a significant impact on the results of these experiments. Therefore the simulated server just uses the simple-minded policy of shipping all objects that are present in server memory and belong to the same page as the requested object x. This policy is very similar to the page-based prefetching policies employed in many systems, but has the advantage that it works robustly in the presence of partial page modifications such as the contents of the MOB, or cached page fragments in the case of the write-optimized disk layout policy.

The default values used in these simulations are simulating a system that has sub-optimal clustering of related objects. A really well clustered system would have all 32 related objects located next to each other on the disk instead of being spread out over a 512 object neighborhood. However perfect clustering is very hard, if not impossible, to achieve in practice. In a study of several object-oriented database applications for VLSI/CAD design, Chang and Katz found that access patterns for the same data varied widely depending on the application used to access the data [11]. Therefore any single clustering policy will not provide perfect clustering for all of these applications. In a study of clustering algorithms Tsangaris and Naughton found that most simple and practical clustering algorithms provide poor clustering [55]. Algorithms

Parameter	Default Value	Description
Hot area size	6 megabytes	About 50,000 objects. This is a small fraction of the database accessed heavily by a client. Half of this area is shared with other clients; the other half is private to this client.
Access to hot area	90%	Percentage of accesses directed to the hot area.
Cache Size	6 megabytes	This is made the same as the hot area size under the assumption that the client can effectively cache the hot part of its working set.

Table 4.4: Skewed access parameters

that provide good clustering were prohibitively expensive. Therefore, the default configuration of 32 objects being picked from a 512 object neighborhood is reasonable. Furthermore this chapter describes experiments that vary these parameters. The range of clustering parameters studied in this set of experiments is listed in Table 4.3.

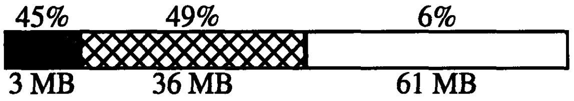
With the default page size of 64 kilobytes (512 objects), each clustered access will in most cases touch exactly two pages. In rare cases the initial object may be near the middle of a page and in that case only that page will be touched by the clustered access. Therefore each clustered access that does not hit in any cache will on average require two page reads unless one of the accessed pages is evicted from the server cache before the clustered access finishes. (Such evictions will be rare because the server cache uses a least-recently-used replacement policy.) With smaller pages the number of reads per clustered access will increase. With larger pages the number of reads will decrease because of our clustering assumptions but the cost of each read will be higher because of larger page transfer times.

The default values for the clustering parameters given in Table 4.3 are only some of the values studied in the simulation experiments. Section 4.3.7 describes the effect on performance of varying these simulation parameters. The resulting system performance consistently shows that maintaining on-disk clustering helps overall performance, and that a large MOB increases system throughput significantly.

4.1.2 Skewed Access Pattern

The pattern we have described so far tells us how a clustered access touches related objects over a short period of time. Over a longer period of time the client repeats such clustered accesses over different regions of the database. Each client directs 90% of its clustered accesses to a small 6 megabyte region of the database. Half of this small *hot* region is shared with other clients; the other half is hot only for this client. The "private" hot region for client i is not really private: cold accesses from other clients can still be directed to this region.

With the default configuration of 12 clients and a 100 megabyte database the database splits up into three distinct areas.



The 3 megabyte region shared by all of the clients receives 45% of the total clustered accesses. Each client's cold accesses are distributed uniformly over the 94% section of the database that does not contain either its private or shared hot region. Therefore the 36 megabyte region that constitutes the combined private regions of the 12 clients gets half of the hot accesses (*i.e*, 45%) as well as (36/94) of the cold accesses. Hence the 36 megabyte warm region receives approximately 49% of the accesses. The remaining 61 megabyte *cold* region gets the remaining 6% of the accesses. This partitioning of the database into hot, warm, and cold areas is a direct result of dividing each client's hot region into a private and a shared area. This division of each client's hot region models clients that do not have identical working sets: only some portion of the working sets of different clients overlaps. Section 4.3.8 describes the impact of varying these parameters that control locality of access.

4.2 Simulator Components

The simulator models an object-oriented database running in a multiple client, single server system connected by a network. The simulator does not model network and processor utilization. The disk itself, and the components of the system that interact with the disk, are modeled very accurately. Therefore, the simulation results apply directly only to a disk-bound system. Techniques that reduce network and processor utilization will also be important in any implementation of an object-oriented database, but are not covered in this simulation study.

4.2.1 Clients

Each client has a fixed size cache. The simulator does not model the contents of the cache directly. Instead, the cache is modeled probabilistically based on the client cache size and the sizes of the hot and cold database regions using the formulas

$$\begin{aligned} \text{hot hit ratio} &= \max(1, \mathit{cache}/\mathit{hot}) \\ \text{cold hit ratio} &= \min(0, (\mathit{cache} - \mathit{hot})/\mathit{cold}) \end{aligned}$$

where *cache* is the client cache size, *hot* is the combined size of the shared hot region and the private hot region for this client, and *cold* is the database size - *hot*. Under the default configuration where the client cache size is exactly equal to the size of the hot region, these formulas produce a hot hit ratio of 1, and a cold hit ratio of 0. *I.e.*, each hot clustered access does not generate any fetch requests, and each cold clustered access generates one fetch request for each page touched by the clustered access. This configuration simulates a system where each client has a cache big enough to hold that client's working set.

Parameter	Default	Description
Page Size	64 kilobytes	*i.e.*, 512 objects. We studied page sizes in range of 1 to 128 kilobytes and found that 64 kilobyte segments tend to work well on most variations of the access patterns generated by this workload.
Disk Region Size	512 kilobytes	This is the amount of data that is written with one write operation by the write-optimized disk layout policy (see Section 3.2.2). This large value allows disk writes proceed at almost the maximum disk transfer rate.

Table 4.5: Disk interaction parameters

4.2.2 Network

The network module used in the simulator assigns each network message a fixed latency of one millisecond. Network congestion is not modeled. Therefore our results do not apply to a network-bound system, only to a disk-bound system. In a network-bound system the relative performance differences between small and large modified object buffers, and between the various disk layout policies examined in this chapter, will be smaller than the performance differences indicated by our simulation results. However, network bandwidth is typically not an issue for a well-engineered object-oriented database system. Techniques such as smart prefetching of objects into the client cache [13], and the shipping of individual modifications as opposed to entire modified pages [56], can be used to reduce the amount of network traffic generated by database activity. For example, Gruber studied concurrency control policies in a large client–server object oriented database. His results show that with an eighty megabits per second network, the network is not a bottleneck under most workloads [2, 27]; either the disk, or the server CPU, saturates first.

4.2.3 Disk

The server has a disk that provides persistent storage for objects. We use the disk simulation algorithm from the Berkeley Raid Simulator [35] to model a Seagate ST18771FC (Barracuda 8) disk drive[1]. This simulator accurately models disk contention, seek times, rotational delays and transfer delays. However, the disk model does not include a read-ahead cache. Read-ahead caches typically help performance by allowing an efficient transfer of a long sequence of data from the disk to the server even if the server breaks up the read request into a number of shorter requests. The disk model used in the simulator compensates for this shortcoming by allowing long sequential read requests to be handed off directly to the disk controller. Therefore, the lack of read-ahead caches in the simulated disk model does not affect the results of our study. Another limitation of the disk model is that it only allows a single track size to be used over the entire surface of each disk platter. The real Seagate Barracuda drive has a track size that

[1] Seagate and Barracuda are trademarks of Seagate Technology, Inc.

Disk Type	Seagate ST18771FC (Barracuda 8)
Rotational Speed	7200 RPM
Track Size	Ranges from 69 . . . 109 kilobytes
Cylinders	5333
Tracks per cylinder	20
Capacity	≈ 9 gigabytes
Seek Times	
Average	8.75 milliseconds
Minimum	0.85 milliseconds
Maximum	19.5 milliseconds

Table 4.6: Disk parameters

Parameter	Default	Description
Cache size	12 megabytes	About 100,000 objects. Also see Section 4.3
MOB size	12 megabytes	About 100,000 objects. Also see Section 4.3

Table 4.7: Server configuration parameters

varies from 69 kilobytes to 109 kilobytes over the disk surface. The simulator uses a fixed track size of 89 kilobytes instead. This approximation forces the simulated disk to have the same maximum transfer rate regardless of the distance of the disk head from the center of the disk. A real Seagate drive would exhibit a smaller transfer rate near the center of the disk, and a larger transfer rate near the periphery. This difference does not affect the simulation results because none of the studied disk policies pay any attention to the disk geometry apart from trying to keep related objects close to each other, and transferring large amounts of data per disk operation.

4.2.4 Server

The simulator models a replicated server with a primary and a single backup. Stable storage for transaction commits is implemented by replicating modified objects in the memory of both the primary and the backup. It suffices to replicate the contents of the MOB, because the MOB itself is cleaned in log order and the log management techniques used in the Harp file system can be used to coordinate the removal of objects from the memories of the two replicas [39]. Only the primary is simulated in detail. The backup is simulated by introducing delay at transaction commit that corresponds to the time required to send the modified objects to the backup and to receive an acknowledgment.

The primary caches related groups of objects in main memory. Under the read-optimized policy the cache stores entire pages. Under the write-optimized policy the cache stores page fragments and headers. An LRU replacement policy is used for this cache. The primary also maintains a modified object buffer. Committing transactions have to wait for space to be created in the MOB before they are allowed to commit. When the MOB is almost full, a flusher thread is started to install pending modifications on disk. The exact parameters that control the triggering and operation of the flusher thread are described in detail in Chapter 3. As modifications are installed on disk, the corresponding objects are deleted from the MOB and any transactions waiting for space in the MOB are allowed to continue.

The 64 kilobyte page size used in our study was chosen to make pages an efficient unit of transfer between disk and memory. In our studies we examined a large range of page sizes and the results indicated that given the performance characteristics of the disk used in our simulator, any page size in the 64 to 128 kilobytes range works fairly well for most reasonable combinations of the clustering parameters listed in Table 4.3. See Section 4.3.10 for more details.

4.2.5 Disk Cleaning

The write-optimized disk layout policy eventually runs out of free disk regions and has to create new free disk regions by compacting the live data on some non-free disk regions. Research on file systems has shown that the cleaning cost can vary widely depending on workload [7, 48, 51, 52]. We side-step the issue of cleaning costs in our simulation study by ignoring all cleaning costs. A "magic" zero-cost disk cleaner runs whenever the number of free regions falls below a certain threshold. The magic disk cleaner performs the normal cleaning action of compacting live data from some disk regions into a small number number of disk regions, but runs in zero simulated time. We run the system with very little extra space (the database occupies 80% of the available disk space.) Therefore the zero-cost disk cleaner runs frequently. This set-up is very favorable for the write-optimized disk layout policy because not only does disk cleaning happen for free, it also happens frequently and reduces the amount of fragmentation. (When the cleaner runs it combines multiple fragments of the same page into a single larger fragment.) Even with this favorable cleaning set-up the write-optimized policy performs poorly when compared to the read-optimized policy.

4.3 Results

We now present simulation results that demonstrate the impact of the modified object buffer and the disk layout policy on the performance of the system. The graphs presented in this section display data averaged from a number of different runs of the simulator. The 95% confidence intervals for these runs are drawn as error bars on all of the graphs, though in many cases these error bars are small enough to be invisible. Unless otherwise stated, each experiment described in this chapter uses the default values of the different system and workload parameters listed in the tables given earlier in this chapter. Any deviations from these parameters are described with each experiment.

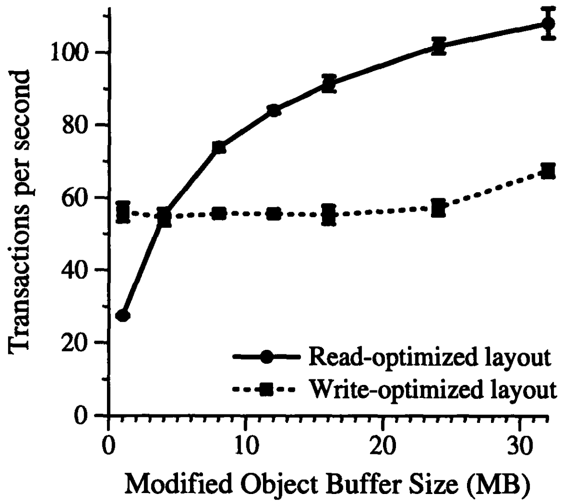


Figure 4-1: Effect of MOB size on system throughput. These experiments were run with a 20% write probability (*i.e.*, 20% of the objects accessed by a transaction were modified.) The server cache size was held fixed at 12 megabytes. Default values for other simulation parameters are listed in earlier tables in this chapter.

4.3.1 MOB Size

Figure 4-1 demonstrates the impact of the modified object buffer on the throughput of the system. The throughput is measured in transactions per second where each transaction accesses 128 objects as described in Section 4.2. With a very small MOB, the read-optimized disk layout has poor performance. However, as the MOB size increases, write performance becomes less and less of a factor because of the buffering characteristics of the MOB. Read performance becomes more important, and with a MOB bigger than four megabytes the read-optimized disk layout policy outperforms the write-optimized policy.

4.3.2 Write Absorption Analysis

The effectiveness of the MOB arises mostly from its ability to combine multiple modifications to objects from the same page into a single page update. (Any possible scheduling benefits will be minor as discussed in Section 3.2.1.)

Figure 4-2 shows the write absorption effects of the MOB by plotting the number of object updates handled per page write as the MOB size is varied under the simulation of the read-optimized disk layout policy. As the MOB size increases, the number of page writes (and installation reads) per modified object drops significantly and therefore the write cost is reduced dramatically. Similar results are obtained for the write-optimized disk layout policy, but in that case the system is read-bound and therefore reducing the write cost does not have a significant impact on the overall throughput.

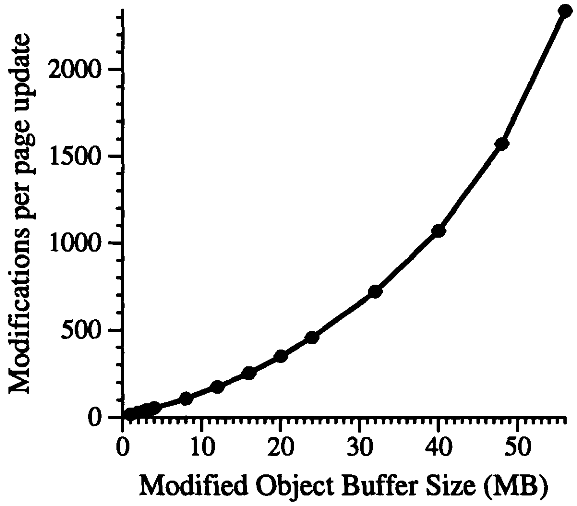


Figure 4-2: Effect of MOB size on the average number of object modifications handled by a page update under the read-optimized disk layout policy. (A page update includes a page write and possibly an installation read.) The Y-axis shows the total number of object modifications divided by the total number of page updates over a long period of time.

This section contains an analysis of the write absorption benefits provided by the MOB. For simplicity, we restrict the analysis to a clustered access pattern that is spread out uniformly over a single region of the database. As shown later, the results of this analysis can be refined to predict write absorption under the hot/warm/cold workload used in the simulation experiments.

Table 4.8 shows the important parameters for this analysis. All size parameters are in units of objects. s is the size of the database region that is the target of the uniform workload. p is the page size and n is the MOB size. Chunk size c is the average number of objects modified on each individual page touched by a transaction. For example, with the default values of the simulator workload, each clustered access touches two pages, and modifies 20% of 32 objects. Therefore the number of objects modified per page (*i.e.*, the chunk size) is 3.2. Table 4.8 also defines two useful derived parameters: μ is the fraction of a page modified by a transaction that touches that page and is equal to c/p. λ is the fraction of the database region that can be stored in the MOB and is equal to n/s. The derived parameters μ and λ are introduced because as will be shown later in this section, the amount of write absorption provided by the MOB can be expressed as a simple function of μ and λ.

The analysis described in this section predicts the amount of write absorption. We start this analysis by computing the number of modified objects that are installed into the persistent database by each disk write. A disk write of page X is triggered when some chunk Y that belongs to page X reaches the beginning of the MOB. Figure 4-3 shows the lifetime of Y in the MOB. Let A be the set of objects that are present in the MOB just before Y is inserted into the MOB. Eventually, as modifications are absorbed and written to disk, Y moves to the head of the MOB. Let B be the set of objects that are present in the MOB at the time Y reaches

Parameter	Description
s	Database region size
n	MOB size
c	Chunk size
p	Page size
μ	c/p
λ	n/s

Table 4.8: Parameters for the analysis of write absorption in the MOB. All sizes are in units of objects. We assume that $n \leq s$, and that $c \leq p$.

the head of the MOB and generates a disk write. We can compute the number of modified objects installed into the persistent database by the disk write of page X by first computing the probability that an object Z that belongs to page X is not one of the installed modifications. Object Z is not one of the installed modifications if and only if the following three conditions are all true:

1. Z is not in chunk Y. The disk write installs all of the objects of Y and therefore if Z were in chunk Y, it would be installed by the disk write.

2. Z is not in B. The disk write installs all objects that are in the MOB and belong to page X. Therefore if Z were in B, the disk write would have installed Z.

3. Z is not in A. There must not have been any writes of page X since chunk Y was inserted into the MOB. If there had been such a write, it would have removed Y from the MOB and therefore Y would not have triggered another disk write of X. Since there were no writes of page X triggered by the contents of B, all modifications to page X from B must have been overwritten by newer modifications present in A. Therefore if Z is in A, it must also be in B. But the previous condition already tells us that Z is not in B. Therefore we can conclude that Z is not in A.

Combining these observations we get that the probability that object Z is not installed by this disk write of page X is

$$\begin{aligned}
&P(\text{object } Z \text{ from page } X \text{ is not installed}) \\
&\quad = P(Z \text{ not in chunk } Y) \times P(Z \text{ not in region } A) \times P(Z \text{ not in region } B) \\
&\quad \approx P(Z \text{ not in chunk } Y) \times (P(Z \text{ not in MOB size region}))^2 \\
&\quad = (1 - c/p)(1 - n/s)^2 \\
&\quad = (1 - \mu)(1 - \lambda)^2 \qquad (4.1)
\end{aligned}$$

There are p objects per page and therefore the expected number of objects installed per page

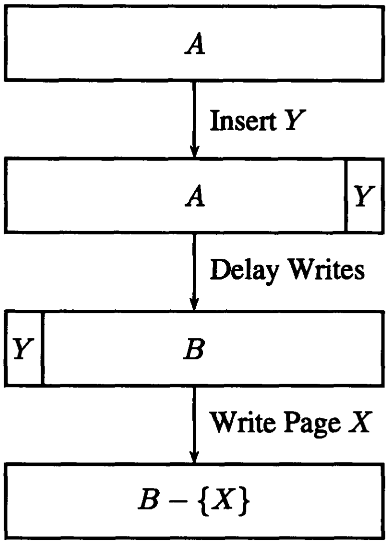


Figure 4-3: Progress of a modified set of objects in the MOB as new modifications are inserted and old modifications are either overwritten or discarded.

write is

$$\begin{aligned} &E(\text{number of modified objects installed per page write}) \\ &\quad = \; p(1 - P(\text{object } Z \text{ from page } X \text{ is not installed})) \\ &\quad = \; p\left(1 - (1-\mu)(1-\lambda)^2\right) \qquad (4.2) \end{aligned}$$

Now we can compute the expected number of writes generated by each chunk of modifications that enters the server. Each chunk has c objects. Because the MOB size is n, and the database region size is s, out of this chunk $cn/s = c\lambda$ objects will just overwrite existing MOB objects. Therefore each chunk will add $c(1 - \lambda)$ objects to the MOB. Equation 4.2 tells us how many modified objects are removed from the MOB with one disk write. Therefore the expected number of page writes generated by each chunk that is inserted into the MOB is

$$\begin{aligned} &E(\text{number of page writes per chunk}) \\ &\quad = \; \frac{c(1-\lambda)}{p\,(1-(1-\mu)(1-\lambda)^2)} \\ &\quad = \; \frac{\mu(1-\lambda)}{1-(1-\mu)(1-\lambda)^2} \qquad (4.3) \end{aligned}$$

This equation leads to a number of interesting observations about the characteristics of the MOB.

- If the size of the MOB is the same size as the database region being accessed (*i.e.*, $\lambda = 1$), the entire region sits in the MOB and the number of page writes drops to zero.

- If the size of the MOB is zero, the number of page writes per chunk is one: each chunk has to be written to disk immediately.

- If entire pages are modified at a time (*i.e.*, $\mu = 1$), the number of page writes per chunk is $1 - \lambda$. The intuition behind this formula is that λ fraction of the incoming chunks will hit a page that is already present in the MOB, and will not generate any writes.

- A small MOB provides significant write absorption. For example, let us consider a system in which the MOB is 10% of the database region size ($\lambda = 0.1$). If 10% of a page is modified at a time ($\mu = 0.1$), the expected number of writes generated per incoming chunk will be approximately 0.33. Without a MOB, each incoming chunk will generate one disk write. Therefore a 10% MOB reduces the number of writes to one third of the number of writes that are required without a MOB.

It is important to note that this write absorption occurs under a uniform workload. If the workload is skewed, the MOB will partition itself into a hot and cold region, and hot writes will be absorbed at a higher rate as demonstrated later in this section. Therefore under a skewed workload the MOB will provide even more write absorption than predicted by Equation 4.3.

Analysis of Simulation Results

The write absorption analysis gives us a way to predict the absorption provided by the MOB for a uniform access pattern. However many workloads will not be completely uniform. We can still use the uniform workload analysis to predict the write absorption under non-uniform workloads by treating the non-uniform workload as a mixture of several uniform workloads.

The simulator workload can be considered a combination of three such uniform access patterns: hot, warm, and cold as described in Section 4.1.2. This section analyzes the write absorption provided by the MOB under this skewed workload and shows that the analysis accurately predicts the behavior of the simulated system.

Figure 4-4 shows the partitioning of the MOB into hot, warm, and cold regions as the MOB size is varied under the skewed workload model of the simulator. The X axis shows the total MOB size. The Y axis shows how the MOB partitions itself into hot, cold, and warm regions of different sizes. As the MOB grows, the portion of the MOB dedicated to the hot region of the database increases until the entire 3 MB hot region is present in the MOB. The portion dedicated to the warm region also increases rapidly. The cold portion of the MOB does not increase as rapidly as the warm portion because very few of the objects that are inserted into the MOB are cold. The sizes of the hot, warm, and cold portions do not add up to the total MOB size because of overhead for MOB management data structures. (The simulator charges 16 bytes of overhead per object present in the MOB, and an extra 16 bytes of overhead per chunk present in the MOB. Changing these overhead parameters just changes the effective size of the MOB. These parameters have no other impact on the simulation results.)

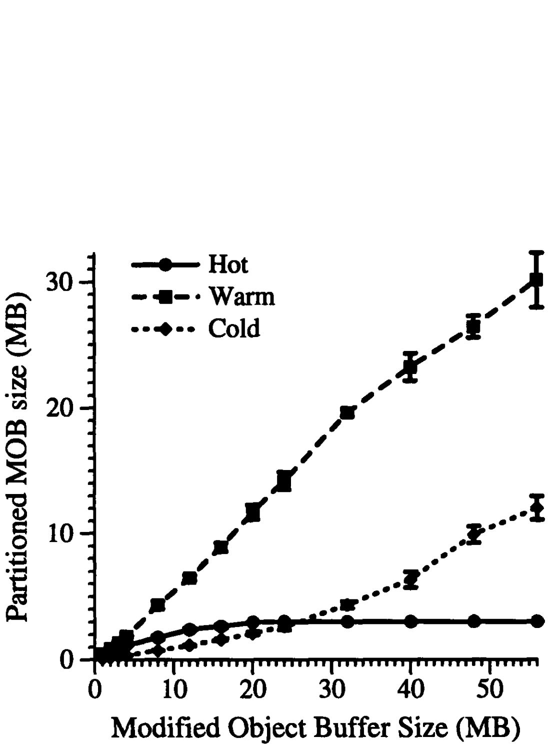


Figure 4-4: Simulator measurements of the partitioning of the MOB into hot, warm, and cold regions as the size of the MOB is varied.

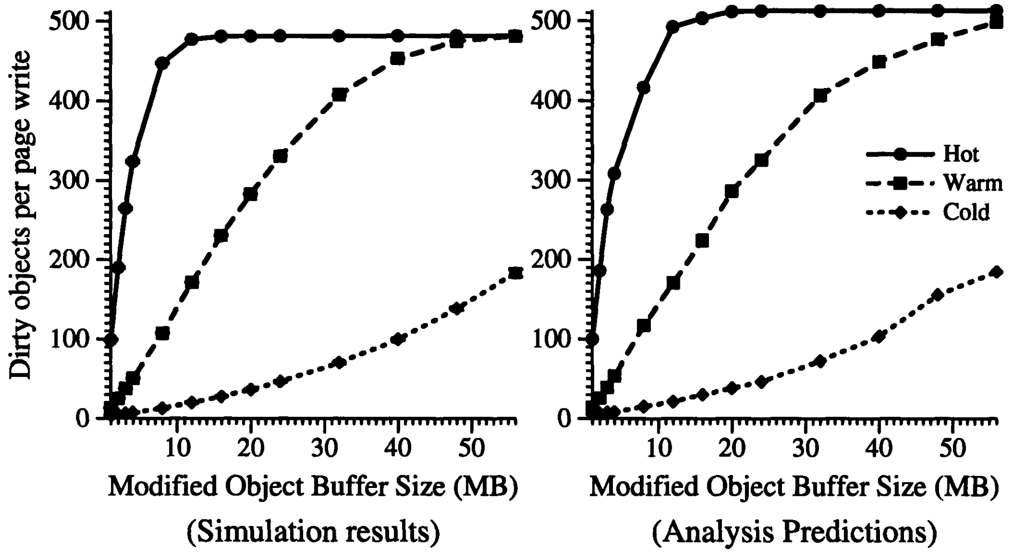


Figure 4-5: Number of dirty objects written out on each page according to the simulator and the analysis of the MOB.

MOB	MOB Division (MB)			Actual dirty			Predicted dirty		
(MB)	Hot	Warm	Cold	Hot	Warm	Cold	Hot	Warm	Cold
1	0.3	0.4	0.1	99	13	4	100	14	5
2	0.6	0.8	0.2	190	25	5	186	26	7
3	0.9	1.3	0.3	264	37	6	263	39	8
4	1.1	1.8	0.3	324	50	7	308	53	8
8	1.7	4.3	0.7	447	107	13	416	117	15
12	2.4	6.5	1.1	477	171	20	492	170	21
16	2.6	8.9	1.6	481	230	28	503	224	30
20	2.9	12.0	2.1	481	283	36	511	286	38
24	3.0	14.2	2.6	481	330	46	512	325	46
32	3.0	19.6	4.3	481	407	70	512	406	72
40	3.0	23.2	6.3	481	453	99	512	448	103
48	3.0	26.4	9.9	481	474	138	512	476	155
56	3.0	30.0	12.0	481	480	183	512	498	184

Table 4.9: Analysis of the write absorption provided by the MOB under the simulator workload. The last two sets of columns show the average number of dirty objects present in the MOB for each page that is written out to disk.

Figure 4-5 shows the average number of dirty objects written out on each hot, warm, and cold page. The left graph shows the measurements from the simulator, and the right graph shows the values predicted by Equation 4.2. The predictions are very accurate except when the number of dirty objects written out per page is large: the prediction is slightly off in such cases because in the simulated server each page has a page header, and therefore each page stores less than p objects. The results presented in Figures 4-4 and 4-5 are also summarized in Table 4.9.

4.3.3 Read Performance

The difference in the read performance of the read-optimized and write-optimized layout policies is illustrated explicitly in Figure 4-6 by showing the throughput of the system when processing read-only transactions. We warm up the system by running it at 20% write probability with a 12 MB MOB before starting the measurements. The warm up period models a database that has been in use for a while (as opposed to a freshly created database.) The choice of MOB size for the warm up period is important because a larger MOB allows more write absorption and therefore creates larger fragments on disk.

This experiment highlights the impact of the clustering provided by a read-optimized disk layout. With a single client, neither system is disk-bound. However since the simulated clients are very active (the clients generate database accesses without performing any computation between accesses), as soon as the number of clients increases to two, the system becomes disk-bound, and the overall throughput does not increase if the number of clients is increased further.

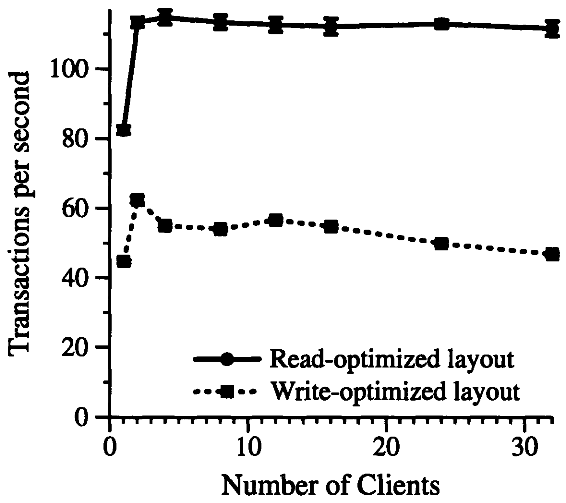


Figure 4-6: Read-performance of the two disk layout policies. These experiments involved no modifications at all. *i.e.*, the write probability was set at 0%. The database was "warmed up" for these experiments by running the system with a 20% write probability with a 12 MB MOB before starting the measurements.

The write-optimized policy does not preserve clustering and therefore has worse performance than the read-optimized policy.

4.3.4 Read Performance Analysis

A simple analysis of the workload and the disk parameters described in Section 4.2 explains the magnitude of the difference in the throughput of these two policies. The client generates clustered accesses that are hot or cold. Hot accesses hit entirely in the client cache. Let us assume that a cold clustered access generates c disk reads at the server on average, and that each of these disk reads transfers b bytes of data. The cost of each disk read is therefore $bt + s + r$ where s is the average seek time of the disk, r is the average rotational latency, and t is the transfer time required per byte based on the peak throughput of the disk. Therefore each cold clustered access requires time $c(bt + s + r)$. Since only 10% of the clustered accesses are cold and each transaction requires 4 clustered accesses, each transaction generates 0.4 cold clustered accesses on average. The server cache has a very low hit ratio because the stream of fetch requests coming in to the server consists entirely of cold clustered accesses. For the default database configuration and workload the server miss ratio is about 88%. (Cold clustered accesses that generate fetch requests for the server are spread out over a 97 megabyte region of the database, and the server cache size is 12 megabytes. This predicts a server cache hit rate of a little above 12%.) Therefore each transaction on average generates $0.4 \times 0.88 = 0.35$ clustered accesses that miss in both the client and the server cache. Therefore the throughput of the system in transactions per second should be

$$\frac{1}{0.35c(bt + s + r)}$$

For the disk and the database in question, the transfer time per byte is obtained simply by dividing the revolution time of the disk by the track size (see Table 4.6):

$$t = 9.14e^{-8}\text{seconds per byte}$$

Assuming that on average each read operation encounters half the revolution time as its rotational latency we get:

$$r = 4.17e^{-3}\text{seconds}$$

The average seek time for the operations in this experiment is *not* the same as the disk's average seek time because the database occupies just a small fraction of the disk. (The disk capacity is 9 gigabytes whereas the database size is 100 megabytes.) Given the size of the database and the disk geometry parameters, a simple calculation shows that the database occupies 57 cylinders. Plugging in this number into the seek time calculator of the disk model gives us an average seek time of

$$s = 1.37e^{-3}\text{seconds}$$

As discussed in the Section 4.2, each clustered access touches two pages. Therefore under the read-optimized disk layout policy there are two page reads of 64 kilobytes each per cold clustered access. Plugging these numbers and the disk parameters into the throughput formula given above we get $1/(0.35 \times 2(64 \times 1024t + s + r)) = 125$ transactions per second. The simulator results in Figure 4-6 show a peak throughput of almost 115 transactions/second. This is very close to the performance predicted by our rough analytical model. The error in the predicted performance arises from our assumption about the average rotational latency of half a revolution. Low-level disk measurements from the simulator indicate that the average rotational latency of disk operations in the simulation is a little higher than half a revolution and therefore the simulated server's throughput is somewhat lower than the throughput predicted by the analytical model.

Performance analysis of the write-optimized layout is slightly more complicated because it has to take into account the fragmentation of each page on the disk. The database is assumed to have been used under a 20% write workload before this read-only experiment is run. Let us assume that the average size of fragments created by this workload is f. Also let the disk space utilization be u. (The write-optimized disk layout policy needs some extra disk space over and above the space required by the database so that the disk cleaner does not thrash when trying to generate free disk regions. u is just the database size divided by the available disk space.) On average each fragment will have uf live data and $(1 - u)f$ obsolete data. With a page size of C, on average $C/(uf)$ disk reads will be required to read the contents of the page. We need this many disk reads because even though the clustered access does not touch the entire page, it touches random objects spread out over possibly the entire page. As discussed earlier, each cold clustered access touches two pages and therefore requires $(2C)/(fu)$ fragment reads under the write-optimized policy. From the earlier analysis of the read-optimized policy we obtained the formula $1/(0.35c(bt+s+r))$ for the throughput of a system that generates c disk reads of size b per cold clustered access. Under the write-optimized policy, the measured value of f is 24 kilobytes, and the disk space utilization was set to 80%. Therefore with the default page size of 64 kilobytes, c has the value $(2C)/(fu) \approx 6.9$. $b = f$ because the write-optimized policy reads individual fragments from the disk. t and r are unchanged from the analysis of the read-optimized policy. s is slightly different because the write-optimized layout uses more disk space than the read-optimized layout:

$$s = 1.45e^{-3}\text{seconds}$$

Plugging all of these values into the throughput formula gives us a predicted throughput of 54 transactions per second. The observed peak throughput of the simulated write-optimized policy is 57 transactions per second for twelve clients, which is a close match to the throughput predicted by our analysis.

4.3.5 Memory Allocation

In the experiments described so far, both the read-optimized and the write-optimized disk layout policies were assigned the same amount of cache memory and the same amount of MOB memory. However, given the characteristics of the two layout policies, it makes more sense for the write-optimized disk layout policy to have a bigger cache, and for the read-optimized disk layout policy to have a bigger MOB. This tradeoff is shown in Figure 4-7. We take a fixed amount of server memory (24 megabytes in these experiments), and vary the percentage of this memory that is assigned to the MOB. The rest of the memory is assigned to the server cache. (It may seem wasteful of resources to require 24 megabytes of server memory for a 100 MB database, but as described in the analysis in Section 4.1, 100 MB is just the part of the database that is being accessed by the currently active clients; the total size of the database may be much larger than 100 MB.)

As expected, the highest performance for the read-optimized layout policy is achieved when it uses a very large fraction of the server memory (about 80%) for the MOB and the remaining 20% for the cache. In contrast, the write-optimized layout policy achieves the best performance when it uses most of the server memory as cache space. An interesting point illustrated by this graph is that increasing the cache space does not help either of these schemes significantly because the server cache is a second level cache: requests coming into the server cache have already been filtered through a client cache and therefore do not exhibit much locality. (Muntz and Honeyman point out a similar phenomenon in the context of distributed file systems [44].) The server cache only needs to be big enough to hold a page or a page fragment in memory long enough for the client to fetch all necessary objects into the client cache.

This experiment determines the optimal division of memory between the cache and the MOB for a 20% write probability. Experiments with other write probabilities indicate that the division for this write probability is also very close to optimal for other write probabilities. Therefore, we chose this division of memory regardless of the write probability. The only workload where this division of memory is significantly sub-optimal is a read-only workload.

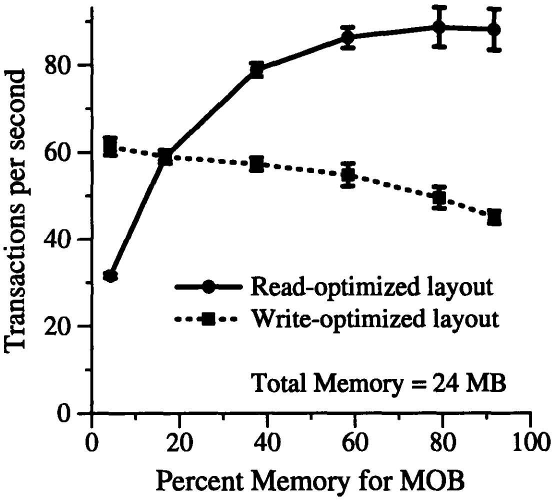


Figure 4-7: Impact of partitioning a fixed amount of server memory between the server cache and the MOB. These experiments were run with a 20% write probability. The graph stops when MOB reaches 90% of the server memory. Beyond that point the server cache locks up because it is not big enough to support the operation of the flusher thread and the object fetching code.

Obviously a minimal MOB size and larger caches would help both policies in that case, but again not by much because the server cache is a second-level cache.

4.3.6 Write Probabilities

Given these optimal assignments of server memory to the cache and the MOB, we ran experiments that compared the performance of the two layout policies on various workloads (see Figure 4-8). The database is created well-clustered under both disk layouts. Therefore when the workload has no writes, the write-optimized policy does not fragment the contents of the database and has slightly better performance because it uses a bigger cache than the read-optimized policy. (The space assigned to the MOB is not used when the workload does not have any writes.) When the workload contains some writes, the write-optimized policy does not preserve the clustering of related objects and therefore has overall worse performance than the read-optimized policy. Therefore, Figure 4-8 shows that for most workloads the read-optimized disk layout policy in combination with a MOB provides much better performance than a write-optimized disk layout policy.

4.3.7 Clustering

As described in Section 4.1, the amount of clustering is controlled by two workload parameters: cluster region size and cluster length. Figure 4-9 shows the performance of the two disk layout policies as the size of the cluster region is increased from 32 to 2048 objects. When the cluster

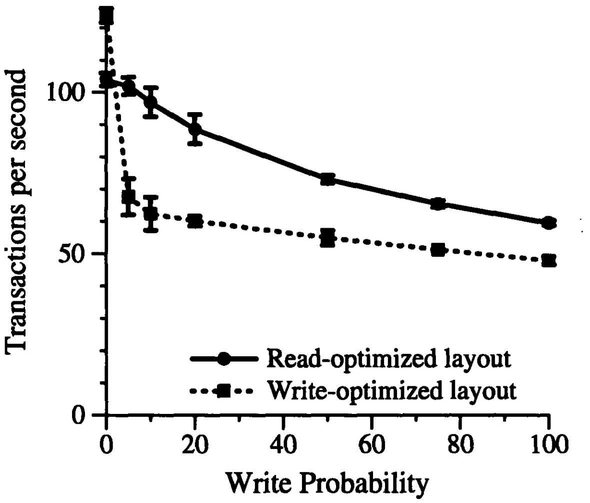


Figure 4-8: Performance of the two disk layout policies over a wide range of write probabilities. Each policy was assigned the optimal division of 24 megabytes of server memory between the cache and the MOB that was computed for a 20% write probability.

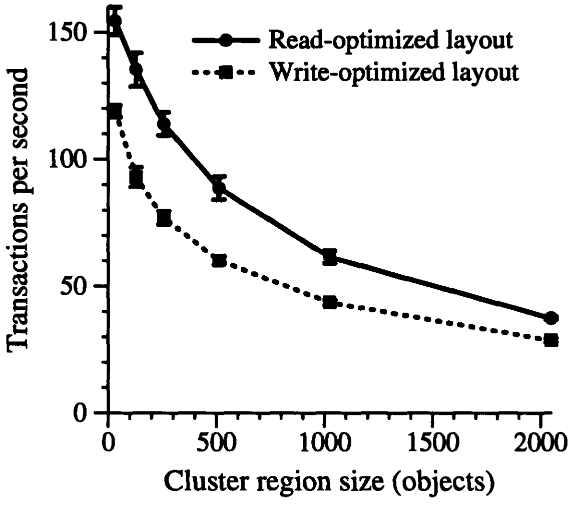


Figure 4-9: Performance of the two disk layout policies as the size of a cluster region is varied. Each clustered access picks 32 objects out of a cluster region. Therefore as the cluster region size is varied from 32 to 2048 objects, the fraction of the region touched by a clustered access falls from 100% to 1.56%.

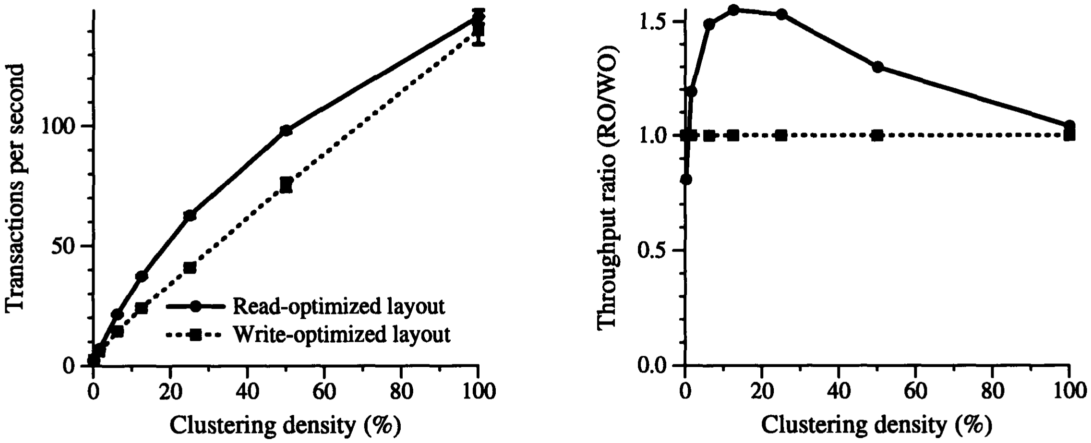


Figure 4-10: Impact of clustering density on performance. The graph on the left shows the throughput of the two disk layout policies. The graph on the right shows the throughput of the read-optimized policy expressed as a multiple of the throughput of the write-optimized policy. The transaction length was increased to 512 objects in this experiment to allow each clustered access to touch up to 512 objects.

region size is small, each clustered access touches one page and therefore the performance of the read-optimized policy is very good. As the cluster region size increases, each clustered access touches many pages and therefore the performance of the read-optimized policy drops.

Under the write-optimized policy each clustered region splits up into many fragments, and up to 32 of these fragments have to be read in per clustered access (because the number of objects touched per clustered access is 32). With smaller cluster region sizes, these 32 objects are spread out over a smaller number of fragments and therefore each clustered access generates less disk reads on average.

The conclusion we can draw from this experiment is that smart clustering policies that generate tightly packed clusters of highly related objects will improve system performance under both the read-optimized and the write-optimized policy. Since clustering policies are not the topic being studied in this thesis, we will ignore the impact of the cluster region size from now on. Instead we will focus on the effect of varying the cluster length. We will hold the cluster region size fixed at 512 objects, and therefore it is helpful to convert the cluster length parameter into a *clustering density* parameter, where we define clustering density as the ratio of cluster length to cluster region size. Figure 4-10 shows the impact of varying the clustering density on the performance of the two disk layout policies. Unlike the experiments described so far, this experiment uses a transaction length of 512 objects. (The simulator constrains each clustered access to occur entirely within one transaction, and in this experiment we use clustered accesses that touch up to 512 objects.) Earlier experiments in this chapter all used a transaction length of 128 objects.

At very low clustering densities, the performance of the two policies is similar: at low clustering densities preserving clustering does not improve read performance. As the clustering

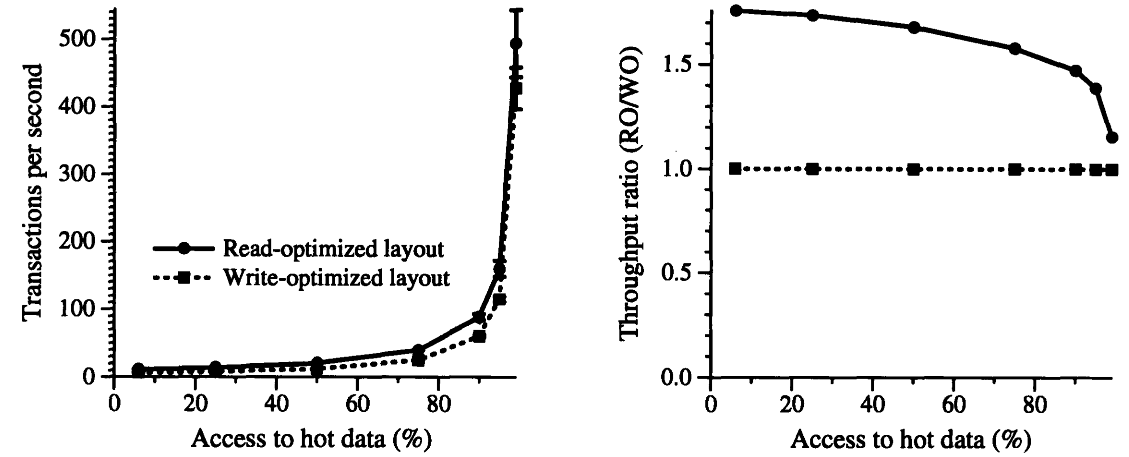


Figure 4-11: Impact of varying the percentage of a client's object accesses directed to its hot region. The graph on the left shows the throughput of the two disk layout policies. The graph on the right shows the throughput of the read-optimized policy expressed as a multiple of the throughput of the write-optimized policy.

density is increased, the performance of the read-optimized policy increases for two reasons: each page read satisfies more client fetches, and each page write cleans out more object modifications from the MOB. The performance of the write-optimized policy also increases because less fragmentation occurs when the clustering density is high.

4.3.8 Locality of Access

Locality of access is very important for obtaining good performance from a wide range of systems. Object-oriented databases are no exception. Figure 4-11 shows the performance of the two disk layout policies as the percentage of accesses directed to the hot region of the database is varied. At the left edge of the graph, the access pattern is completely uniform over the entire database. The client cache is not very effective in this scenario; most object fetches result in client cache misses and therefore the read-optimized policy out-performs the write-optimized policy. At the right edge of the graph the access patterns is extremely skewed (99% of the object accesses are directed to the hot region of the database.) The client cache is very effective for this workload and the load seen at the server consists mostly of writes. Therefore the performance gap between the two policies decreases. However the write-absorption in the MOB also increases as the locality of access is increased, and therefore the write-optimized policy never catches up with the read-optimized policy.

4.3.9 Client Cache Size

The client cache can also have a significant impact on the performance of the system because a large cache can filter out most reads from the workloads presented to the server. Figure 4-12

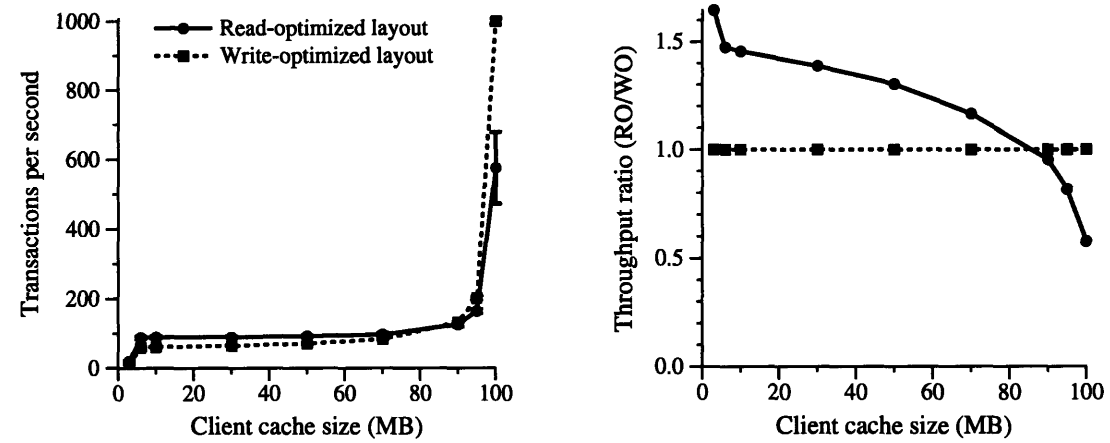


Figure 4-12: Impact of varying the size of the client cache. The graph on the left shows the throughput of the two disk layout policies. The graph on the right shows the throughput of the read-optimized policy expressed as a multiple of the throughput of the write-optimized policy.

shows the performance of the two disk layout policies as the size of the client cache is increased from 3% of the database to the size of the entire database. The big jump in performance as the cache size is increased from 3 megabytes to 6 megabytes is because the size of each client's hot area is 6 megabytes. As the cache size is increased beyond 6 megabytes, the performance of the two policies does not change very much until the client cache becomes large enough to hold more than 90% of the entire database. At that point, very few fetch requests are being sent to the server and therefore the server disk is being used mostly for writing out modifications. The write-optimized policy is much more efficient at handling these modifications than the read-optimized policy and therefore with really large client caches, the write-optimized policy significantly out-performs the read-optimized policy. This result is consistent with Rosenblum's predictions about the usefulness of a write-optimized disk layout in systems that have a large amount of memory [48].

It is important to realize that the write-optimized policy will out-perform the read-optimized policy only if clients have a large amount of memory, and the server do not. If the servers also have a large amount of memory, the read-optimized policy will be able to use a much larger MOB and the resulting write absorption will allow it to compete with the good write performance of the write-optimized policy.

4.3.10 Page Size

The choice of page size has an impact on the clustering of related objects and therefore the performance of both the read-optimized and write-optimized disk layouts depends heavily on the page size used in the simulation. The read-optimized layout policy has to perform a disk seek for every page transfer. When the page size is small, each clustered access touches a large number of pages and the disk seeks required to read all of these pages into the server cache

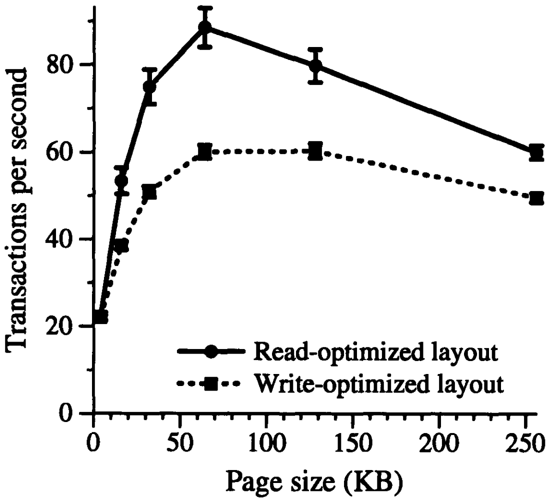


Figure 4-13: Impact of page size on performance

eat up a large chunk of the available disk bandwidth. When the page size becomes larger, fewer disk reads are required per clustered access and therefore disk seek overhead is reduced. However with really large page sizes, much of the data fetched into the server cache by a page read is not used and therefore the time spent reading this unnecessary data is wasted. The results in Figure 4-13 illustrate these points. The read-optimized layout performs best when the page size is in the 64 kilobytes. Smaller and larger page sizes do not work as well.

Under the write-optimized disk layout policy, modified objects that belong to the same page are written as part of the same page fragment. With small page sizes, related objects are more likely to end up in different pages and therefore the clustering of related objects into page fragments will not work very well. As a result individual page fragments will be small and a large number of page fragments will have to be read to process a single clustered access. The throughput of the system will be low because most of the available disk bandwidth will be used up seeking from one tiny page fragment to another. With large page sizes, page fragments will be larger, and therefore the number of disk seeks required per clustered access will go down. However, as under the read-optimized disk layout policy, many unrelated objects will end up being grouped into the same page and therefore when a page fragment is read into the server cache in response to a fetch request, some unrelated objects will be read in along with the related objects. This unnecessary transfer of data into the server cache will reduce the overall server throughput. These effects are illustrated in Figure 4-13. The write-optimized policy has the best performance for page sizes of 64 to 128 kilobytes.

Figure 4-13 shows the impact of page sizes on a workload that has a cluster region size of 512 objects (64 KB). Surprisingly, the optimal page sizes for the two disk layout policies are not very sensitive to the cluster region size. Table 4.10 shows the optimal page sizes for different cluster region sizes. Page sizes of 32, 64, or 128 KB work well for all cluster sizes in the range 4 to 128 KB.

Cluster size (KB)	Optimal page size (KB)	
	Read-optimized	Write-optimized
4	32,64	32
16	32,64	32,64
32	64	64
64	64	64,128
128	64,128	64,128

Table 4.10: Optimal page sizes for the two disk layout policies for different cluster region sizes. Some of the table entries contain multiple values because the performance differences between these values were statistically insignificant. All units are in kilobytes.

Optimal page sizes mostly depend on the geometry and performance characteristics of the disk. A disk operation can be categorized as having two phases: in the first phase the disk head moves to right place on the disk (this phase involves some seeking and rotational latency), and in the second phase the data is transferred. By using larger page sizes, the fixed cost of the first phase is amortized over the transfer time required in the second phase. Disks that have a relatively high set-up overhead will work better with a larger page size, whereas disks with a smaller set-up overhead will work better with a smaller page size.

64 KB is an optimal page size for this simulation study because the disk used in the simulator has large track sizes and high rotational speeds. Disks with smaller tracks and smaller rotational speeds will require smaller page sizes. Disk technology trends indicate that rotation speeds and magnetic densities will keep improving faster than seek latencies. Therefore, larger page sizes may be required for future disks.

4.3.11 Database Size

This section explores the impact of database size on the performance of the two disk layout policies. Figure 4-14 shows the performance of the system as the size of the database is increased from 100 megabytes to 900 megabytes. The size of each client's hot region is held fixed at 6 MB, and therefore as the size of the database increases, the cold accesses spread out over a larger region of the database. For the read-optimized policy, the amount of write absorption in the MOB decreases, and as a result its overall performance drops as well. The performance drop under the write-optimized policy is a result of two factors. First, there is less absorption in the MOB and therefore the average size of page fragments written out to disk drops as the database size increases. This results in poor clustering and worse read performance. Second, with a larger database, installation reads for page headers are less likely to hit in the server cache and therefore more time is spent reading these headers into memory before writing out corresponding page fragments to the disk.

Figure 4-14 illustrates that a relatively small MOB (approximately 20 megabytes) is effective at improving the write performance of a read-optimized disk layout policy for large

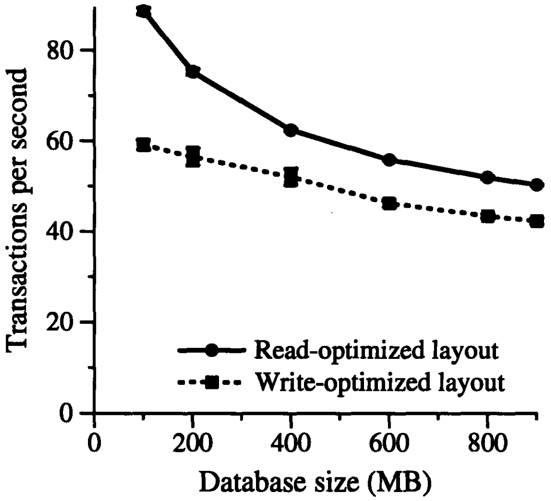


Figure 4-14: Impact of database size on performance

databases. The MOB does not have to be scaled with the size of the database because much of the MOB's effectiveness arises from the absorption of modifications to the hot and warm regions of the database. As long as a significant fraction of these regions fits inside the MOB, the MOB will significantly improve write performance regardless of the size of the database. Of course adding more memory to the server will improve the performance of both disk layout policies. A larger MOB will provide better write absorption under both policies and also reduce the amount of fragmentation under the write-optimized policy. A larger server cache will reduce the number of page header installation reads required under the write-optimized policy.

4.3.12 Object Size

The impact of varying the object size is illustrated in Figure 4-15. Clustering is more important for small objects than it is for big objects because a disk read of a single big object will amortize seek and rotational latency over a larger data transfer than a disk read for a single small object. Therefore as the object size increases, the performance differences between the read-optimized and write-optimized policies decrease. The overall throughput of both policies drops as the object size is increased because each transaction touches 128 objects and therefore as the object size increases the amount of data touched by each transaction increases.

Limitations in the simulator implementation prevented running experiments with object sizes larger than two kilobytes. However it is easy to predict that at large object sizes (say object sizes that approach page sizes), the read performance of the two disk layout policies will be similar. The write performance of the write-optimized policy will be better than the write performance of the read-optimized policy because the latter will require installation reads. Of course these installation reads can be avoided by placing large objects in pages of their own. A

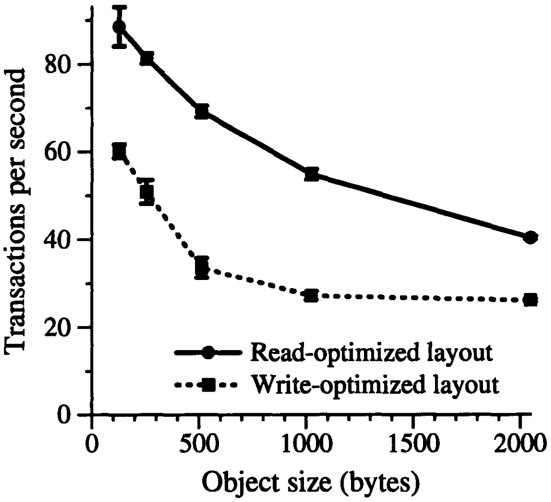


Figure 4-15: Impact of object size on performance

page that holds a single object does not require any installation reads because the entire page contents can be replaced whenever the contained object is modified.

4.3.13 Hybrid Policy

The experiments described so far have focused on the read-optimized and write-optimized policies. Results for the hybrid policy have not been presented because in the vast majority of cases, the performance differences between the hybrid and read-optimized policies are minor. The reason for this is that the hybrid policy has similar read performance to the read-optimized policy because both policies preserve the clustering of objects into pages. The only difference between the performance of the two policies is that under the hybrid policy modified pages are written out sequentially into a large empty disk region whereas under the read-optimized policy, the pages are written back out to their original positions using the read-modify-write cycle described in Section 3.2.1. A simple analysis at the end of this section shows that this difference in writing policies will lead to at most a 10% better throughput under the hybrid policy. Because of this small gain in performance, and the complication of adding a disk cleaner, the hybrid policy was abandoned and not implemented in Thor.

Write Performance Analysis

This section contains an analysis of the hybrid policy. The analysis shows that the hybrid policy does not provide very significant write performance benefits over the read-optimized policy. This analysis builds on the analysis of the read-modify-write cycle presented in Section 3.2.1. As Section 3.2.1 shows, each read-modify-write cycle has the following components:

1. an average length seek

2. an average rotational latency of half a revolution before the read
3. the transfer time for the page read
4. rotational delay in which the server turns around and initiates the disk write, and the disk controller waits while the disk head rotates back to the beginning of the page
5. the transfer time for the page write

The write portion of the cycle occupies the last two steps. The disk head position at the beginning of step 4 is the same as the position of the head at the end of step 5. Therefore the write portion of the cycle takes exactly one disk revolution. Since we are writing a 64 kilobyte page and the track size is 89 kilobytes, the page write operates at a rate $64/89 = 0.72$ times the peak bandwidth of the disk. The hybrid policy on the other hand writes out many pages with one sequential disk write and therefore obtains almost the peak disk bandwidth. For most of the studied workloads the fraction of time the disk is busy writing under the read-optimized policy is less than 30%. (30% is an extreme number; under most workloads writes accounts for less than 20% of the disk utilization.) In the hybrid policy write times will be only 0.72 times the write times of the read-optimized policy. Since the read performance of the hybrid policy and the read-optimized policy is similar, the total amount of disk activity under the hybrid policy will be at least

$$70\% + (30\% \times 0.72) = 91\%$$

of the activity under the read-optimized policy. Therefore the hybrid policy will provide at most 10% better throughput than the read-optimized policy. This performance analysis does not take any cleaning costs into account, and therefore in general the performance benefits of the hybrid policy will be lower than 10%.

4.4 Summary

Here is a summary of the conclusions drawn from the simulation experiments described in this chapter.

- The modified object buffer substantially improves the performance of the read-optimized disk layout policy. For the workload used in this chapter, the throughput of the read-optimized policy increases by over 200% as the MOB size is increased from 1 MB to 12 MB.
- The performance benefits of the MOB arise from the extra write absorption present in a system with a larger MOB. For example, in a system where the size of the MOB is 10% of the database size, and each transaction modifies 10% of a page, the MOB reduces the total number of writes to one third of the number of writes that would be required without a MOB.

- A large server cache is not useful under the read-optimized disk layout policy because once client access patterns have been filtered through the client caches, the resulting stream of fetch requests arriving at the server demonstrates very poor locality. A server cache just has to be large enough to hold individual pages in memory until the useful portions of these pages have been fetched into the client cache. The rest of the server memory should be devoted to the MOB.

- The write-optimized disk layout policy destroys the on-disk clustering of related objects, and therefore it exhibits poor read performance. Even though it writes modifications out to the disk very efficiently, when this write efficiency is combined with its poor read performance, the overall throughput of the system is worse than that of a system that uses a read-optimized disk layout policy.

- The read-optimized policy provides better overall throughput than the write-optimized policy under a wide range of workloads that cover all likely combinations of different write probabilities, clustering densities, access skews, database sizes, and object sizes.

- When the client cache is large enough to hold the entire region of the database that is ever accessed by the client, a huge fraction of the client fetches hit in the client cache, read performance at the server becomes a non-issue, and the write-optimized policy provides better performance than the read-optimized policy. This situation fits the assumption made in the log-structured file system that most reads hit in a cache [48]. However, given that memory prices are much higher than disk prices, most people will not be able to afford to configure all of their machines with enough memory to hold a copy of all of their persistent data.

Chapter 5

Page-Shipping Systems

Most databases and file system organizations devote all of server memory to a page cache. We call this architecture a page cache architecture (or PAGE for short.) The MOB architecture also has a page cache that is used to satisfy fetch requests, but in the MOB architecture only a portion of the server memory is used for the cache: the rest of the memory is used for the MOB. In PAGE, modifications are inserted directly into the page cache, but in the MOB architecture modifications are inserted into the MOB, and the contents of the page cache never contain any modifications.

The choice between PAGE and MOB depends on how clients ship modifications back to the server. The preceding chapters have focused on a system in which clients ship modified objects to servers at transaction commit. We now switch gears and examine a system in which clients ship back entire pages at commit. In this comparison of MOB and PAGE we assume that the server uses a read-optimized disk layout. As Chapter 4 showed, the write-optimized disk layout policy does not perform well because it destroys clustering.

The different possible combinations for system organization are:

1. Object-shipping and PAGE
2. Page-shipping and PAGE
3. Object-shipping and MOB
4. Page-shipping and MOB

PAGE is not practical in an object-shipping system. The server has to perform installation reads at transaction commit so that the incoming modified objects can be installed into pages before the pages are inserted into the cache. Results of various studies have shown that these immediate installation reads can significantly degrade system performance [45, 56]. The Quickstore study described in [56] used these results to conclude that page-shipping is better than object-shipping, but it did not consider a MOB architecture that can delay installation reads until modifications have to be flushed out of server memory.

PAGE in an object-shipping system will have strictly worse performance than PAGE in a page-shipping system because the first combination will require installation reads whereas the second combination will not require any installation reads. Therefore we ignore the first combination in our study. The last two combinations, object-shipping and page-shipping in a MOB architecture, are similar in their use of the disk. In both cases only modified objects are stored in the MOB, and the server has to perform installation reads when flushing modifications to disk. Therefore in this chapter we will focus on a page-shipping system and compare the performance of PAGE and MOB in such a system.

The simulations presented in this chapter show that if only small portions of a page are being modified together, MOB provides better performance than PAGE. MOB holds just the modified objects in memory, but PAGE has to hold entire pages in memory, even if only a small portion of each page is dirty. The MOB architecture can make more efficient use of a limited amount of memory and thus provides better write performance than the PAGE architecture.

If transactions modify large portions of a page at once, PAGE can provide up to twice the throughput of MOB. Under such workloads PAGE does not waste as much memory by caching entire dirty pages, and therefore its write performance is comparable to that of MOB. PAGE has the added benefit that it never has to perform any disk reads when it is lazily writing out modifications to the disk: the pages are already in memory. MOB on the other hand has to perform these installation reads so that individual modified objects can be installed into appropriate locations on the corresponding pages before the pages are written back to disk with the new modifications. Installation reads reduce the performance of MOB below that of PAGE. If we ignore the cost of these installation reads, the performance of MOB is competitive with that of PAGE even when transactions modify large portions of a page at once.

It is important to remember that these results only apply to page-shipping systems. As described above, PAGE does not perform well in an object-shipping system. Many object-oriented database implementations use object-shipping for important reasons that are described in Section 5.4. For these object-shipping systems MOB will be a more efficient server architecture.

5.1 PAGE Simulation

The PAGE architecture was added to the simulator described in Chapter 4. In the PAGE simulator all of server memory is dedicated to a page cache. The operation of the simulated server is simple: as modifications arrive at the server, they are streamed to a stable log, and the corresponding pages are installed into the page cache if necessary and marked as dirty. Since we assume that the clients ship back entire pages at transaction commit, the server does not have to perform any installation reads. The modified pages can be inserted directly into the page cache (perhaps after evicting some other pages from the cache as described later in this section.)

When the server receives a fetch request, the appropriate page is read from the disk into the page cache if necessary, and then the contents of the entire page are sent back to the client. The client may not need the entire page, and therefore this blind prefetching of the entire page may generate more network traffic than would be generated by a smarter prefetching policy that takes relationships among different objects into account when deciding which objects to send back to the client. But since the simulator does not model network utilization, this simple-minded prefetching policy does not hurt the performance of PAGE.

A simple LRU replacement policy is used to manage the page cache: when the server has to insert a new page into the cache (in response to either a fetch request or a transaction commit), the least recently used page in the cache is evicted to make room for the new page. If the evicted page is dirty, it is written out to disk before being removed from the cache. Commit and fetch latencies will be lower if the server always keeps a few free slots for incoming fetch and commit requests by writing out dirty pages before it is absolutely necessary to do so. This optimization is not implemented in the simulator. Since the system is disk-bound, adding the optimization will not improve the throughput of PAGE because the optimization does not reduce the total number of disk operations, it just overlaps some disk activity with client activity.

5.2 MOB Simulation

The MOB architecture in a page-shipping system is similar to the MOB architecture in an object-shipping system. In particular, there is no difference in the operation of the flusher thread that writes modified objects from the MOB to the disk. The only significant difference is at transaction commit. Since clients are shipping back entire pages, the server has to identify the modified portions of the page so that just the modified objects are installed into the MOB.

However, this identification of modified objects is also necessary in PAGE. Suppose the page contains objects *a* and *b*, and the committing transaction has modified *a* and some other recently committed transaction modified *b*. The page arriving from the client might have an old version of *b*, and therefore the PAGE server will have to store the new value of *b* into the incoming page before this page can be installed into the page cache.

Therefore we assume that each commit request contains enough information to allow the server to find the location of individual modified objects in each incoming page. The MOB server uses this information to extract the modified objects from the incoming pages. Once the modified objects have been extracted from the pages, these objects are stored into the MOB and the rest of the pages are discarded.

5.3 Results

The workload described in Chapter 4 was used to evaluate the performance of PAGE and compare it to MOB. The experiments varied the clustering density and the write probability of the workload to determine the sensitivity of PAGE to the workload characteristics and also to compare it to MOB under a wide range of conditions. Most of the default values of the simulation parameters from Chapter 4 are used unchanged in these experiments. The transaction length was changed from 128 object accesses to 512 object accesses to allow us to simulate transactions that fetch all of the objects from one page. The values of the simulation parameters are listed in Table 5.1. Both MOB and PAGE were assigned 24 megabytes of server memory. MOB used the optimal division of 24 megabytes into 19 megabytes for the MOB

Workload Parameters	
Object size	128 bytes
Database size	100 megabytes
Number of clients	12
Objects fetched per transaction	512
Write probability	From 0 to 100%

Clustering Parameters	
Cluster region size	512 objects
Cluster length	From 1 to 512 objects

Skewed Access Parameters	
Hot area size	6 megabytes
Access to hot area	90%
Client cache size	6 megabytes

System Parameters	
Page size	64 kilobytes
Disk type	Seagate ST18771FC
Total server memory	24 megabytes

Table 5.1: Simulation parameters for page cache study

and 5 megabytes for the cache. (See Section 4.3.5). PAGE used all 24 megabytes for its page cache.

Even though this study was conducted in the context of a page-shipping system, the modified object buffer stores just the modified portions of the pages shipped back by the client. Therefore, when modifications are finally written out from memory to the disk, MOB has to perform installation reads before it can install the modified objects into the appropriate places on the corresponding pages. PAGE on the other hand does not require any installation reads. To gauge the cost of these installation reads, we also studied a variant of MOB that can perform installation reads for free. This architecture will be called the *free installation read* variant of the MOB architecture (or the FREEMOB architecture for short). The FREEMOB architecture is actually feasible to implement by using a variant of the client–server cooperative caching scheme described by Franklin [20]. Instead of performing installation reads from the disk, the server can fetch the required pages from client caches. These fetches may sometimes fail because the client has evicted the corresponding page from its cache, and the fetches will also increase overall network traffic. However, the FREEMOB scheme serves as a useful first approximation to a system that uses cooperative caching between clients and servers to reduce the number of installation reads. For example, installation reads for the hot and warm regions of the database can always be avoided by fetching the appropriate pages from one of the client caches. Installation reads for the cold region of the database may also be avoided if the client that generated the modification to the cold page has not yet evicted the contents of the cold page from its cache. (See Section 4.1.2 for a definition of hot, warm, and cold regions.)

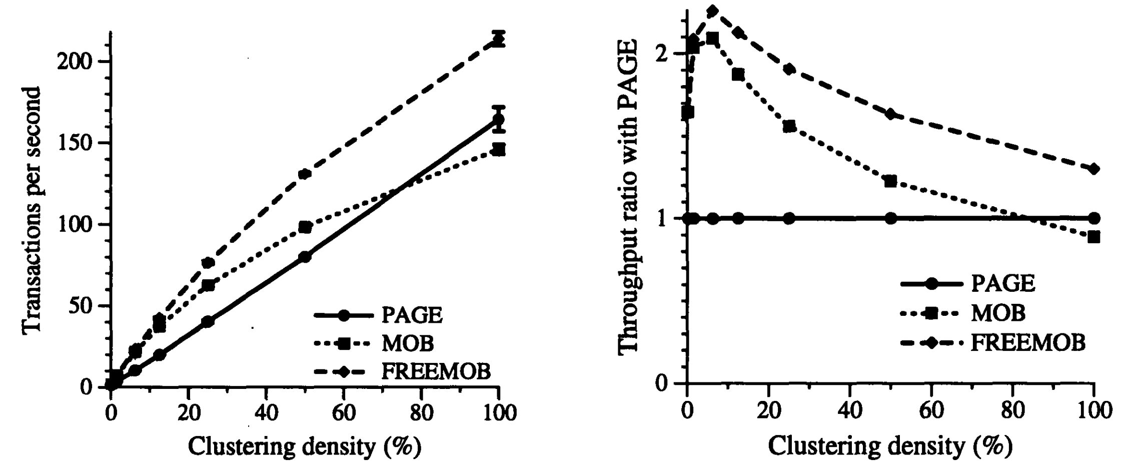


Figure 5-1: Impact of clustering density on performance. The graph on the left shows the throughput of the server architectures. The graph on the right shows the throughput of all of the architectures expressed as a multiple of the throughput of the PAGE architecture.

5.3.1 Clustering Density

The main difference between PAGE and MOB is that one stores entire dirty pages where the other stores just modified objects. Therefore the performance differences between the two organizations will be highly dependent on the clustering patterns of the workload. Figure 5-1 plots the performance of the different storage organizations as the clustering density of the workload is varied. (Recall that clustering density is defined to be the fraction of each page touched by a clustered access.) At low clustering densities, MOB provides much better performance than PAGE because only small portions of each page are dirty and PAGE wastes a lot of space by storing these pages in their entirety. As the clustering density increases, the space wasted in PAGE decreases and therefore its performance relative to MOB improves. At really high clustering densities MOB requires installation reads whereas PAGE does not require any installation reads. Therefore the performance of PAGE eventually exceeds that of MOB. However, if we allow MOB to complete its installation reads at no cost (for example by fetching the relevant pages from client caches instead of from disk), its performance is always superior to that of PAGE. This is because the write probability in these experiments is 20% and therefore even with a clustering density of 100%, only 20% of each page is being modified and therefore PAGE is wasting a significant amount of memory even with a clustering density of 100%.

5.3.2 Write Probability

The previous experiment shows that when the clustering density is low, MOB provides better performance than PAGE. The experiment also shows that a high clustering density alone is not sufficient to get superior performance from PAGE because even if the clustering density is high,

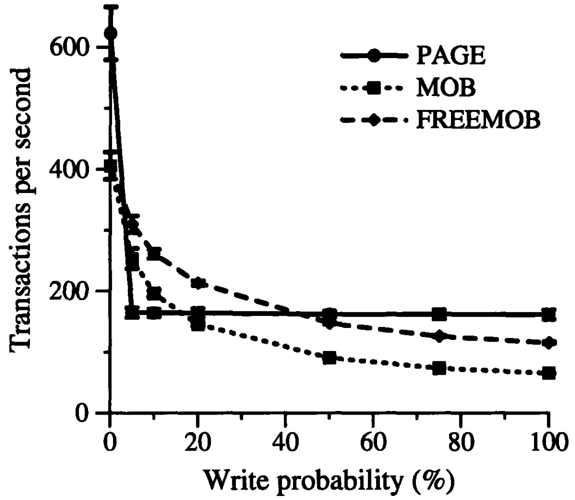


Figure 5-2: Impact of varying the write probability under a workload with a clustering density of 100%.

as long as the write probability is low, only small portions of each page will be modified and therefore PAGE will waste memory. In this section we explore the impact of varying the write probability under a workload with a high clustering density. Figure 5-2 shows the performance of MOB and PAGE as the write probability is varied for a workload with a clustering density of a 100%. The performance at 0% write probability is a special case: there are no modifications, and therefore the portion of server memory dedicated to the modified object buffer is lying useless. PAGE on the other hand can use its entire server memory to satisfy fetch requests and therefore PAGE provides better performance. However, as we increase the write probability, reads and writes start competing for cache space in PAGE and its performance drops sharply as soon as the write probability becomes non-zero. If we increase the write probability further, the performance of PAGE is not affected because it caches entire pages regardless of how much of the page has been modified. MOB on the other hand provides better performance overall until the write probability exceeds 20%. At that point its performance drops below that of PAGE because MOB requires installation reads whereas PAGE does not. It is important to remember that write probabilities much higher than 20% are exceedingly rare. For example, in the applications studied by Chang and Katz, overall write probabilities of different applications did not exceed 25% [11].

The FREEMOB architecture out-performs PAGE for write probability under 50%. At really high write probabilities PAGE provides better performance because under PAGE all of server memory is managed in an LRU fashion, whereas under the MOB architecture the contents of the modified object buffer are managed in a FIFO fashion. Therefore, PAGE will tend to never write out pages that are being modified frequently, but MOB will periodically write out such pages.

It is possible to manage the contents of the MOB in an LRU fashion as well by having the flusher thread write out only the objects that belong to a page that has not been modified recently. The only drawback is that stable log truncation may become more complicated than in the current system. In the current system modifications are flushed out in log order, and therefore a tail of the log can be discarded whenever some modifications are written out. The details of how log truncation will work in a system in which the MOB is managed in an LRU fashion have not been worked out in this thesis and are left as future work.

5.4 Discussion

The results of the experiments described in this chapter show that the relative performance of MOB and PAGE is highly dependent on the workload: if either the clustering does not match the application's access patterns, or if the workload does not have an excessive number of writes, MOB provides better performance than PAGE because PAGE has to store entire pages in memory even if only part of the page has been modified. For example, with a write probability of 20%, MOB has better performance for clustering densities in the range 0 to 90%.

If most of a page is modified at once, then PAGE provides better performance for two reasons. First, the entire server memory in PAGE is managed with an LRU replacement policy. Under MOB, the portion of the memory dedicated to the modified object buffer is managed in a FIFO fashion. Therefore PAGE can do a better job of handling access patterns where most modifications are to a certain hot region of the database. Under PAGE, pages that belong to this hot region will never be written out, whereas under MOB, these pages will be written out periodically. Second, PAGE does not require any installation reads whereas MOB requires installation reads. The simulation results show that the performance of MOB can be improved significantly by using a cooperative caching scheme to reduce the number of installation reads: installation reads can be replaced by page fetches from client caches. These page fetches will occur in the background, and as long as the system is not network bound, the cost of these page fetches will not affect the overall throughput of the system.

It is important to note that this comparison of PAGE and MOB is incomplete. PAGE does not work well in object-shipping systems because in such systems a PAGE server will have to perform expensive installation reads at commit time to convert incoming modified objects into dirty pages that can be stored in the page cache. There are many good reasons for preferring the use of an object-shipping system. The simulator only models the disk usage of the system, and therefore the results presented in this chapter do not reflect all of the ramifications of a page-shipping system:

- A page shipping system requires that client caches be managed in terms of pages instead of objects. Even if the application is using only a small part of a page, the entire page will have to be stored in the client cache in a page shipping system. In an object-shipping system, client caches can be organized as object caches that store just the portions of the page that are currently being used by the application. Many object-oriented database systems use an object cache at the client for this reason [16, 34, 38, 40].

- Page-shipping architectures can have high network bandwidth requirements: if only part of a page is modified by a transaction, clients in an object-shipping architecture will send just the modified portions of the page whereas clients in a page-shipping architecture will send the entire page. Therefore page-shipping will have inferior performance over slow network links such as phone lines.

- An object-shipping architecture also provides a simple mechanism for performing concurrency control at an object-level. In a page shipping architecture, object-level concurrency control can require complicated merging of modified pages sent back by different clients [31].

More details on differences between object-shipping and page-shipping architectures can be found in the literature [13, 45, 56].

Given all of these differences, it is clear that MOB is a better storage management policy than PAGE unless the system is a page-shipping system and most modifications modify large portions of a page. We expect workloads that modify large portions of a page at a time to be rare. Chang and Katz point out several important characteristics of object-oriented database applications that support our claim [11]. They studied several VLSI/CAD applications layered on top of an object-oriented database and found that access patterns for the same data varied widely depending on which application was used to access the data. This finding suggests that no clustering policy will work perfectly for all application access patterns and therefore in general the database system should not be designed under the assumption of perfect clustering. Chang and Katz also found that reads are much more common than writes. For one phase of one of the applications, the write probability was as high as 65%, but each application's overall write probability ranged from a tiny fraction of one percent to 25%.[1] These findings suggest that many object-oriented database applications will not modify large portions of a page together and therefore PAGE will not be a good storage management policy for such applications even if the entire system is organized as a page-shipping system.

[1] Chang and Katz report read/write ratios instead of write probabilities. The total number of accesses in each application can be obtained by adding together the number of reads and writes. Therefore I used the formula $1/(1 + r/w)$ to convert from the read/write ratio (r/w) to the write probability metric used in this thesis.

Chapter 6

Implementation

The results of the simulation study described in Chapter 4 were validated by implementing some of the proposed storage management techniques as part of the Thor object-oriented database system [38]. The implementation of Thor partitions server memory into a modified object buffer and a page cache as described in Chapter 2. Both the read-optimized and the write-optimized disk layout policies are supported as part of the Thor server. This chapter contains a detailed description of the implementation of the relevant parts of Thor. The chapter also describes the results of running the OO7 benchmark [8, 9] on Thor. These performance results confirm the important conclusions of the simulation study: the MOB improves system performance, and the read-optimized disk layout policy provides better performance than the write-optimized disk layout policy.

6.1 Client Implementation

The client run-time system for Thor provides a number of useful properties such as the ability to run Theta [37] programs, garbage collected storage, interfaces to multiple languages, and isolation from unsafe languages such as C and C++ [38]. However, the implementation of this run-time system was unstable and incomplete at the time the experiments described in this chapter were run. Therefore, I wrote my own simple implementation of the client run-time system for running these experiments. This section contains a description of this alternate run-time system. The run-time system described here is designed to be used only by C++ programs; it does not provide any isolation from errors in application code; and it does not support transactions that span multiple servers. Also, the cache management and object layout policies used in the client are sub-optimal. However, these deficiencies in the implementation of the client do not affect the results described in this chapter: those results are dependent only on the storage management policies used at the server, and not on small details of the client implementation.

Each client has a cache that stores recently accessed objects. The client processes a sequence of transactions, and caches objects across transaction boundaries. Each transaction runs entirely inside the client cache. If an object required by the executing transaction is not present in the client cache, a *fetch request* is sent to the server. The server responds with the

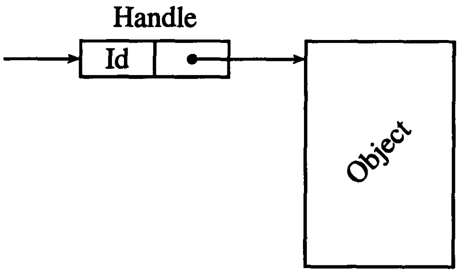


Figure 6-1: Object layout at the client

requested object (and possibly some other objects as described in Section 6.2.1.) The fetched objects are stored in the client cache. Modifications are made to cached copies of objects. When a transaction finishes, all of the modified objects are shipped back to the server as part of a *commit request*. The processing of a commit request at the server is described in detail in Section 6.2.3.

6.1.1 Object Layout and Access

The client cache holds copies of objects fetched from the server. Because objects can be evicted from the cache as the cache fills up, all pointers to an object have an extra level of indirection through a small *handle* object as shown in Figure 6-1. Each handle holds the object identifier of the corresponding object and can be in one of two states: *full*, or *empty*. An empty handle has a nil pointer that indicates that the corresponding object is not present in the client cache. A full handle contains a pointer to the cached copy of the object. If an object has to be evicted from the cache, the corresponding piece of memory is deallocated, and the pointer in the handle object is set to *nil*. All object accesses check for a nil pointer in the handle. If that pointer is nil, a fetch request is sent to the server with the object identifier found in the handle.

A hash table maintains a mapping from object identifiers to the corresponding handles. When objects arrive at the client, they contain references to other objects in the form of object identifiers. The run-time system *swizzles* these object identifiers on demand: when an object is first accessed, all object identifiers in the referenced object are converted to the handle objects found by looking up the object identifiers in the global hash table. An empty handle is created when swizzling an object identifier that is not present in the global hash table.

More efficient object management techniques such as *node marking* and *edge marking* have been described in the literature [13]. The indirection technique described here may not perform as well as these other techniques, but is easy to implement and does not require the cooperation of a garbage collector.

6.1.2 Cache Management

The contents of the cache are not laid out explicitly by the client run-time system. Instead, a standard memory allocator is used to allocate space for objects as they are fetched into the client. As objects are evicted from the cache, the corresponding pieces of the client memory are explicitly de-allocated. The run-time system keeps track of the total amount of allocated memory, and evicts objects from the cache when this amount exceeds the configured cache size. By choosing an appropriate cache size based on the amount of available physical memory, users of the system can avoid any paging by the virtual memory system.

In the current implementation, objects are evicted from the cache only at the end of a transaction. Each object has a set of flag bits maintained in a special header word. One of these bits is a *use-bit* that is set whenever the object is read or written. If the amount of allocated memory exceeds the cache size, the hash table is scanned, and any object whose use-bit is not set is evicted from the cache. In addition, all use-bits are cleared as part of this scan. Therefore the cache replacement policy throws out all objects that have not been used since the last time the cache was scanned. Studies by Day show that a least-recently-used policy for evicting objects provides better performance than the simple one-bit scheme described here, but is hard to implement efficiently [13]. The single bit policy described here works quite well for hot-cold workloads if the cache space is made 50-60% bigger than the anticipated hot region size. Therefore, I implemented the single bit cache replacement policy and used extra client cache space when running the experiments described in this chapter.

6.1.3 Transaction Management

Thor uses an optimistic concurrency control mechanism [1, 2, 27]. Concurrency control checks are performed at the server when a transaction commits. Therefore the client has to send the server the identifiers of all objects read and written by the transaction. The relevant information is collected at the client as follows: the client run-time system keeps track of the list of objects read and written so far by the current transaction: the *access list*. Each object also has a *read-bit* and a *write-bit* that are stored in the header word of the object along with the use-bit described earlier. The run-time system maintains the invariant that an object is stored in the access list if and only if its read-bit is set. Object access proceeds as follows: if the object's read-bit is not set, it is appended to the access list and the read-bit is set. If the read-bit is already set, then the object is already present in the access list and therefore does not have to be stored there again. Therefore a simple bit check on each access is sufficient to prevent duplicates in the access list. Modifications to an object are handled in exactly the same way as object reads, but in addition the write-bit is also set.

At the end of the transaction, the access list is scanned, and the values of all objects that have their write-bit set are sent to the server. The identifiers of all of the objects present in the access list are also sent to the server to allow the server to carry out the required concurrency checks. When the transaction has committed (or aborted because of a failure or a concurrency control violation), all of the read-bits and write-bits, as well as the access list are cleared and the next transaction is started.

6.1.4 Object Creation

New objects are created in the client heap just like normal C++ objects. These objects are initially not persistent and are not stored in the cache. Transient objects become persistent only when they become reachable from some previously persistent object. This promotion of transient objects to persistent status happens at transaction commit time. All of the persistent objects modified as part of the transaction are scanned for pointers to transient objects. If any such pointer is found, that transient object is added to the *new object set* for the transaction. Furthermore, each object added to the new object set is also recursively scanned for pointers to more transient objects. The new object set is sent to the server along with the transaction access list and the modified objects. If the transaction commits, the server responds with object identifiers for the new objects. When these object identifiers are received at the client, they are stored in the handles for the corresponding objects, these handles are entered into the global hash table that maps from object identifiers to object handles, and the newly persistent objects are moved into the client cache.

6.2 Server Implementation

The client implementation described in the previous section is not the real client implementation used in Thor because the real client was not stable enough to serve as the basis for the experiments described in the thesis. However Thor's server implementation works correctly and was used unchanged for these experiments. This section describes the implementation of the Thor server.

The server is organized as a multi-threaded user-space program. It can store the database persistently either on a dedicated disk partition, or on a file that is part of the file system. The server memory is partitioned into a MOB and a cache. Clients connect by making TCP/IP connections to the server. As each connection is established, the server creates a new *client manager* thread. All communication with a particular client is handled by the client manager thread created for that client. If one client manager thread is blocked waiting for an expensive operation such as a disk read to complete, other client managers are not blocked, and can continue processing client requests. The server also has a *flusher thread* as described Chapter 2. The flusher thread is responsible for moving modifications from the MOB to the disk. Each server also contains some other threads that communicate with other servers to implement distributed transactions. However, the distributed transaction protocols used in Thor are beyond the scope of this thesis and are not discussed here.

6.2.1 Fetch Requests

When a client manager thread receives a fetch request from a client, it looks for the object first in the MOB and then in the server cache. (If an object is present in both the MOB and the cache, the version in the MOB is guaranteed to be the latest version.) If the object is not present in either of the two places, the thread reserves some cache space, issues a disk read to fetch the appropriate page (or page fragment in the case of a write-optimized disk layout policy) into the cache, and then blocks waiting for the disk read to complete. Once the disk read is completed, the search for the object in the MOB and the cache is restarted, and this time the search will succeed. It is important to restart the entire object location process when the disk read completes because while this thread was blocked waiting for the disk read to complete, another thread may have installed a new version of the object into the MOB.

The server also sends a number of objects that are related to the requested object. However, it only sends objects that are in server memory; the server never performs additional disk I/O for sending objects that the client has not explicitly requested. The related objects to be sent to a client are determined by a simple breadth-first search of the pointer graph starting at the object requested by the client. The breadth-first search is stopped after 32 objects have been sent to the client.

A set of recently sent objects (the *prefetch set*) is maintained at the server for each client and objects in this set are sent only if explicitly requested by the client. The prefetch set prevents the server from repeatedly sending the same object to the client because of cycles in the pointer graph. The size of the prefetch set is limited to about a thousand entries to limit the consumption of valuable server memory. The prefetch set is cleared whenever it reaches this threshold size. More discussion of prefetching of related objects into the client cache can be found in [13].

6.2.2 Object Location Information

Under the read-optimized disk layout, each page is stored at some fixed location on disk. The page map is itself stored in some special pages that are located at a well-known place on the disk. At recovery time, the server reads the page map into the server cache and locks it in place: *i.e.*, the cache replacement policy never evicts the page map from the server cache. The page map does not occupy very much space: each page requires eight bytes of location information and a gigabyte of data can be stored in 2^{15} 32 KB pages. Therefore the page map for this gigabyte database will occupy just 2^{18} bytes (256 KB).

The page map is formatted as a two-level array indexed by page identifiers. (A two-level array is used so that not all of the page map has to be written out even if just a small part of the page map has been modified.) Each entry in the page map contains the disk location and the size of the corresponding page. Modifications to the page map are treated like modifications to other objects. The modified objects are written out to the stable transaction log and also inserted into the MOB. The on-disk copy of the page map is not modified until these modified objects have to be removed from the MOB to make space for other transactions.

The implementation of the write-optimized layout uses a very simple implementation of the object location map. The entire object location map is stored in memory, and records for each object the disk location and size of the fragment that contains the object. Therefore object location never requires any page header reads in this implementation. Furthermore, in the performance experiments described in this chapter, the read-optimized policy does not get to use the extra memory used for the write-optimized object location map. Even with the extra memory for the object location map, the write-optimized disk layout provides poor read performance as will be shown later in this section.

Under the write-optimized policy changes to the object location map are never explicitly written out to the disk. Instead, the entire disk is scanned at recovery time to reconstruct the object location map. Therefore recovery is much slower under this implementation of the write-optimized policy than would be tolerable in a production implementation. A production implementation would have to periodically checkpoint the contents of the object location map to reduce recovery time. This check-pointing would slow down other activity at the server, but this slowdown is not measured in the experiments described in this chapter. Therefore the experimental setup used in this chapter is biased in favor of the write-optimized policy. Even with this bias, the read-optimized policy provides better performance.

6.2.3 Transaction Commits

The server avoids all disk reads at transaction commit. The optimistic concurrency control checks used in Thor do not require disk reads because only small volatile tables and data structures are used to validate incoming transactions. The details of these checks can be found in [1, 2, 27].

The server also has to check that there is enough disk space available to allow the transaction to commit. (There may not be enough disk space if the transaction has created new persistent objects, or increased the sizes of existing objects.) The server therefore keeps volatile tables that track available disk space and also makes conservative estimates to avoid performing any disk reads at commit time. The server also has to assign object identifiers to new objects. Each particular object identifier belongs to a particular page, and therefore the server has to find an appropriate page to hold each new object. The assignment of object identifiers is complicated by the fact that each page can hold at most 2^{16} objects because of space constraints in object identifiers. Furthermore the size of each page is limited to prevent the accumulation of large numbers of unrelated objects in the same page. Therefore the server has to check the availability of identifiers and space in specific pages before assigning pages to new objects. A volatile *reservation table* keeps track of the free space per page and next free index available in each page to allow the server to perform this check without having to read any information from the disk at commit time.

The reservation table is volatile and therefore its contents have to be reconstructed when a server recovers from a failure. The reservation information for page p is constructed by reading the page header from disk to find the current space and object identifier usage for page p. Furthermore, the stable log is scanned, and any active reservations for page p that are found in the log are inserted into the reconstructed reservation table.

To avoid reading all of the page headers at recovery, the server does not completely reconstruct the reservation table at recovery. Only the portions of the reservation table for pages with active reservations in the stable transaction log are initialized. The rest of the reservation table is lazily initialized as pages are fetched into server memory as part of the normal operation of the server.

6.2.4 Clustering

The server has a preferred page size. The choice of this page size is dependent on the performance characteristics of the server disk. For the experiments described in this chapter, the preferred page size was set to 32 kilobytes.

The server partitions objects into pages according to the order in which objects are created. The server keeps track of a page that is the current target of new objects. At transaction commit time, new objects are assigned to this target page until either the page fills up to the preferred page size, or the space of identifiers within the page is exhausted. When the target page fills up, a new page is allocated and made the new target page. Therefore pages created by this clustering policy all have the preferred page size.

Objects bigger than a certain threshold are treated specially. These objects are assigned a page of their own. Pages for big objects have the exact size required to store the object.

This simple-minded clustering has obvious short-comings. It depends on the application to provide clustering information in a round-about way: by creating related objects and making these objects persistent as part of the same transaction. The other short-coming is that if different clients are creating completely unrelated objects at the same time, the server might end up clustering these unrelated objects together on the same page. As the results in Chapter 4 show, good clustering improves database performance significantly. Therefore a better clustering policy will benefit the Thor server implementation greatly. The simple-minded clustering policy that is currently implemented worked fairly well for the experiments described here because the database was created by a single client while the server was otherwise inactive. Therefore unrelated objects from other applications did not get mixed in with the experimental database.

6.2.5 Modified Object Buffer

The server uses a MOB to delay disk writes (and installation reads in the case of the read-optimized disk layout). New and modified objects are added to the MOB at transaction commit. The combined size of the MOB data structures and the objects stored within the MOB is limited by a configurable server parameter. Allowing the MOB to grow in an unlimited fashion would generate a lot of paging activity that would degrade system performance.

A special *flusher thread* is responsible for moving modifications from the MOB to the data disk when the MOB fills up. The flusher thread is woken up when the MOB fills up beyond a certain percentage. The thread starts scanning from the oldest object in the MOB. As the MOB is scanned, the flusher thread builds up a set of pages that have pending modifications in the scanned portion of the MOB. The MOB scan stops when the flusher thread has scanned a certain fraction of the MOB.

The choice of how much of the MOB to scan, and when to start scanning can have a significant impact on the throughput of the system and the commit latency observed by clients. Starting the MOB scan when the MOB is completely full will increase commit latencies because all commits will have to be delayed until the flusher thread creates more space in the MOB. In general, we cannot avoid these delays if client modifications are arriving faster than the flusher thread can write modifications out to the disk. Therefore, we configure the flusher thread under the assumption that modifications are arriving at the same rate at which they are being cleaned out of the MOB. If the flusher thread scans s percent of the MOB, it starts scanning when the MOB is s percent empty. Under our previous assumption, by the time the flusher thread has cleaned out the targeted s percent of the MOB, the remaining s percent of the MOB will have been filled up with new client modifications. As described in Section 3.2.1, large values of s

are counter-productive. Therefore the Thor server is configured with a value of 10% for s. *I.e.*, the flusher thread starts when the MOB is 90% full, and it scans 10% of the MOB at a time. In the simulator the MOB scan was started when the MOB was 99% full. The implementation of Thor starts the MOB scan earlier to provide better handling of bursty activity that can generate a lot of modifications in a short amount of time. For example, a modification burst of 5% of the MOB size can be handled by the Thor server without requiring any commit delay for MOB space whereas in the simulator the transaction would have to be delayed while the requisite 5% of the MOB was written out by the flusher thread.

Once the scan is complete the thread writes out all pending modifications for the pages identified by the MOB scan. Under the read-optimized disk layout policy, these pages are sorted by disk address and then each page is processed in order. For each page the thread performs an installation read to fetch the page's old contents into the server cache if necessary. All pending modifications for this page are written into the cached copy of the page, and then the new page contents are written out to the disk. Once all of the pages have been processed, all of the modified objects written out by the flusher thread are discarded from the MOB. The MOB is scanned starting with the oldest object, but if a page has some pending modifications in the scanned portion of the MOB, and some modifications in the un-scanned portion of the MOB, all of these modifications will be written out together regardless of age. The data structures that are used to implement these scans efficiently are described in Section 3.1.4.

The flusher thread proceeds in a similar manner under the write-optimized disk layout policy. The MOB scan starts when the MOB is 90% full, but here instead of scanning 10% of the MOB, the MOB scan stops when enough modifications have been identified to fill one disk region. These modifications are streamed out to the disk with a single disk write to a previously empty disk region. When the disk write is complete, the objects that have been written out to the disk are removed from the MOB. Again, if a page has some objects in the scanned portion of the MOB, and some objects in the un-scanned portion of the MOB, all of these objects will be written out together regardless of age in an attempt to reduce page fragmentation on disk.

6.2.6 Stable Transaction Log

The implementation of Thor has been designed to provide highly available storage by using a primary copy replication scheme to replicate the contents of each server. Therefore, stability of the transaction log is guaranteed by keeping a copy of the modified object buffer at both the primary and the backup replicas as described in Section 2.6. However, the current implementation of Thor is incomplete and does not support replication. The contents of the transaction log are stored in the MOB of just one replica, and therefore Thor servers currently cannot survive crashes. When replication is added to the system, the latency of transaction commits will be increased by the amount of time required to send the transaction's modifications to the backup replica over the network. Since networks are fast, the overall transaction latency observed by the clients should not change significantly when replication is added to the system. The impact of the MOB on system performance will also not be affected by the addition of replication: the MOB improves the performance of background writes and installation reads, and this improvement is independent of the commit latency observed by clients.

6.2.7 Implementation Status

All of the pieces of the Thor server described in this chapter are completely functional. However, the server cannot currently recover from failures because modifications generated by committing transaction are not logged to stable storage. We plan to implement primary-copy replication in Thor for high availability. Both the primary and the backup will have un-interruptible power supplies, and therefore stable logging will be implemented by storing the modified objects in the MOB at both the primary and the backup as described in Section 2.6.

Some other pieces of functionality such as garbage collection of unreachable persistent objects, and migration of objects from one server to another have also not been implemented.

6.3 Performance

This section presents some performance measurements of Thor. These measurements use overall system performance to make conclusions about the effectiveness of the MOB, and to compare the read-optimized and write-optimized disk layout policies.

The experiments described in this chapter use the OO7 benchmark [9] to measure the performance of Thor. The OO7 benchmark is designed to model complex CAD/CAM/CASE applications built on top of an object-oriented database. The benchmark measures the speed of pointer traversals, object updates, and simple queries. The Thor implementation does not currently provide any query processing facilities and therefore this thesis just uses the portions of the OO7 benchmark that test pointer traversal and object modification speed.

6.3.1 The OO7 Database

A OO7 database is partitioned into a set of modules. Each module has two distinct layers: the top layer is an *assembly hierarchy*, and below that layer is a set of *composite parts*. (See Figure 6-2.) The assembly hierarchy is a complete k-ary tree of *assembly* objects. Each non-leaf node in the hierarchy has pointers to its k children, and each non-root node has a pointer to its parent. The leaf nodes in the hierarchy contain pointers to k composite parts. The pointers from the leaf nodes to the composite parts are not necessarily distinct: a benchmark parameter c controls the number of composite parts created in each module. The k pointers from each leaf node lead to randomly selected composite parts from this pool of c composite parts. Therefore the same composite part may be reachable by multiple paths from the root of the assembly hierarchy. Each composite part is a randomly constructed graph with n nodes. These nodes are called *atomic parts*, and each atomic part has outgoing connections to e other atomic parts within the same composite part. The object-level representation of the assembly hierarchy and each composite part is described in [9]. For the purpose of understanding the experiments described in this thesis, all that is really necessary to understand about this representation is that each composite part contains many objects: there is an object for each atomic part; there are separate objects for the in-lists and out-lists associated with each atomic part; and furthermore each connection between a pair of atomic parts is itself represented by a special *connection* object.

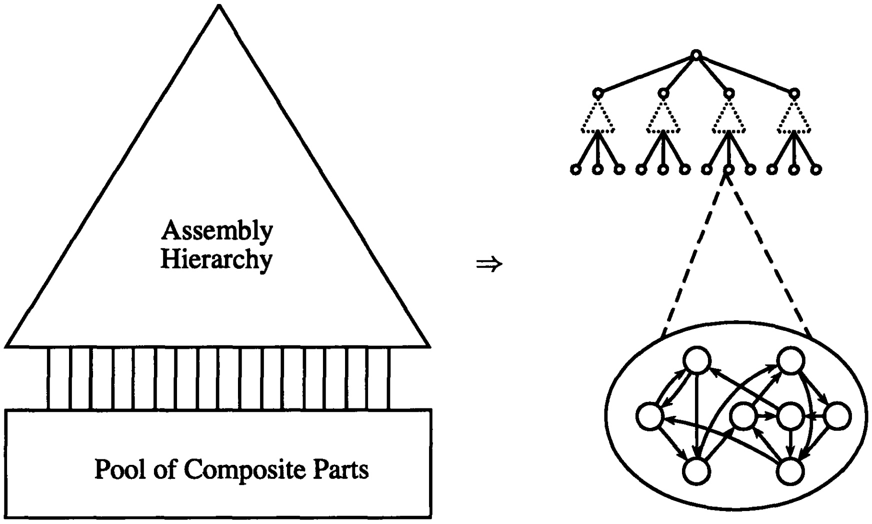


Figure 6-2: A single module in the OO7 database

6.3.2 Database Configuration

The experiments described in this thesis were conducted on a database containing twenty OO7 modules. Each module was created using the "small" configuration defined by the OO7 benchmark. The relevant properties of the database configuration are listed in Table 6.1. The entire database occupies 100 megabytes, the composite parts occupy 95% of the available space, and average object size is quite low (on the order of 80 bytes.) The preferred page size is 32 kilobytes and therefore the database spans 3200 pages. The code used to create the database is a straight-forward adaptation of the sample code provided by the creators of the OO7 benchmark. The creation code creates the contents of a module one composite part at a time. The server clusters objects according to creation order, and therefore the contents of each composite part are clustered tightly into one or at most two pages. (Composite parts that start near a page boundary may not fit within that page and may spill over into the next page.) The assembly hierarchy for each module is also created as a unit, and therefore each hierarchy is also tightly clustered into a small number of pages.

6.3.3 OO7 Workloads

The OO7 benchmark description defines a number of standard traversals that can be run over the contents of a module. However, none of the traversals as defined in [9] test the performance of the MOB or the disk layout policies effectively. OO7's *traversal 1* tests the speed of loading an entire module into the the client cache and also the speed of traversing a module that has already been loaded into the cache. Obviously a traversal over the contents of a cached module will not shed any light on the performance of disk management policies. Even the cold traversal that fetches the contents of the module into the client cache is not all that informative except

No. of modules	20
Branching factor of assembly hierarchy (k)	3
Depth of the assembly hierarchy	7
No. of composite parts per module (c)	500
No. of atomic parts per composite part (n)	20
No. of outgoing edges per atomic part (e)	3

Total no. of objects per composite part	123
Total no. of assemblies per module	1093
Total no. of objects per module	$\approx$ 64,000
Total no. of objects in database	$\approx$ 1,280,000

Size of atomic part object	80 bytes
Total size of composite part	$\approx$ 10 kilobytes
Total size of one module	$\approx$ 5 megabytes
Database size	$\approx$ 100 megabytes

Table 6.1: Database properties

possibly for determining any read performance differences between the read-optimized and write-optimized disk layouts. The cold traversals time of the write-optimized policy will be highly dependent on any write patterns that might have reduced the clustering of the database. Therefore unless we combine traversal 1 with some write pattern, we will not get any useful results about disk management policies.

OO7's *traversal 2* can modify atomic parts as the contents of the module are being traversed. However, even this traversal does not stress the relevant portions of the system. The OO7 benchmark definition runs each traversal at most four times (once to warm up the client cache, and three times to provide a significant measure of the performance of a hot traversal.) Even a small MOB can hold all of the objects modified by these four traversals and therefore these traversals will not cause any disk writes.

Because of these limitations in the definition of the OO7 benchmark, this thesis uses the OO7 benchmark in combination with a hot/cold workload similar to the one used in the simulation studies described in Chapter 4 and Chapter 5. As mentioned earlier, the database consists of twenty modules. Each client is assigned one of these modules, and the client directs 90% of its accesses to one of the modules in the database. The remaining 10% of the client's accesses are spread out uniformly over the other 19 modules. This setup has been designed for use in a multiple-client setup where each client has its private hot region that consists of one module. The contents of the private module can be held in the client cache, but the rest of the database does not fit and therefore accesses to the rest of the database require object fetches from the server. However because of some limitations in our experimental setup we were not able to run experiments with a large number of clients. These limitations, and the corresponding work-around is described in the next section.

6.3.4 Scalability and Experimental Setup

The experiments were run on the Thor research group's set of workstations at night. Most of these workstations are connected by a 10 megabits/second ethernet. This network is relatively slow by today's standards. It is easy to saturate the network with one or two clients that access the server frequently. Therefore we used a 100 megabits/second FDDI ring for our experiments. The limitation in this case is that only three of the machines are connected to the network. If one of these machines is made the server, then there can be at most two clients. Therefore with this experimental setup it is not feasible to run large multi-client experiments directly. Therefore, this thesis uses only single client experiments, but sets up the experiments in a way that allows us to predict the behavior of a larger system.

A system with many clients will either saturate the network, the server processor, or the server disk. Since the focus of this thesis is on disk management techniques, in attempting to scale down from a multi-client system to a single client system, we ignore processor and network saturation issues and instead focus on disk saturation. Disks may become a bottleneck either because of fetch requests that miss in the server cache, or because of the installation-read and write activity that is generated by the flusher thread whenever the MOB fills up. In both cases the amount of disk traffic is highly dependent on the amount of memory used for the server cache and the MOB. Since this memory has to be shared by all of the clients in the system, an easy way to scale down to a single client experiment is to allow the server to use just a small fraction of available memory. Therefore we run our single client experiments with a very small amount of server memory: approximately 2.5 megabytes. A reasonably configured server may have ten or twenty times as much memory, but many clients would have to share this memory and unless the clients were all reading and writing the same portion of the database, each client would effectively benefit from only a small portion of the total server memory.

The complete list of relevant system parameters is given in Table 6.2. The experiments were run on fairly fast machines. (The machines have SPECINT92 ratings of approximately 75.) Following the reasoning given earlier in this section, the server memory was limited to 2.5 megabytes. Because of the relative strengths and weaknesses of the two disk layout policies, under the read-optimized policy two megabytes were assigned to the MOB, and the remaining half megabyte was assigned to the server cache whereas under the write-optimized policy only half megabyte was assigned to the MOB, and two megabytes were assigned to the cache. (See Section 4.3.5 for some simulation results that validate a corresponding division of 24 megabytes of memory for a 12 client system.) The 1.5 megabytes of extra cache space assigned to the write-optimized policy will not decrease its cache miss ratio significantly, but giving it a larger MOB will also not help because the write-optimized policy has very good write performance even with a half megabyte MOB.

The client cache was assigned eight megabytes of memory. This is sufficient memory to allow the client to cache its 5 megabyte private module. Therefore the 90% of the accesses that are directed to the client's private module will not require any fetch requests. Accesses directed to the rest of the database will be likely to miss in the client cache and will generate fetch requests to the server.

Hardware	
Machine	DEC Alpha 3000/400 (133 MHz)
Memory	128 megabytes

Disk	
Type	1 gigabyte DEC RZ26
Size	400 megabyte partition
Speed	2.5 megabyte/second peak throughput
Rotation	5400 RPM
Track size	28.5 kilobytes
Seek Time	Unknown

Configuration	
Server Memory	2.5 megabytes
Client Cache	8 megabytes
No. of clients	One
Communication	100 megabits/second FDDI ring
Free Disk Space	100 megabytes

Table 6.2: Important implementation parameters.

The disk connected to the machine is relatively slow with a peak bandwidth of 2.5 megabytes/second. An example of a more modern disk is the SEAGATE disk used in the simulations. That disk has a peak bandwidth of over 9 megabytes/second. The relative slowness of the RZ26 disk drive used in these experiments actually biases the results in favor of the write-optimized policy. More modern disk drives have much higher transfer rates, but comparable latencies because seek times have not been decreasing as fast as magnetic densities and rotational speeds. Hence smaller operations are relatively more expensive in modern disk drives. The write-optimized disk layout performs very large writes, but on the other hand tends to perform small reads because it has to read individual page fragments. The read-optimized disk layout on the other hand reads and writes entire pages. The MOB significantly reduces the amount of write activity and the performance differences between the two policies hinge on read performance. The read-optimized policy reads larger chunks of data than the write-optimized policy. Therefore as long as disk transfer rates keep increasing faster than seek delays, the performance of read-optimized policies relative to write-optimized policies will keep improving.

The experiments assigned a large amount of free disk space to the write-optimized disk layout policy. The server was allowed to use twice as much disk space as was required by the database. Therefore in all of the experiments described in this section, the cleaner ran very infrequently. Even in the most write-heavy workloads used to drive the write-optimized policy, the cleaner was responsible for less than 3% of the overall disk activity. Therefore the results described in this chapter represent a best-case performance for the write-optimized policy. In configurations with tighter disk space, the performance of the write-optimized policy will drop because of higher cleaning overheads.

6.3.5 Client Workloads

The experiments use the client traversals defined in the preliminary specification of the multi-user version of the OO7 benchmark [8]. All of these traversals start at the root of a module, follow a random path down to a leaf node of the hierarchy and then perform a depth-first traversal of a composite part pointed to by the leaf node. According to [8], the depth-first traversal can be either a read-only traversal, or a write-traversal that modifies all of the atomic parts in the composite part. If we limit our experiments to traversals of this form, then under the write-optimized policy the contents of a composite part will tend to split up into exactly two fragments: one fragment will contain the mutable atomic parts, all of which get modified together. The other fragment will contain the rest of the composite part, *i.e.*, the connection objects and the in-lists and out-lists. This fragmentation will reduce the read performance of the write-optimized policy, but not significantly. To experiment with workloads that may create more fragmentation, we defined a new traversal: the *random* traversal. This traversal also follows a random path down the assembly hierarchy to a composite part and then performs a depth-first search of the composite part. As part of this depth-first search, each atomic part visited by the search is modified with probability p. The read and write traversals defined in [8] are equivalent to random traversals with p set to 0 and 1 respectively.

This thesis uses two kinds of client workloads constructed out of read, write, and random traversals. Both workloads are characterized by a write probability w. In the *clustered-write workload*, the client executes a mixture of read and write traversals. The ith traversal is a write traversal with probability w (and a read traversal otherwise.) The clustered-write workload never executes a random traversal that modifies only some of the atomic parts from a composite part and therefore the contents of a composite part do not fragment significantly even under the write-optimized workload.

In the *random-write workload* all traversals are random traversals and the probability of modifying an atomic part in the depth-first search of the composite part is w. On average a random-write workload will generate as many modifications per traversal as the clustered-write workload, but these modifications will not be as well organized as the modifications under the clustered-write workload. Therefore the random-write workload will create a significant amount of fragmentation under the write-optimized workload.

The definitions of the clustered-write and the random-write workloads control the fine-grain characteristics of the client behavior. As described earlier, the client directs 90% of its traversals to its private module, and the remaining 10% of its traversals to the 19 modules that make up the rest of the database. Also, each client traversal is executed in a separate transaction.

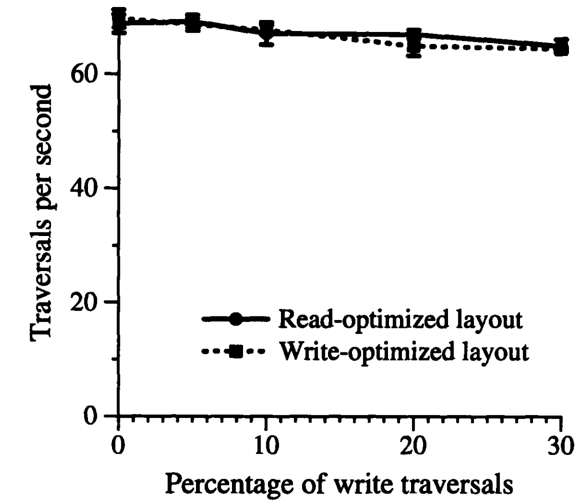


Figure 6-3: Performance of the two disk-layout policies as the percentage of write traversals in the clustered-write workload is varied.

6.3.6 Clustered-Write Experiments

Figure 6-3 shows the results of running the clustered-write workload at write probabilities ranging from 0% to 30%. The performance of the two policies is very similar in all cases. This should not be surprising because each write traversal modifies all atomic parts in a composite part. Under the write-optimized policy the contents of a composite part split up into exactly two fragments: one fragment contains all of the atomic parts and the other fragment contains the other pieces of the composite part. Under the read-optimized policy, a traversal of a composite part that is not present in the client cache requires one disk read, whereas under the write-optimized policy the same traversal requires two disk reads. The cost of the extra disk read in the write-optimized policy is offset by the better write performance of the write-optimized disk policy.

6.3.7 Random-Write Experiments

Figure 6-4 shows the results of running the random-write workload at write probabilities ranging from 0% to 30%. The two policies have identical performance when the client workload contains no modifications because the database creation code creates a fairly well-clustered disk layout under both disk layout policies. However as we increase the write probability, the write-optimized disk layout policy writes out just the modified objects and therefore destroys this good clustering. Therefore the performance of the write-optimized policy quickly drops below that of the read-optimized policy.

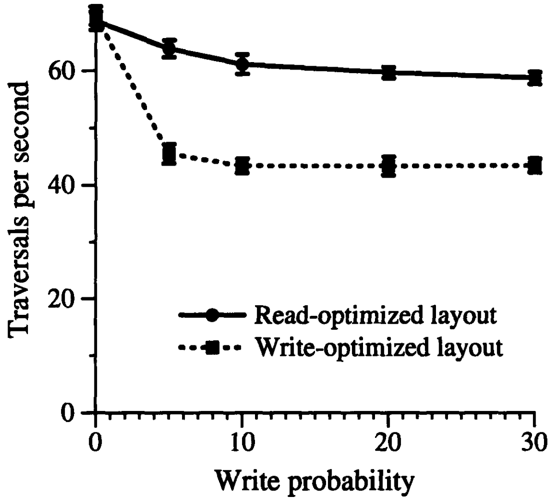


Figure 6-4: Performance of the two disk-layout policies as the percentage of modifications in the random-write workload is varied.

6.3.8 Scalability Implications

The preceding experiments were for a single client system where the server was given a very limited amount of memory. This section discusses how system performance will change as we increase the number of clients and the total amount of server memory. There are two primary issues to consider: the expected server cache hit rate, and the amount of write absorption in the MOB.

To make things concrete, let us consider a system in which the client workloads have been fixed to use a 20% write probability, the number of clients has been increased to 10, and available server memory has been scaled up by a factor of ten to 25 megabytes. Under the write-optimized policy 5 megabytes are used for the MOB, and under the read-optimized policy 20 megabytes are used for the MOB.

According to our workload definition client i directs 90% of its traversals to module i. The remaining 10% of the traversals are spread out uniformly over the other 19 modules in the database. The client cache is big enough to satisfy the hot traversals without requiring fetches from the server. However the cold traversals will all generate fetch requests to server. When we combine the fetch requests from all ten clients, the resulting fetch pattern will be almost uniformly spread out over all 20 modules. The fetch pattern will not be perfectly uniform because all 10 clients will direct cold accesses to the 10 modules that are not hot for any of the clients whereas client i will not direct any cold accesses to module i. The first 10 modules will receive only 9/10th as many fetch requests as the last 10 modules. However this unevenness in the fetch pattern is minor and we will ignore it in our analysis. The overall read pattern in the system is that 10% of the client traversals will require fetches from the server cache and furthermore all of these 10% traversals are spread out uniformly over the entire database.

Therefore as we increase the amount of memory assigned to the server cache, read performance will not improve significantly unless we make the cache almost as big as the entire database.

We can now compute the expected performance of the two disk layout policies with ten clients. We achieve this by counting the total number of disk operations required to execute 100 client traversals. These traversals will touch 10 cold composite parts on average because 90% of each client's traversals are directed to its hot module. Because the read-optimized policy uses a very small server cache, most of these cold composite part traversals will require a disk read. On average the read-optimized policy will require 10 disk reads every 100 traversals. Under the write-optimized policy the amount of read traffic will vary depending on the amount of fragmentation. Let us assume that each composite part has split into f fragments. The server cache size is 20 megabytes and therefore 20% of the fragment reads will hit in the server cache: for every 100 client traversals, 90 traversals will hit in the client cache and of the remaining 10 traversals two will hit in the server cache and therefore there will be $8f$ disk reads generated under the write-optimized policy.

Now let us consider the write pattern generated by the ten clients. 90% of client i's modifications will be directed to module i. Of the remaining 10%, almost half will be directed to one of the first 10 modules and the other half will be directed to the last 10 modules that do not belong to any client in particular. All modifications are flushed to the server and therefore when we consider the stream of modifications arriving at the server, 95% of the modifications will be directed to the first 10 modules and the remaining 5% to the rest of the database. The combined size of the atomic parts in each module is approximately 800 kilobytes. (Each atomic part is 80 bytes, there are 20 atomic parts per composite part, and there are 500 composite parts per module.) Therefore all of the atomic parts in the first 10 modules will occupy 8 megabytes. Under the read-optimized policy all of these atomic parts can easily fit in the 20 megabyte MOB and we assume that the 95% of the modifications that are directed to the first 10 modules are all absorbed in the MOB and do not generate any disk writes. We assume that the remaining 5% of the modifications all require disk writes. Suppose the client workload generates updates to w composite parts for every 100 traversals. Using the analysis given above 95% of these updates will be absorbed in the 20 megabyte MOB used for the read-optimized disk layout policy. Therefore every 100 traversals will require $0.05w$ installation reads and $0.05w$ disk reads. Since the write-optimized policy writes out modifications very efficiently, let us assume that its write cost is negligible.

Combining these formulas we get that for 100 client traversals the read-optimized policy will generate $10 + 0.1w$ disk operations (ten disk reads, $0.05w$ installation reads and $0.05w$ disk writes.) The write-optimized policy will generate $8f$ disk reads.

For the clustered-write workload with a 20% write probability, 20 composite parts are modified for every 100 traversals ($w = 20$). Under the write-optimized disk layout policy each composite part splits into two fragments and therefore $f = 2$. Therefore for 100 client traversals, the read-optimized policy will generate 12 disk operations ($10 + 0.1 \times 20$). The write-optimized disk layout policy will generate 16 disk operations (8×2). The read-optimized policy will provide better performance than the write-optimized policy, but the performance difference will not be very large, particularly when we take into account the fact that each disk read generated by the write-optimized policy only reads a fraction of a page.

For the random-write workload with a 20% write probability, each client traversal will modify 4 out of 20 atomic parts on average. Therefore every client traversal is highly likely to modify at least one of the atomic parts in the traversed page ($w = 100$). Under the write-optimized disk layout policy the MOB is small enough that we would not expect it to improve the clustering of modifications to cold modules. We assume that each cold composite part fragments into 5 parts under the write-optimized policy. For 100 client traversals, the read-optimized policy will generate 20 disk operations ($10+0.01 \times 100$). The write-optimized policy will generate 40 disk operations (8×5). Therefore the read-optimized policy will provide much better performance than the write-optimized policy under the random-write workload.

6.4 Summary

This chapter describes the implementation of the modified object buffer in the Thor object-oriented database system and presents performance results of running OO7 benchmark traversals on this implementation. The performance results indicate that a MOB significantly improves the write performance of a read-optimized disk layout policy. Furthermore, the performance results and an analysis show that the combination of a MOB and a read-optimized disk layout provides better overall performance than a write-optimized disk layout that does not preserve the clustering of related objects on disk.

The performance differences between the two disk layout policies are highly dependent on the write patterns generated by the clients. If the clients update whole groups of related objects at a time, then the write-optimized policy does not reduce on-disk clustering significantly and the two disk layout policies provide similar performance. If the clients update objects in a non-clustered fashion, then the write-optimized policy substantially reduces clustering and performs much worse than the read-optimized policy. These results indicate that the disk layout policy should preserve the on-disk clustering of related objects. A modified object buffer allows the Thor server to use a disk layout policy that preserves clustering under all application access patterns without sacrificing write performance.

Chapter 7

Related Work

There has been a lot of previous research on storage management techniques that attempt to reduce the impact of the speed mismatch between disks and memory. This chapter provides an overview of some of the important trends in this research and describes how the modified object buffer fits into the class of storage management techniques used in other systems.

7.1 Relational Databases

Storage management techniques for relational databases are very well understood. These databases typically maintain a large page cache similar to the one examined in Chapter 5 [25]. Entire pages are brought into the cache to satisfy read requests. Modifications are installed into the page cache. Dirty pages are written out to disk when they are evicted from the cache to make room for other pages. Pages are also written out by a periodic check-pointing process that purges large number of dirty pages from the page cache and deletes the corresponding information from the stable transaction log. The PAGE organization used in many object-oriented databases is a direct descendant of this organization.

Clustering is straightforward in relational databases because the database system has fairly accurate information about typical access patterns: tuples in a relation are typically scanned sequentially, or in order based on some indexed fields. Therefore such systems can achieve extremely good clustering by ordering and grouping tuples on an appropriate set of pages.

7.2 Object-Oriented Databases

Most distributed object-oriented databases use the PAGE organization described in Chapter 5 [16, 34, 10, 31]. As the results of this thesis show, the PAGE organization provides poor performance under typical object-oriented access patterns. The modified object buffer organization proposed in this thesis will provide a much better storage organizations for these object-oriented databases.

The RVM storage manager is not an object-oriented database, but it has similar access characteristics because it allows applications to modify individual byte ranges of persistent data [50]. The RVM storage manager uses a write-ahead transaction log to hold new values of modified objects. RVM is built on the assumption that the entire database fits in memory, and therefore these modifications are also installed immediately into the in-memory copies of the appropriate database pages. The dirty pages are periodically written out to disk to allow the on-disk transaction log to be truncated. This scheme allows the absorption of many modifications to a page before the page is written out to disk, but because of its main-memory requirements, it is only feasible to use RVM for fairly small databases. The MOB architecture is designed for databases that are too big to store completely in memory.

Disk reads for handling partial block modifications (*installation reads*) in a system like Thor were studied by Shrira and O'Toole in [45]. Their study uses simulations to evaluate the effectiveness of clever scheduling techniques that reduce the impact of installation reads on server performance. These techniques depend on very good knowledge about the disk layout and performance characteristics. Ruemmler's and Wilkes' description of disk characteristics and technology trends indicates that it will be very hard for software built above the disk interface to reliably discover and exploit this disk-specific information [49]. Therefore the scheduling techniques described in [45] will work reliably only if they are implemented inside the disk interface itself. Since the disk interface is block-oriented, such an implementation will not help with partial block modifications. The implementation of Thor uses a much simpler disk scheduler and depends mostly on the write absorption benefits provided by the modified object buffer.

Shrira and O'Toole's study does not obtain as much write absorption as we do. There are several reasons for this difference between the two studies. First, in their workload the write pattern arriving at the server is completely uniform over the entire database. Second, each one of their transactions modifies half of a page. Our transactions modify much smaller fractions of a page and therefore the MOB in our system can hold the modifications for many more transactions. Third, their system schedules operations out of the entire MOB. We schedule modifications only from the tail of the MOB and this leads to better write absorption as predicted by the analysis in Section 4.3.2.

7.3 File Systems

File systems store two kinds of data with very different access patterns: file data and metadata. The contents of a file tend to be read and written sequentially in their entirety [4]. Therefore it is easy to cluster the contents of a file efficiently by storing the file sequentially on disk [32, 41, 42]. Metadata information such as inodes and directory entries on the other hand are somewhat similar to objects in object-oriented databases: metadata items are typically small and therefore many such items are stored on each page. Most file systems do not have very good clustering policies for metadata, and therefore often just a single item on a page will be read or modified within a short period of time. For example, the BSD fast file system's clustering policies for inodes attempts to place the inodes for files from the same directory on the same cylinder of the disk [41]. This clustering policy reduces seek delays, but is much less efficient than a policy that places these inodes on the same page. Part of the reason file systems do not pay much attention to clustering of metadata is because metadata forms only a small part of the total file system data, and therefore with a reasonably sized metadata cache, most metadata accesses will not require disk reads [47].

File system implementations improve write performance for metadata either by delaying metadata writes [12, 28, 21, 33], or by using a write-optimized disk layout [48, 15, 30]. A write-optimized disk layout works well for file systems for two reasons:

- File contents remain clustered because most files are read and written sequentially.
- Metadata is not clustered, but the amount of metadata is small enough to fit in a reasonably sized cache.

Therefore for file system workloads, a write-optimized disk layout provides as good read performance as a read-optimized disk layout.

File systems differ in their reliability guarantees, and therefore some file systems that use delayed metadata updates use a stable log for reliability whereas other file systems are less reliable. We will ignore the reliability properties of file systems and just focus on metadata write performance issues. Most file systems that use delayed writes use a PAGE organization for server memory. Therefore installation reads may be required before modifications can be installed into the cache. However since file systems can cache metadata very effectively, most updates will not require any installation reads. Therefore the PAGE architecture will probably work just as well as the MOB architecture in a file system.

The *soft updates* approach of Ganger and Patt uses a memory organization very similar to the MOB architecture [21, 22]. Instead of caching entire pages, part of the server memory holds individual modified metadata items. Unlike the MOB, this organization is not motivated by a desire to reduce memory usage. The soft updates approach does not use a stable log to ensure the recoverability of delayed writes. Therefore to maintain the consistency of the on-disk copy of the file system, the server has to carefully sequence its delayed writes. This sequencing is achieved by keeping dependency information between metadata updates. If metadata updates are grouped together into pages, then false dependencies are introduced because two unrelated metadata items may be stored on the same page. These false dependencies can create cycles in the dependency graph, and can also generate excessive disk writes. By storing modified metadata items separately instead of together in pages, the file system avoids introducing such false dependencies.

Many file systems use non-volatile memory to speed up synchronous meta data updates [3, 43, 30]. If NVRAM is available to an object-oriented database, it can store the modified object buffer in this NVRAM and therefore avoid the necessity of having a separate stable transaction log.

The Harp file system provides highly available storage for files [39]. It is implemented with a primary-backup replication scheme in which the primary builds up in main-memory a log of modifications and continually streams the contents of this log to the backup. The work described in thesis was motivated by the somewhat unexpected performance gains observed in Harp: the replicated file system provided better performance than the underlying un-replicated file system because modifications stored in this main-memory log were applied lazily to the disk instead of being applied to the disk right away. The Thor storage architecture is much more aggressive about exploiting delayed writes and therefore reaps extensive performance benefits. The benefits observed in Harp were mainly due to the reduction in commit latency because modifications just had to be sent to the backup over the network instead of being written out to disk.

Chapter 8

Conclusions

This thesis proposes and evaluates a new storage architecture for distributed object-oriented databases. The proposed architecture uses a large portion of server memory for a modified object buffer (MOB) that stores recently modified objects. The MOB provides excellent write performance because of write absorption: multiple modifications to the same page can be written to disk at the same time. The MOB can be used in conjunction with a read-optimized disk layout that preserves clustering and therefore provides good read performance. This thesis contains simulation studies of the MOB, and describes an implementation of the MOB as part of the Thor object-oriented database. This chapter summarizes the important results of this thesis and describes some future work.

8.1 Results

This thesis focuses on the performance impact of the MOB. Simulation results and an analysis of the MOB show that the MOB significantly improves the write performance of a read-optimized disk layout. This improvement in write performance is a result of the substantial write absorption provided by the MOB. For example, in a system where the size of the MOB is 10% of the database size, and each transaction modifies 10% of a page, the MOB reduces the total number of writes to one third of the number of writes that would be required without a MOB. The overall throughput of a system that uses a read-optimized disk layout improves by over 200% when a reasonably sized MOB is added to the server.

This thesis also shows that a write-optimized policy that does not preserve clustering has poor read performance. Because of the improvements in write performance from the MOB, the read-optimized policy provides much better overall performance than the write-optimized policy on a wide range of workloads and system configurations. The write-optimized disk layout provides better performance only under configurations in which read performance becomes irrelevant because clients have enough memory to cache the contents of the entire database. Most systems will not be configured this way because memory is currently over two orders of magnitude more expensive than disk storage.

The thesis also compares the MOB architecture to the PAGE architecture that is used in most databases and file systems. Other research has shown that PAGE performs poorly in an object-shipping system [45, 56]. The results of this thesis show that even in a page-shipping system, MOB provides better overall performance than PAGE under typical object database access patterns. For example, in workloads with a write probability of 20% or less, the MOB provides better throughput than PAGE under almost all clustering patterns.

Under certain access patterns that are likely to be more common in file systems than in object databases, PAGE provides better performance than MOB. This thesis suggests some simple optimizations that can significantly improve the performance of MOB under such workloads.

The overall conclusion of this thesis is that the modified object buffer substantially improves the write performance of an object-oriented database. Furthermore, using a read-optimized disk layout in combination with a MOB provides good performance under a wide variety of workloads and system configurations. This combination has good read performance because most accesses hit in the client caches, and the accesses that miss in the client caches can be satisfied with an efficient disk read of an entire set of clustered objects. The organization also has good write performance because of the write absorption benefits of the MOB.

8.2 Future Work

This thesis raises some important questions. The choice between object-shipping and page-shipping has significant ramifications for an object-oriented database system:

- Client caches may be organized as object caches because clustering may often be sub-optimal for a given application's access pattern. An object cache allows the client to discard the unused portion of each page. Page-shipping will not work in such systems because the client may have discarded part of a page that has to be shipped back to the server.

- Shipping entire pages between the client and server may waste scarce network bandwidth if only small portions of each page are being modified.

- Applications can have better structure and are easier to write if the database provides concurrency control at the level of individual objects. Object-level concurrency control is simpler to implement in an object-shipping architecture: in a page-shipping architecture the server may have to merge the contents of several incoming dirty pages to create a version that contains the most up-to-date versions of all of the objects that belong to that page.

- Installation reads can be avoided in a page-shipping system. If the server has a PAGE architecture, then installation reads are obviously not necessary. If the server has a MOB architecture, then the server can fetch the old contents of page that has to be written out to disk from a client cache instead of the disk.

- Page-shipping systems may have lower CPU overheads because the clients and the server can manage storage in terms of a small number of large pages instead of a large number of small objects.

Shrira and O'Toole have a preliminary study that examines the tradeoff between client cache performance and the cost of installation reads [45]. A more complete study that takes all of the preceding factors into account will be very beneficial to system designers.

The characteristics of object-oriented database applications reported by Chang and Katz in [11] were very helpful in devising some of the policies and client workloads used in this thesis. Reports from more real world applications would provide more guidance for such decisions. Other designers and implementors of object-oriented databases will also benefit from more information about object database applications. Therefore a study of more applications will be very valuable.

Various optimizations may be added to the basic design of the MOB to further increase its performance benefits. One of the performance problems of the current MOB design was noticed in a comparison of the MOB architecture to the PAGE architecture under a workload where the clients were modifying an entire page full of objects at a time. MOB had worse performance than PAGE because PAGE was maintaining its cache using an LRU replacement policy whereas MOB uses a FIFO eviction policy. Under a hot-cold workload the LRU policy will provide better write absorption than a FIFO policy. Therefore, it might improve performance to change the eviction policy used in the MOB. The benefit of the FIFO policy is that it writes out modified objects in log order and therefore simplifies the process of log truncation. If we switch to another policy, we will have to implement a more complicated scheme to perform log truncation before log space fills up. Some of the check-pointing algorithms used in databases can be adapted for this purpose [26].

Installation reads form a significant component of the overall disk utilization in the MOB architecture. Some of these installation reads can be avoided by using a cooperative caching scheme that allows servers to fetch pages from client caches [20]. For example, in the simulated workload defined in Chapter 4, installation reads for hot and warm regions of the database can always be avoided by just fetching the appropriate page from one of the client caches. Installation reads for a cold page may also be avoided if the client that generated modifications to the cold page has not discarded the contents of the cold page by the time the page is written out by the server. It will be useful to study the performance impact of cooperative caching as a mechanism for reducing the number of installation reads.

One potential performance problem of the MOB architecture is that an organization that devotes most of server memory to the MOB and very little to the cache will perform sub-optimally under a completely read-only workload. It should be possible to devise an adaptive scheme that dynamically adjusts the division of memory between the cache and the MOB based on current access patterns. Such an adaptive scheme might be able to improve the read performance of the system under read-heavy workloads.

The MOB implementation can waste memory because it uses a standard memory allocator to allocate space for modified objects. These allocators typically waste space proportional to the number of allocated objects. In workloads where most of a page has been modified, memory usage may be decreased by storing the entire page in memory instead of storing each object individually. Frequently compacting the contents of the MOB by rearranging objects in memory may also decrease memory requirements. It will be useful to experiment with memory management policies that can reduce the amount of space wasted by the MOB implementation.

Bibliography

[1] A. Adya. Transaction management for mobile objects using optimistic concurrency control. Technical Report MIT/LCS/TR-626, MIT Laboratory for Computer Science, 1994.

[2] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, 1995.

[3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast reliable file systems. In *Architectural Support for Programming Languages and Operating Systems*, pages 10–22, Boston, Massachusetts, 1992.

[4] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *13th Symposium on Operating System Principles*, pages 198–212, Pacific Grove, CA, 1991.

[5] V. Benzaken and C. Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. Technical Report 50-90, Altair, 1990.

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

[7] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Winter Usenix Technical Conference*, pages 277–288, New Orleans, LA, 1995.

[8] M. Carey, D. DeWitt, C. Kant, and J. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *OOPSLA Proceedings*, pages 414–426, Portland, OR, 1994.

[9] M. Carey, D. DeWitt, and J. Naughton. The OO7 benchmark. In *ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, DC, 1993.

[10] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS database system. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 91–100, Kyoto, Japan, 1986.

[11] E. Chang and R. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. In *ACM SIGMOD International Conference on Management of Data*, pages 348–357, Portland, OR, 1989.

[12] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Winter Usenix Technical Conference*, pages 43–60, San Francisco, CA, 1992. USENIX.

[13] M. Day. *Client cache management in a distributed object database*. PhD thesis, Massachusetts Institute of Technology, 1995.

[14] M. Day, B. Liskov, U. Maheshwari, and A. Myers. References to remote mobile objects in Thor. In *ACM Letters on Programming Languages and Systems*, pages 115–126, 1994.

[15] W. deJonge, F. Kaashoek, and W. Hsieh. Logical disk: A simple new approach for improving file system performance. In *14th Symposium on Operating System Principles*, pages 15–28, Asheville, NC, 1993.

[16] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[17] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *ACM SIGMOD International Conference on Management of Data*, pages 1–8, Boston, MA, 1984.

[18] P. Drew and R. King. The performance and utility of the Cactis implementation algorithms. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 135–147, Brisbane, Australia, 1990.

[19] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notion of consistency and predicate locks in a database system. *CACM*, 19(11):624–633, 1976.

[20] M. Franklin, M. Carey, , and M. Livny. Global memory management in client-server DBMS architectures. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 596–609, Vancouver, Canada, 1992.

[21] G. Ganger and Y. Patt. Metadata update performance in file systems. In *Usenix Symposium on Operating System Design and Implementation*, pages 49–60, Monterey, CA, 1994.

[22] G. Ganger and Y. Patt. Soft updates: A solution to the metadata update problem in file systems. Technical Report CSE-TR-254-95, University of Michigan, 1995.

[23] J. Gray. Notes on database operating systems. In *Operating Systems – An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[24] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G. M. Nijssen, editor, *Modeling in Data Base Management Systems*, pages 365–394. North-Holland, Amsterdam, 1976.

[25] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzoli, and I. Traiger. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–243, 1981.

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

[27] R. Gruber. *Concurrency control mechanisms for client-server object-oriented databases*. PhD thesis, Massachusetts Institute of Technology, forthcoming.

[28] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *11th Symposium on Operating System Principles*, pages 155–162, Austin, TX, 1987.

[29] Hewlett-Packard. HP C2240 series 3.5-inch SCSI-2 disk drive: technical reference manual. Part number 5960–8346, edition 2, Hewlett-Packard, 1992.

[30] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Winter Usenix Technical Conference*, pages 235–246, San Francisco, CA, 1994.

[31] M. Hornick and S. Zdonik. A shared segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):71–95, 1987.

[32] IBM. *MVS/XA JCL User's Guide*. International Business Machines Corporation.

[33] M. Kazar, B. Leverett, O. Andersen, A. Vasilis, B. Bottos, S. Chutani, C. Everhart, A. Mason, S. Tu, and E. Zayas. DECorum file system architecture overview. In *Summer Usenix Technical Conference*, pages 151–164, Anaheim, CA, 1990.

[34] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, 1990.

[35] E. K. Lee. Software and performance issues in the implementation of a RAID prototype. Technical Report UCB/CSD 90/573, University of California at Berkeley, 1990.

[36] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, 1989.

[37] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers. Theta reference manual. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, 1995.

[38] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In M. T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 79–91. Morgan Kaufmann, 1993.

[39] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *13th Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, 1991.

[40] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Massachusetts, 1987.

[41] M. K. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181, 1984.

[42] L. McVoy and S. Keiman. Extent-like performance from a UNIX file system. In *Winter Usenix Technical Conference*, pages 33–43, Dallas, TX, 1991.

[43] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, and B. Lyon. Breaking through the NFS performance barrier. In *Proceedings of EUUG*, pages 199–206, Munich, Germany, 1990.

[44] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems -or- your cache ain't nuthin' but trash. In *Winter Usenix Technical Conference*, pages 305–314, San Francisco, CA, 1992. USENIX.

[45] J. O'Toole and L. Shrira. Opportunistic log: Efficient installation reads in a reliable storage server. In *Usenix Symposium on Operating System Design and Implementation*, pages 39–48, Monterey, CA, 1994.

[46] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM Operating Systems Review*, 23(1):11–28, 1989.

[47] K. J. Richardson and M. J. Flynn. Attribute caches. Technical note TN-48, Digital Equipment Corporation, Western Research Laboratory, 1995.

[48] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *13th Symposium on Operating System Principles*, pages 1–15, Pacific Grove, CA, 1991.

[49] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[50] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere, and J. Kistler. Lightweight recoverable virtual memory. In *14th Symposium on Operating System Principles*, pages 146–160, Asheville, NC, 1993.

[51] M. Seltzer, K. Bostic, and M. K. McKusick. An implementation of a log-structured file system for UNIX. In *Winter Usenix Technical Conference*, pages 307–326, San Diego, CA, 1993.

[52] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Winter Usenix Technical Conference*, pages 249–264, New Orleans, LA, 1995.

[53] J. W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, 1984.

[54] M. M. Tsangaris and J. F. Naughton. A stochastic approach to clustering. In *ACM SIGMOD International Conference on Management of Data*, pages 12–21, Denver, Colorado, 1991.

[55] M. M. Tsangaris and J. F. Naughton. On the performance of object clustering techniques. In *ACM SIGMOD International Conference on Management of Data*, pages 144–153, California, 1992.

[56] S. J. White and D. J. DeWitt. Implementing crash recovery in Quickstore: A performance study. In *ACM SIGMOD International Conference on Management of Data*, pages 187–198, San Jose, CA, 1995.