LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT-LCS-TR-708

OPTIMISM VS. LOCKING:
A STUDY OF CONCURRENCY CONTROL FOR CLIENT-SERVER OBJECT-ORIENTED DATABASES

Robert E. Gruber

February 1997

545 TECHNOLOGY SQUARE; CAMBRIDGE, MASSACHUSETTS 02139

Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases

Robert E. Gruber

S.B. Massachusetts Institute of Technology (1989)
S.M. Massachusetts Institute of Technology (1989)

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

February 1997

Author ....
Department of Electrical Engineering and Computer Science
January 30, 1997

Certified by ....
Barbara H. Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases

Robert E. Gruber

Abstract

Many client-server object-oriented database systems (OODBs) run applications at clients and perform all accesses on cached copies of database objects. Moving both data and computation to the clients can improve response time, throughput, and scalability. For applications with good locality of reference, retaining cached state across transaction boundaries can result in further performance and scaling benefits.

This thesis examines the question of what concurrency control scheme is best able to realize these potential benefits. It describes a new optimistic concurrency control scheme called AOCC (Adaptive Optimistic Concurrency Control) and compares its performance with that of ACBL (Adaptive-Granularity Callback Locking), the scheme shown to have the best performance in previous studies. Like all optimistic schemes, AOCC synchronizes transactions at the commit point, aborting transactions when synchronization fails; ACBL, like other locking schemes, synchronizes transactions while they execute.

Earlier studies concluded that locking is a better choice than optimism for client-server systems. Our research leads to the opposite conclusion. We present the results of simulation experiments showing that AOCC outperforms ACBL across a wide range of system and workload settings. In addition to having better performance, AOCC is simpler than ACBL, and scales better with respect to the number of clients using shared data.

The thesis also presents a model that allows us to understand these results. The two major costs associated with locking are the messages that must be exchanged for clients to acquire locks and the cost of waiting for locks held by other clients. The major cost associated with optimism is the need to re-execute aborted transactions. The reason AOCC performs so well (and that previously-studied optimistic schemes did not do well) is that its design allows it to have very low abort and restart costs. Both the model and the insights gained from the performance study should help in the design of future client-server concurrency control schemes.

Keywords: Concurrency Control, Atomic Transactions, Optimism, Pessimism, Locking, Object-Oriented, Client-Server, Databases, Simulation, Design Analysis

Thesis Supervisor: Barbara H. Liskov
Title: Ford Professor of Engineering

Acknowledgments

First, I would like to thank my advisor, Barbara Liskov. I have learned much more under her guidance than how to do good research. I am grateful for her counsel and support throughout my graduate career.

Many people have provided useful feedback and encouragement for this thesis work, including Bill Weihl, Frans Kaashoek, John Guttag, Sanjay Ghemawat, and Atul Adya.

Of the many friends who deserve credit for making my life interesting, I owe special thanks to the three amigos who were with me from the start: Sanjay Ghemawat, Wilson Hsieh, and Anthony Joseph. My life at MIT was also enriched by my time spent as a floor tutor on Conner Three, as a "staffer" at NL, as a member of the Programming Methodology Group, and as an avid participant in Tuesday Night Hockey.

Finally, I would like that thank all of the members of my family for their love and support. This thesis is dedicated to them.

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.

Contents

1 Introduction **13**

1.1 System Model 14
1.2 Database and Workload Assumptions 15
1.3 Contributions 17
1.4 Thesis Overview 18

2 Optimistic and Locking Designs **21**

2.1 Common Assumptions 22
2.2 AOCC 24
2.2.1 Detailed Description 26
2.2.2 AOCC Discussion 28
2.3 ACBL 28
2.3.1 Detailed Description 31
2.3.2 ACBL Discussion 35
2.4 Design Discussion 37
2.4.1 Undo log 37
2.4.2 Adaptive Cache Maintenance 38
2.4.3 Adaptive Directory Information 39
2.5 Design Comparison 41
2.5.1 Low Contention 41
2.5.2 High Contention 44
2.5.3 Read-Only Transactions 47

3 Related Work **49**

3.1 Franklin and Carey 50
3.1.1 C2PL vs. CBR vs. CBA 51
3.1.2 O2PL vs. CBR 51
3.1.3 Invalidate vs. Propagate 53
3.1.4 O2PL vs. AOCC 53
3.1.5 Cache Maintenance Strategies 54
3.2 Wang and Rowe 55
3.2.1 C2PL vs. CBA 55
3.2.2 NWL vs. C2PL 56
3.2.3 NWL vs. NWL-Notify 56
3.2.4 C2PL vs. Certification 57

3.3 Wilkinson and Neimat 57
3.3.1 Cache Locks vs. Notify Locks vs. AOCC 58
3.4 Parallel Database Concurrency Control 59

4 Experimental Framework 61
4.1 System Model and Settings 62
4.1.1 Database 62
4.1.2 Processors 64
4.1.3 Server 64
4.1.4 Client 65
4.1.5 Network 65
4.1.6 Disk 66
4.2 Workload Model and Settings 67
4.2.1 Workload Model 67
4.2.2 Motivating the Six Workloads 72
4.2.3 Workload Descriptions 73
4.2.4 Summary of Workload Generator Settings 81
4.3 Related Experimental Framework 82

5 Main Experimental Results 85
5.1 Interpreting the Simulation Results 85
5.2 PRIVATE Workload 89
5.3 HOTCOLD Workload 91
5.4 SMALL+HOTCOLD Workload 98
5.5 UNIFORM Workload 102
5.6 HICON Workload 109
5.7 TINY+PRIVATE Workload 113
5.8 Summary of Key Insights 117

6 Sensitivity Analysis 121
6.1 System Model Experiments 121
6.1.1 Core System Parameters 122
6.1.2 Future System Results 125
6.2 Workload Model Experiments 128
6.2.1 Write Probability, Write Clustering 128
6.2.2 Read-Only/Read-Write Mix 129
6.2.3 Restart Behavior 131
6.2.4 Transaction Length 133
6.2.5 Compute-Intensive Applications 134
6.3 ACBL Wins Region 135
6.4 Summary 138

7 Conclusions 147
7.1 Results and Insights 147
7.2 Future Work 149

7.2.1 Starving Transactions 149
7.2.2 Read-Only Transactions 150
7.2.3 New Studies 151

A On System Parameter Settings **153**
A.1 Fixed Settings 153
A.2 Network Settings 155
A.2.1 Network Bandwidth 156
A.2.2 CPU costs 156
A.3 Disk Settings 160
A.3.1 Choosing the Disk Bandwidths 160
A.3.2 Choosing the Disks Per Server 163

List of Figures

2-1 Low Contention Message Count Model 42

3-1 The Nine Concurrency Control Schemes Discussed 50
3-2 Cache Maintenance Strategies 54

4-1 Summary of System Parameter Settings 63
4-2 Workload Parameters 68
4-3 UNIFORM Workload 75
4-4 PRIVATE Workload 76
4-5 HOTCOLD Workload 77
4-6 SMALL+HOTCOLD Workload 78
4-7 HICON Workload 79
4-8 TINY+PRIVATE Workload 80
4-9 Parameter Settings: UNIFORM, HICON, PRIVATE, TINY+PRIVATE 81
4-10 Parameter Settings: HOTCOLD, SMALL+HOTCOLD 81

5-1 PRIVATE: Throughput, % Improvement, Message Count 89
5-2 PRIVATE: Server and Disk Utilization 91
5-3 HOTCOLD: Throughput, % Improvement, Message Count 92
5-4 HOTCOLD: Blocks, Aborts, Accesses (per Commit) 93
5-5 HOTCOLD: Message Breakdown 94
5-6 HOTCOLD: Server and Disk Utilization 97
5-7 SMALL+HOTCOLD: Throughput, % Improvement, Message Count . 98
5-8 SMALL+HOTCOLD: Message Breakdown 99
5-9 SMALL+HOTCOLD: Blocks, Aborts, Accesses (per Commit) 101
5-10 SMALL+HOTCOLD: Server and Disk Utilization 101
5-11 UNIFORM: Throughput, % Improvement, Message Count 102
5-12 UNIFORM: Message Breakdown 104
5-13 UNIFORM: Blocks, Aborts, Accesses (per Commit) 105
5-14 UNIFORM: Lock Waiting and Wasted Work 107
5-15 UNIFORM: Server and Disk Utilization 108
5-16 HICON: Throughput, % Improvement, Message Count 110
5-17 HICON: Message Breakdown 111
5-18 HICON: Blocks, Aborts, Accesses (per Commit) 112
5-19 HICON: Lock Waiting and Wasted Work 113
5-20 HICON: Server and Disk Utilization 114

5-21 TINY+PRIVATE: Throughput, % Improvement, Message Count . . . 114
5-22 TINY+PRIVATE: Message Breakdown 116
5-23 TINY+PRIVATE: Blocks, Aborts, Accesses (per Commit) 117
5-24 TINY+PRIVATE: Lock Waiting and Wasted Work 118
5-25 TINY+PRIVATE: Server and Disk Utilization 119

6-1 Core System Parameters: Summary of Experiments 123
6-2 Core System Parameter Experiments: Throughput Results 124
6-3 CURRENT and FUTURE Parameter Settings 126
6-4 CURRENT vs. FUTURE: Low-Contention Example (HOTCOLD) . . 140
6-5 CURRENT vs. FUTURE: High-Contention Example (HICON) . . . 141
6-6 Varying Write Probability, Write Clustering (HOTCOLD) 142
6-7 SMALL+HOTCOLD: Read-Only Mix Experiment 143
6-8 SMALL+HOTCOLD: Main Memory Server 143
6-9 Varying the Restart Change Probability 144
6-10 Varying the Average Transaction Length 144
6-11 Varying the Range of Transaction Sizes 145
6-12 SMALL+HOTCOLD: Think Time Experiment 145
6-13 UNIFORM: ACBL Wins Region 146

A-1 Size-Based Parameters 153
A-2 Misc. Client and Server Costs 154
A-3 Client and Server CPU Speeds 154
A-4 Network Parameters 155
A-5 Latency for 1 KB Message 155
A-6 Kernel-Based Messaging Costs 157
A-7 Reported Latencies: User-Level Messaging 159
A-8 Disk Parameters 160
A-9 Results of Intelligent Disk Scheduling 161
A-10 Disk Utilization vs. Number of Server Disks 164
A-11 "Masking Effect" of Disk Saturation 165

Chapter 1

Introduction

This thesis explores concurrency control for transactional databases used in a distributed environment. In particular, we study client-server object-oriented databases (OODBs). We focus on systems where servers provide the storage for a universe of shared persistent objects (persistent stores), while clients run application code that uses this object state.

For most client-server OODBs, including research prototypes such as ORION [37], EXODUS [26], SHORE [9], and Thor [43] and also commercially available systems like GemStone [6], O2 [16, 17], ObjectStore [40], Ontos [46], Objectivity [45], Statice [60], and Versant [52], object accesses are performed at clients on cached copies of the persistent objects. Moving data to the applications has two main advantages. First, work is offloaded from the servers to the clients, improving the scalability of the system and allowing it to take advantage of the CPU and memory resources available at clients. Second, many object accesses can be performed without contacting the servers, allowing efficient fine-grained interaction between the application and the OODB.

This thesis asks the following question: which concurrency control scheme should be used for client-server OODBs, optimism or locking? Under optimism, a transaction simply uses objects in the client cache; it "optimistically" assumes that the objects are consistent with the committed state of the database, and that its accesses will not conflict with the accesses of other concurrent transactions. These optimistic assumptions are checked at the commit point of the transaction. The only messages that are required are fetch requests (to access missing data) and commit requests. Locking, on the other hand, must also acquire the proper lock before using an object. Some locking schemes require messages to acquire both read and write locks, while others require them only for write locks. (Chapter 3 discusses specific examples of these two cases.) For workloads with read-write sharing, therefore, all locking schemes will require extra messages.

Intuitively, optimism would seem to be the preferred approach for low contention workloads, since it uses fewer messages. However, previous work does not support this intuition. There are two client-server studies we know of that compare optimism with locking. One study shows that locking is always better [57, 58]; the other shows that optimism is slightly better, but only under low contention [23, 24].

This thesis reverses these earlier results. It defines a new optimistic concurrency control scheme, Adaptive Optimistic Concurrecy Control (AOCC), and presents the results of detailed performance studies that compare AOCC to Adaptive-granularity CallBack Locking (ACBL). Conventional wisdom is that most OODB applications have low contention; ACBL is the best currently-known client-server locking scheme for low-contention workloads. Our experiments show that AOCC outperforms ACBL for these workloads, and also for a large majority of workload and systems settings. The thesis also provides an analysis of why our new scheme is superior, including a set of insights that should influence the design of future schemes.

In addition to performance advantages, there are other good reasons for choosing an optimistic concurrency control scheme rather than a locking scheme. Optimistic schemes are easier to implement, as we discuss in Chapter 2. Since optimistic schemes use fewer synchronous round-trip requests, they are less sensitive to an increase in latency; optimism scales better with respect to distance between client and server. Thus, while we use a local-area network model in this thesis, we believe AOCC's advantage over ACBL would be higher for a wide-area network. In addition, under optimism the execution of a transaction only depends on the client executing the transaction and the servers in the system, while locking can cause a transaction at one client to block on a transaction running at another client. Thus, optimism is more robust with respect to a high variance in the availability or performance properties of clients. (Similar arguments about the benefits of lock-free synchronization have been made at the multiprocessors and operating system level [31, 34].) Due to this client-independence property, optimistic schemes are more easily extended to support a "disconnected client" semantics [32].

The remainder of this chapter is organized as follows. Section 1.1 discusses our system model assumptions, while Section 1.2 discusses our database and workload assumptions. Section 1.3 then summarizes the contributions of the thesis in more detail. Section 1.4 describes the organization and contents of the remaining chapters.

1.1 System Model

For our purposes, a client-server OODB consists of a server and a set of clients connected by a network. The server stores database pages on a set of data disks. We assume that objects are small and that pages contain many objects. For example, in the OO7 benchmark [8, 7], a widely accepted object-oriented database benchmark, most objects are smaller than 100 bytes.

The clients share the objects stored at the server. Each client has a page cache that contains copies of some of the database pages. An application runs at each client, issuing a sequence of transactions. A transaction accesses objects in the client cache; if an object on page P is accessed and P is not cached, the client *fetches* P from the server and installs the resulting copy of P in its cache. The server also has a page cache. Fetch requests are serviced through this cache; server cache misses result in disk reads. (Both the client and server caches use an LRU page replacement policy.)

A transaction's modifications are performed on the cached copies of the modified objects. When a transaction is ready to commit, copies of these objects are included in a *commit request* sent from client to server. If the server allows the commit, it installs the updates sent in the commit request so that future fetches will return the updated object state. At the end of the commit, the client retains the cached pages in case they may be of use to the next transaction.

The server and clients together implement a concurrency control scheme that provides atomic serializable transactions. While the caches are page-based, conflict detection is performed at the object level. One result of this fine-grained concurrency control is that cached pages can have "holes" in them; objects within a cached page can be marked as "missing." Both schemes studied in the thesis use object-level marking.

1.2 Database and Workload Assumptions

In addition to assuming that the normal case for an OODB is to have multiple objects per database page, we assume that a clustering algorithm [3, 14, 19, 51, 54, 55] is used to produce a well-clustered database with respect to common access patterns. In other words, a transaction accessing page P is likely to access multiple P objects.

The database can be very large (much larger than the client or server caches). We assume the set of pages accessed by the full set of clients is larger than the server cache size; many page fetches result in disk accesses.

We assume the client cache is large enough to hold all pages touched by a *single* transaction. This assumption holds for many present-day systems and applications. As we discuss in the concluding chapter, Chapter 7, we believe current memory trends and recent work on more efficient cache management both suggest that this assumption is very likely to hold for future systems.

We expect transactions to be restartable. Although transaction abort is more likely to occur under optimism, aborts will happen no matter what the concurrency control scheme. We assume aborts cause the code of the aborted transaction to be re-executed; this code must written to work correctly if such a restart occurs. As discussed further below, a restart does not necessarily imply that the transaction repeats exactly the same access pattern.

We expect transactions to run for a relatively short time. Long-running transactions are a problem for any concurrency control scheme since they increase the probability of conflicts. They can be avoided by implementing application-specific coordination semantics. For example, a check-in/check-out model is used by many applications that support cooperative design. Short restartable transactions (one to check out a design and another to check it back in again) can be used to implement this model. Thus, the application uses the transactions provided by the OODB to implement a higher-level semantics, which is then used to do application-level concurrency control.

The loss of "user work" due to aborts is often cited as a reason not to use an optimistic scheme. However, for interactive applications users should observe a high-level semantics, not OODB transaction semantics: aborts at the OODB level can be masked from the user. The choice of optimism vs. locking for the OODB transactions is orthogonal to the choice of an appropriate application-level semantics.

To study the efficacy of concurrency control schemes, we must use workloads with different characteristics. Ideally, a scheme will work very well on the expected cases, and reasonably well on the unexpected cases. There are three workload questions to be addressed:

1. How do different transactions at the same client relate to each other?
2. How do transactions at different clients relate to each other?
3. How does a restarted transaction relate to the previous (aborted) transaction?

We expect that transactions at a given client overlap in their access patterns. This makes sense: one transaction sets the stage for the next. It is common for traditional applications to exhibit locality of reference, and we expect OODB applications do as well, including locality of reference across transaction boundaries. Therefore, we study schemes that do not discard the client cache contents at end of each transaction. Previous studies have shown that such inter-transaction caching provides performance benefits for workloads with inter-transaction locality [23, 24, 57, 58, 61]. We study workloads with both low and high inter-transaction locality of reference.

With respect to the interaction of transactions at different clients, we use three different sharing patterns (in different combinations) in our workloads. The patterns define different kinds of database regions. The simplest region is a uniformly shared region: all clients access the region with equal likelihood. We also use per-client regions, where each client "owns" a different region of the database. Such client regions can be private (only the owner accesses the region) or they can be "mostly private" (the owner frequently accesses the region, while other clients occasionally access the region). By varying the access probabilities and write probabilities for different regions, we produce six workloads with six different sharing patterns and six different contention levels. For each workload that has contention, the contention level also increases with additional clients. We expect the lower contention cases to be the most common; however it is important to "degrade gracefully" as contention increases.

Finally, we discuss the behavior of transaction restarts. When a new transaction is executed, we say this is a *first-run* execution. If this execution aborts, its effects are undone and a *restart* execution is performed. Restart assumptions are very important for studies that include optimistic schemes, as such schemes have a high abort rate relative to locking schemes. Most simulation studies use a "perfect restart" assumption: exactly the same access sequence is repeated during restart. Since a first-run execution will "preload" the client cache with the objects that it accesses, using perfect restarts can result in very low fetch costs. If a new access pattern is chosen for a restart, the fetch costs will be closer to the fetch costs for a first-run execution.

One contribution of this thesis is to study a range of restart behaviors. We use a "middle ground" that falls between perfect restarts and new-access restarts for most of our experiments, and we also perform experiments designed to examine the impact of restart behavior.

1.3 Contributions

This dissertation makes the following contributions to the field of client-server concurrency control:

1. We describe a new optimistic scheme, AOCC, that has both very low message cost under low contention and low restart costs when aborts occur (under higher contention).

2. We summarize previous client-server concurrency control studies, and explain why previously-studied optimistic and semi-optimistic schemes have very high restart costs.

3. We describe an existing locking scheme, adaptive callback locking (ACBL); ACBL has been shown to be the best locking design for low-contention workloads.

4. We give an analysis that contrasts the ACBL and AOCC designs. For low-contention workloads, we argue that message count will determine the relative performance of different concurrency control schemes. We present a simple message-count model for AOCC and ACBL that demonstrates AOCC's lower message costs in terms of both number of messages used and number of round-trip delays.

5. For high contention workloads, we give a model of transaction latency that breaks down costs into components that can be compared across optimistic and locking designs. We use this model to show that AOCC should normally outperform ACBL under high contention.

6. We present results and analysis from a simulation study. We examined AOCC and ACBL behavior across six different workloads with different sharing patterns, different degrees of inter-transaction locality, and different contention levels. We also describe sensitivity analysis experiments that show the impact of varying the system and workload parameters. The results of these experiments are consistent with the expected results derived from our low and high contention models.

7. Our experiments produced a number of insights into the optimistic/locking tradeoff:

   (a) As predicted by our model, message count determines relative low contention performance. AOCC outperforms ACBL, as expected.

   (b) AOCC's advantage over ACBL increases with increased update frequency and also with increased contention caused by increasing the number of clients; AOCC scales better with the number of clients. Moreover, AOCC is less sensitive than ACBL to the quality of *write clustering*: ACBL's performance can change significantly depending on whether objects that tend to be updated in the same transaction are clustered together on the server disks.

   (c) For high contention workloads, ACBL suffers from excessive blocking and from expensive aborts (due to deadlocks). While AOCC has a much higher abort rate, its aborts have relatively low cost; the "preloaded" client cache contents result in fast restart executions.

8. The sensitivity analysis experiments also demonstrate two cases where ACBL can outperform AOCC. ACBL performs better on read-only transactions, but only for a multi-server setting, and only when the latency of the read-only transaction execution is not significantly higher than the latency of a commit request. (For example, if a read-only transaction incurs latency due to disk reads, AOCC's use of an extra commit request has little impact on relative latency.) ACBL also performs better if: there is high contention; restarts do not repeat previous accesses and moreover tend to access state that was not recently accessed by earlier transactions; fetch costs for AOCC are much higher than fetch costs for ACBL due to the fact that AOCC has driven the server disks into saturation while ACBL has not. This latter case requires both atypical restart behavior and a "disk poor" server configuration; this case is unlikely to occur in practice.

Both the model of transaction latency components and the insights gained from our performance studies should prove useful for analyzing existing designs and for producing new concurrency control designs.

1.4 Thesis Overview

The thesis is organized as follows.

- Chapter 2 presents the details of AOCC and ACBL, the optimistic and locking schemes used in our comparison studies. It also presents a model of transaction latency that allows us to compare their expected behavior.

- Chapter 3 discusses related work and compares our work to earlier client-server concurrency control studies.

- Chapter 4 covers our experimental setup. It describes both our simulator and the six workloads that we use in our studies.

- Chapter 5 presents our main experimental results for each of our six workloads. These results are based on a set of default system parameter settings that are realistic for hardware in use today.

- Chapter 6 describes sensitivity analysis experiments. It explores what happens as we deviate from the assumptions used in the experiments presented in Chapter 5, and shows that the conclusions drawn from our main experimental results hold across a wide range of system and workload parameter settings. The experiments reported in this chapter also allow us to characterize the conditions required for locking to outperform optimism.

- Finally, Chapter 7 concludes the thesis by summarizing our contributions and discussing some areas of future work, including improvements to our optimistic design.

In addition, Appendix A provides a detailed description of how we chose the default and "future" system parameter settings used in our experiments.

Chapter 2

Optimistic and Locking Designs

This chapter describes two concurrency control schemes:

- AOCC, a new optimistic scheme that we designed, is a variant of the "OCC" scheme [1] we developed for use in the Thor OODB [43]. AOCC retains the key elements of the OCC design but uses a different underlying architecture and a different client cache maintenance policy; these changes were required to perform a meaningful comparison to the ACBL design.

- ACBL is an adaptive-granularity callback locking scheme that is based closely on the PS-AA scheme designed by Carey, Franklin, and Zaharioudakis [10]. PS-AA's design was inspired by the adaptive locking algorithm described by Joshi for the VAXCluster version of Rdb/VMS [35]. We use this scheme in our optimism vs. locking comparison study because previous client-server concurrency control studies suggest it is the best locking choice for a client-server OODB. Comparative performance studies by both Franklin and Carey [23, 24] and Wang and Rowe [57, 58] conclude that callback locking is the best locking approach for low-to-moderate contention workloads. These studies used page-level conflict detection for all of the schemes compared. A later study introduced adaptive-granularity callback locking and demonstrated that using adaptive conflict detection is better than using either purely page-level conflict detection or purely object-level conflict detection [10].

In addition to describing the two schemes, this chapter presents a comparative design analysis. We define a simple model of the cost of executing a transaction. For low contention workloads, the model shows that message cost predicts performance. A message-count model for low contention shows that AOCC has lower message cost for this case. For high contention workloads, all components in the transaction cost model are relevant, where some costs are only incurred by locking schemes and some are only incurred by optimistic schemes. We discuss expected behavior for AOCC and ACBL in terms of these costs.

The rest of this chapter is organized as follows. Section 2.1 describes assumptions common to both designs. Sections 2.2 and 2.3 describe the AOCC and ACBL designs, respectively. A key design goal was to produce AOCC and ACBL schemes that use similar or identical mechanisms wherever possible; this approach allows us to focus on the fundamental differences between optimistic and callback locking designs. Section 2.4 discusses details of the two designs that are related to this goal. Finally, Section 2.5 presents the comparative design analysis.

2.1 Common Assumptions

This section describes some common design assumptions that underlie both the AOCC and ACBL designs described in this chapter.

We assume a single-server system. Thus, multi-server issues, such as the use of a two-phase commit protocol, are ignored. Both schemes are easily extended to support the multiple-server case. In addition, we assume one active transaction per client. Thus, we often use "client C" as a shorthand for "the active transaction T running at client C." This approach simplifies the descriptions. Both schemes can easily be extended to support multiple transactions per client.

The underlying OODB system architecture is the same for both schemes, to allow a direct simulation comparison of the two designs. We now describe the basic fetch, commit, and abort processing steps that are used by both schemes, including the client and server data structures used to carry out these steps. Mechanisms for concurrency control are omitted from this description.

The system is a hybrid of a page-based system and an object-based system. We assume that objects are small and do not span pages; pages contain many objects. The page-level mechanisms are as follows. The client and server both use page-based caches; these caches store pages and use a page-based LRU replacement strategy. When a miss occurs in a client cache, a fetch request is used to fetch one page. The server uses its page cache to handle fetch requests. If a requested page is not in the cache, it is read from disk into the server cache and then sent in a fetch reply.

There are four object-level mechanisms: objects can be discarded from client caches; clients can restore modified objects to their committed states if an abort occurs; object updates (rather than page updates) are sent to the server in commit requests; and the server uses a modified object buffer (or MOB) to efficiently store recently-committed object updates in main memory. The next two sections describe how these mechanisms are used at the client and server, respectively.

Common client architecture

A client can discard object X from cached page P by "marking" the object as missing. Once X is marked as missing, an access to X will cause the client to re-fetch page P, as a new version of object X must be brought into the cache. Such "object marks" are stored in the cache along with the page, and are replaced when the page is replaced. (A bitmap can be used to efficiently store the object marks for a page.)

When a client fetches a page from the server, the fetch reply contains both a new copy of the page and an initial set of object marks. (AOCC pages always begin with no marks, while ACBL pages sometimes begin with a set of initial marks. This distinction is made clear in the descriptions of the schemes.) If the page is already cached, the new page and marks over-write the existing page and marks. There is one exception for this over-write step: an object that has already been updated by an active transaction is not over-written. In other words, uncommitted updates are not replaced.

In the descriptions below, when we say that a client discards an object, this means the object is marked as missing. When we say a client discards a page, this means the page is dropped from the client cache along with any associated marks. When we say that a fetch reply is installed in the client cache, we mean that the new page and new marks are installed, possibly using the over-write step just described for the case where an older version of the fetched page is already in the cache.

The client architecture has a restore-on-abort mechanism that restores the committed state of objects that were modified by an aborted transaction. There are a number of ways to implement such a mechanism. We chose to use an object-based *undo log* to store unmodified copies of modified objects; object X is appended to this log prior to the first write of X, where X is then updated "in place" in the client cache. If an abort occurs, the object copies in the undo log are copied back to their pages to "undo" the aborted modifications. Undo logging at the client has very low space and time overheads. We discuss these costs in Section 2.4.1.

Common server architecture

When a read-write transaction reaches its commit point, the client sends a commit request to the server that includes object-level updates for each modified object. At the server, a transaction's updates are written to a reliable transaction log as part of commit processing. When the commit succeeds, these object updates are also added to a main-memory structure called the MOB (modified object buffer). The MOB holds main memory copies of all updated objects that are not yet safely installed on the data disks. The MOB is similar to a standard redo log, but is more compact: if object X has been updated many times, the MOB contains only the most recent committed state of X. (If main memory is lost, the MOB contents can be reconstructed from the transaction log.)

The server also has a page-based cache that is used for servicing client fetch requests. Fetching interacts with the server cache and MOB as follows. When a fetch for page P occurs, if P is not in the server cache, it is read from disk, and any P updates stored in the MOB are applied to the cached page. At this point P is up-to-date and is sent in a fetch reply. If P entries are added to the MOB while P is cached, they are not applied to the server cache immediately, since it is not clear that P will be fetched again before it is discarded from the server cache. As a result, when a fetch request finds that P is already cached, any MOB entry for P that has not yet been applied is now applied so that an up-to-date version of P is sent in the fetch reply. In the detailed descriptions of the schemes, we use "apply all necessary MOB entries for P objects" as a shorthand for the application of MOB entries just described.

A background thread at the server is responsible for "cleaning" the MOB so that its size does not grow without bound. This thread selects a set of pages to be updated based on various criteria (location on disk, time of last update, *etc.*). Some of these pages will not be in the server cache; they are read from disk as part of the update process. The necessary disk reads and writes are ordered by a disk scheduling algorithm that minimizes seek times at each server disk, resulting in a high effective disk utilization. Once page P's updates are safely on disk, they are discarded from the MOB.

Ghemawat's dissertation includes a comparison of a MOB-based design and the more standard approach of installing updates in the server cache at the time of commit [28]. This study shows that the MOB design has higher write absorption and makes more efficient use of the server disks for installations. In addition, message sizes are small relative to a scheme that sends page-level updates in commit requests. (The PS-AA study [10] uses page-level commit requests and a standard server cache. A "page-merge" operation is used for pages that are read-write shared, where a disk read is required if the page is not already cached.)

We use a MOB-based server architecture for both concurrency control schemes; both schemes benefit from the improved server performance. The use of a well-performing server is important, since it means any differences between concurrency control schemes are not being masked by other, unnecessary, overheads.

We now describe the designs of the two schemes. The schemes are described at an abstract level, in the sense that that data structures are chosen for clarity of exposition rather than for efficient implementation.

2.2 AOCC

AOCC allows a transaction to use objects in the client cache without any communication with the server. There are only two times that a client must communicate with the server, when a cache miss occurs, and at the commit point of a transaction.

While transaction T is running at client C, the client keeps track of the objects T reads and writes. Object modifications are performed "in place" on cached pages, and the client retains these pages; modified state is not sent to the server until T's commit point. When this point is reached, the client sends a description of T's reads and writes to the server along with the new states of objects that T modified. The server then validates T: it ensures T used only up-to-date versions of all objects it accessed. In other words, T passes validation if it can be serialized after all previously-committed transactions[1]. If this check succeeds, T commits, otherwise it aborts and restarts. The server sends the outcome to the client using a commit reply or an abort reply.

The basic optimistic design just described is good because it uses the minimum number of messages required to execute a transaction (for read-write transactions). For low contention workloads, this low message cost results in excellent performance.

[1]For a multi-server system, local validation at each server includes checks against other transactions that are also attempting to validate and commit. An efficient two-phase commit protocol for this case is described in [1].

However, for high contention workloads, if aborts are frequent and restart executions use as many messages as first-run executions, then the total cost per successful commit will be very high. It is easy to design an optimistic scheme that has good low-contention performance; providing good performance for high contention is more challenging.

AOCC is designed to reduce the probability of aborts. However, aborts are bound to occur under an optimistic scheme, thus AOCC is also designed to reduce the cost of aborts. Aborts can be detected early (prior to the end of the transaction), avoiding some unnecessary "wasted work." In addition, restarts are designed to have low fetch costs and thus to execute very quickly. The result is a scheme that works well across a wide range of contention levels.

The AOCC design reduces abort probability and lowers abort costs without adding high message overheads to the basic optimistic design described above. In fact, no additional messages are introduced. All extra information exchanged between the client and server for concurrency control purposes is "piggy-backed" on required messages (fetch and commit requests and replies).

The AOCC server keeps track of which pages are cached at which clients. When a commit occurs at the server with updates for pages cached at clients other than the committing client, the server generates *invalidation messages* for these clients so that they will discard their out-of-date state. There is an *invalid set* per client used to store unacknowledged invalidation messages (including messages that have not been sent yet). Invalid sets are piggy-backed on fetch replies and commit replies sent to the clients, while clients piggy-back acknowledgements for invalidations on fetch and commit requests[2]. When a server receives a client message, it removes any acknowledged invalidations from the client's invalid set.

When a client receives an invalidation message, it discards the invalid state from its cache. (AOCC uses both object and page discards; see Section 2.2.1.) Although invalidations are piggy-backed after updates have already occurred, each message exchange with the server removes all recently-updated state. Thus, client cache contents remain "nearly" up-to-date, and most aborts are due to real conflicts between transactions executing at approximately the same time. Little cache space is "wasted" for storing stale state.

If the current transaction T running at a client has already used an invalidated object, this indicates a conflict with a recently-committed transaction; the client aborts and restarts T. Thus, invalidations cause aborts to happen early. In addition to avoiding unnecessary transaction execution, this mechanism avoids the use of an unnecessary commit-request/abort-reply exchange with the server, and also reduces the cost of commit-time validation checking (described below). In other words, the early abort mechanism "offloads" a significant amount of work from the server.

AOCC improves restart performance by restoring or updating state in the client cache that was accessed by the failed transaction. Prior to the first update of an object, a client appends the object's unmodified state to an object-based undo log.

[2]If there is no normal message traffic between a client and the server for a long time interval, an "I'm alive" message exchange occurs for failure detection purposes. Thus, the exchange of invalid sets and acknowledgements has a bounded worst-case delay.

If an abort occurs, the client retains its modified pages. It uses its undo log to restore the state of any modified object that was not invalidated due to an update by another client. In addition, the server sends updates in an abort reply for all objects accessed by the failed transaction that are out-of-date in the client cache. Due to these restore and update mechanisms, when a restart repeats an access of the previous failed transaction, it is very likely to hit up-to-date information in the client cache.

Note that a failed transaction execution has a *natural* preloading effect: fetches performed during such an execution bring state into the client cache that is likely to be used by the restart. The client-based restore-on-abort mechanism and the server-based update-on-abort mechanism both preserve and augment this effect. Replacing these mechanisms with page discards destroys the natural preloading effect and results in high restart costs.

As discussed further in Chapters 4–6, our simulator uses a range of restart assumptions; we do not assume perfect restarts. Even though our default setup does not use perfect restarts, our results show that AOCC restarts normally have low fetch costs. Relative to first-run executions, restarts are faster and are much more likely to commit. Overall, fast restarts, low message costs, early abort detection, and "nearly" up-to-date client caches all contribute to AOCC's good performance.

2.2.1 Detailed Description

AOCC sends piggy-backed information on client requests and server replies. The server adds object invalidations to each fetch reply, commit reply, and abort reply. In addition, the server sends object updates in an abort reply for objects that were accessed by the aborted transaction and are currently invalid in the client's cache. A client adds acknowledgements for the invalidations it has processed to each commit request and abort request; it also adds discard notifications to inform the server of recent page discards due to cache replacements. When a client or the server receives a message, it always processes the piggy-backed information *prior* to performing other steps.

AOCC Client Description

The client data structures include the client cache described in Section 2.1. For active transaction T, the *ROS* records OIDs (object ids) of objects read by T, while the *MOS* records OIDs of objects modified by T. Writes are also considered reads; the MOS is a subset of the ROS. The *undo log* contains unmodified versions of MOS objects. The *FetchFlag* indicates whether a fetch is in progress when processing invalidations.

1. First read or first write of object X on page P.

- **1a.** **X is missing from cache.** Send server fetch for page P. On reply, process invalidations (FetchFlag = true). Install the fetch reply; if page P replaces page Q, send discard notification for Q with next message. If invalidations caused an abort, restart T, otherwise continue with step 1b.

1b. **X is in cache.** If read access, add X to ROS. If write access, add X to ROS and MOS and append copy of X to undo log.

2. Transaction T reaches commit point. Send a commit request to the server with ROS, MOS, and updated object state for each MOS object; wait for reply.

2a. **Commit reply.** Process invalidations (FetchFlag = false). Clear the ROS, MOS, and undo log; begin execution of next transaction.

2b. **Abort reply.** Process invalidations (FetchFlag = false). Install object updates sent in abort reply. Use undo log to restore state of any MOS object that was not discarded or updated. Clear ROS, MOS, and undo log; restart T.

3. Processing invalidations

3a. For each invalid object X on page P, if the ROS contains no P objects, discard page P, otherwise discard object X. Send piggy-backed invalidation acknowledgement to server with highest invalidation sequence number processed and a list of the pages that were discarded.

3b. If FetchFlag is true and a ROS object was discarded in step 3a, then abort: use undo log to restore state of any MOS object that was not discarded; clear ROS, MOS, and undo log. (Step 1a restarts T.)

AOCC Server Description

We assume a multi-threaded server is used, where any necessary synchronization steps are performed to ensure correct synchronization between threads processing different client requests. For example, the server must send a transaction-consistent copy of a page in a fetch reply. We ignore such details to keep the description simple.

The server data structures include a page-based cache and a modified object buffer (MOB), as described in in Section 2.1. For each page P, *Dir(P)* is the (possibly empty) set of clients caching page P. For each client C, *Invalid(C)* records a sequence of ordered invalidation messages for C that have not been sent or have not been acknowledged. Each message in Invalid(C) has a set of object identifiers and a sequence number. The server piggy-backs the current contents of Invalid(C) on *each* message sent to client C.

1. Piggy-backed information from client C.
On receipt of a message, this processing occurs first:

1a. For a discard notification for page P, remove C from Dir(P).

1b. For an invalidation acknowledgement with sequence number N, remove any message with sequence number $\leq N$ from Invalid(C). For each page P in the list of discarded pages sent with an acknowledgement, remove C from Dir(P).

2. Fetch for page P from client C. If P is not in server cache, read P from disk. Apply all necessary MOB entries for P objects to page P. Send C a fetch reply containing P, and add C to Dir(P).

3. Commit request from C; contains ROS, MOS, object updates. If no ROS object is present in Invalid(C), commit, otherwise abort.

3a. **Read-only commit (MOS is empty).** Send commit reply to C.

3b. **Read-write commit (MOS not empty).** For each page P with objects in MOS and for each client C' other than C in Dir(P), add an invalidation message to Invalid(C') containing the P objects in MOS. Force a commit record containing the object updates to a recoverable transaction log. Add the object updates to the MOB. Send commit reply to C.

3c. **Abort.** Send abort reply to C that includes object updates for any invalid ROS object that is available in main memory (in MOB or cache).

2.2.2 AOCC Discussion

As described above, a commit request is used for both read-only and read-write transactions. In fact, in a single-server setting, AOCC does not need to use a commit request for the read-only case. Section 2.5.3 discusses this optimization.

AOCC's use of invalid sets for out-of-date version checking is a novel mechanism that has some important efficiency properties. Most optimistic schemes maintain version information (version numbers or timestamps) for this purpose. To perform object-level checking, version information must be maintained for each object, resulting in high overhead. For example, if the average object size is 100 bytes, an 8 byte version number adds 8% space overhead. Moreover, it may be necessary to read version information from disk to validate the read set of a committing transaction.

The use of invalid sets avoids the above problems. No per-object state is used, and invalid sets remain small as long as client/server interaction occurs on a regular basis. Since invalid sets are always in main memory, validation never requires disk reads. For read-only commit replies and for abort replies, no log force is required, thus the server can process a commit message and generate a reply within a single short critical section. Note that AOCC only sends new state for an invalid object in an abort reply if this state is available in main memory; disk reads are not used by the update-on-abort mechanism. This approach preserves the fast generation of abort replies. Since invalid set entries for a given client correspond to *recent* updates by other clients, new object state is almost always available in main memory (in the MOB).

2.3 ACBL

To introduce callback locking, we first describe a non-adaptive scheme that uses only page-level locking. We then extend the description to include ACBL's adaptive use of both object-level and page-level mechanisms.

The page-based callback locking design we describe caches read locks across transaction boundaries (while write locks revert to read locks on commit). Following Franklin and Carey we refer to this scheme as CBR [23, 24]. Under CBR, a page is cached at a client if and only if the client holds a lock on the page; any cached page that is not write-locked must be read-locked. Many clients can be caching a page if there is no writer, while if a client holds a write lock, it must be the only client caching the page.

When a client first accesses cached page P, if the access is a read access, it is performed immediately; the presence of P in the cache implies permission to read it. However, if the first access is a write, the server is sent a write lock request for P. When a lock-granting reply is returned, the client records the fact that it now holds a write lock on P; additional write accesses to P by the current transaction do not require write lock requests. For a missing page, a read fetch or write fetch is sent to the server (depending on the mode of the access that caused the fetch). A page is returned; for a write fetch, a write lock is also returned, allowing further writes to the page. When a successful commit occurs, write locks on cached pages are released; normally, read locks are retained and the pages remain cached.

The server keeps track of which clients are caching which pages; a directory data structure maps a page to the clients caching the page. The directory is updated each time a client is sent a page and each time the server learns that a client has discarded a page. Since additions are immediate while removals involve communication, the directory is a conservative estimate of the actual contents of client caches. (This is true for the directory in AOCC as well.) Every client caching page P is implicitly holding a read permission; the directory is an efficient way to keep track of a very large set of read locks. (The server uses standard read-write locking structures for pages that are being updated. The set of pages with active updaters is much smaller than the set of all pages cached at clients.)

The server coordinates read-write access to shared pages via its handling of the read and write fetches and write lock requests sent from the clients. When client C sends either form of write request for page P, the server sends a *callback request* for P to any client other than C that is caching page P. When a client receives a callback for P, if the current transaction has not already read the page, P is discarded and a callback acknowledgement is sent to the server. If the page has already been read, the callback blocks at the client until the local transaction completes; at this point the page is discarded and the deferred callback acknowledgement is sent to the server. Once all callbacks for P have been acknowledged, the server knows that C is the only client caching P, and grants a write lock on P to client C.

When a callback is blocked due to a local reader, the callback acknowledgement is deferred, but an immediate message is sent to the server indicating that a block has occurred. Thus, the server is informed of all blocking initiated at clients. (Additional blocking can occur at the server.) By collecting accurate blocking information, the server is able to detect a deadlock cycle. When such a cycle occurs, the youngest transaction is aborted and its client discards all pages necessary to resolve the deadlock.

To understand the benefits of caching read locks across transaction boundaries, one must compare CBR to a two-phase locking (2PL) scheme that does not retain locks across transaction boundaries. A 2PL scheme uses a round-trip lock request for each new page access, for both read and write accesses. CBR avoids lock requests for read accesses to cached pages, but it can incur two round-trip delays for a write lock request. Since read accesses are normally more frequent than write accesses, the message tradeoff made by CBR should result in lower message costs when compared to a 2PL scheme. Several performance studies have demonstrated that CBR outperforms a 2PL scheme except when high contention workloads are used [23, 24, 57, 58].

CBR has excellent performance for read accesses, and moreover the cost of a write lock acquisition is amortized across accesses to multiple objects on a page. However, the use of page-level locking can perform unnecessary blocking due to *false sharing*: two transactions can conflict over page P even if the object accesses performed by these transactions are disjoint. One can use an object-level locking scheme to avoid this false sharing problem, but this approach introduces a new problem: if a scheme always uses object-level locking, every object update requires a new write lock acquisition. Moreover, for a callback locking scheme, if clients always discard individual objects when they receive callback requests, a client can be sent multiple callbacks for a set of updates to a single page.

The ACBL design uses adaptive-granularity locking to avoid unnecessary blocking due to false sharing while still retaining the benefit of amortized costs for most write lock acquisitions. For pages that are not concurrently read-write shared by two or more clients, ACBL uses page-level locking; for pages that *are* read-write shared, ACBL switches to object-level locking. If concurrent read-write sharing is rare, a client normally uses one write lock request per page updated.

In addition to using two granularities for locks, ACBL uses two granularities for processing callback requests at clients. When possible, clients discard pages from their caches in response to callbacks; if necessary, object discards are used instead while the page remains cached. When a page discard is used, the client is sent only one callback for a page when another client begins to update this page.

These two adaptive mechanisms are clearly related. In fact, the adaptive locking mechanism requires the use of the adaptive cache discard mechanism. Therefore, we first describe adaptive callbacks, and then adaptive locking.

When a client C receives a callback for object X on page P, if the local transaction T running at the client has not read any P objects, C discards P; if T has read some P objects but has not read object X, C discards object X but retains P; if T has read X, C blocks the callback request until transaction T completes, then discards the page. The server is immediately informed of which of these three cases occurs; for the case where the callback is blocked, a callback acknowledgement is sent to the server once page P is discarded.

At the server, when there is a write lock request for object X on page P, a callback request containing X and P is sent to all clients caching page P. After all callback acknowledgements have been received, one of two cases will hold: either all the callbacks ultimately resulted in page P discards, in which case a page-level write lock is granted, or some callbacks resulted in object X discards, in which case an object-level write lock is granted.

Clients can read any object in their cache; discarded objects are not cached, and cause page fetches if accessed. For a page P where object-level locking is used, many clients can be caching P, but if object O is write-locked, only the write-lock holder will have object O in its cache; the other clients have marked O as "missing" from their caches. Such "object marks" are added to a page as objects are discarded due to callbacks. In addition, when the server sends page P in a fetch reply, it must include an initial set of object marks for objects that are write-locked by other clients; these marks are installed along with the fetched page. (Most pages have no object-level writers and no object marks.)

After the server has granted client C a page-level write lock for page P, another client C' can send a fetch request for X on this page. In this case, the server sends a *lock de-escalate* request[3] to C; this request causes C to discard its lock on P. C retains object-level write locks on the P objects that the current transaction has already modified; it sends a list of required object-level write locks to the server in the lock de-escalate reply, and the server grants these write locks prior to processing the original fetch request. As long as the fetch from C' does not conflict at object-level with one of the locks granted to C, client C' can be granted an object-level lock and sent a fetch reply.

Thus, page-level write locks do not cause immediate blocking. A de-escalation is always performed when a second client wishes to use a write-locked page, and blocking only occurs if an object-level conflict exists. Similarly, a callback request only blocks at a client if an object-level read-write conflict occurs, and the server only uses write-write blocking if it receives two write requests for the same object. Since blocking never occurs at the page level, false sharing cannot occur.

2.3.1 Detailed Description

This section presents a detailed description of the ACBL client and server. In this description, callback requests and lock de-escalate requests have been folded together: "callback" requests are used for both purposes.

A client maintains a PROMISES set listing ids of objects it has "promised" to discard once its transaction T reaches its commit point. These are objects for which the client refused a callback since they were in use by T. Additionally, both commit and abort replies contain a "discard list" listing ids of objects to be discarded. (An abort reply happens in response to a fetch request when the server has broken a deadlock by aborting this client's transaction.) These are objects where the server did not send callbacks because it knew (because of responses to previous callbacks) that the request would have been refused.

[3] If C' sent a write fetch request and C holds a page-level write lock, the lock de-escalate request also performs a callback for X.

ACBL Client Description

The client data structures include the client cache described in Section 2.1 and the ROS, MOS, and undo cache described in Section 2.2.1 (as used by the AOCC client). Two additional sets are used. *PLOCKS* records page IDs for the page-level write locks granted to the client. *PROMISES* records the OIDs of objects the client has promised to discard once active transaction T reaches its commit point.

1. First read or first write of object X on page P.

1a. X is missing from cache *or* access is write and P is not in PLOCKS. Send server fetch or lock request that includes X, P, and access mode (read or write); wait for reply.

- **i. Abort reply.** Discard any page containing either an object in PROMISES or an object in the discard list sent in abort reply. Use undo log to restore state of any MOS object that was not discarded. Clear ROS, MOS, PLOCKS, PROMISES, undo log; restart T.
- **ii. Fetch or lock reply.** If fetch reply, install the page and initial marks; if page P replaces page Q, send discard notification for Q with next message. If reply indicates page-level write lock was granted, add P's ID to PLOCKS. If reply includes "Add to PROMISES" directive, add X to PROMISES. Continue with step 1b. (Add to PROMISES occurs when this client was blocked and later removed from a queue for X and other clients remain on the queue.)

1b. X is in cache; C holds required lock. If read access, add X to ROS. If write access, add X to ROS and MOS and append copy of X to undo log.

2. T reaches commit point. Discard any page containing an object in PROMISES.

2a. MOS is empty. If PROMISES is not empty, send the server an *asynchronous read-only commit notification*. No wait occurs; continue with step 2c.

2b. MOS is not empty. Send a commit request to the server with the MOS and updated object state for each MOS object; wait for commit reply. Discard any page containing an object in the discard list sent in the commit reply; continue with step 2c.

2c. Clear ROS, MOS, PLOCKS, PROMISES, undo log; start running next transaction.

3. Callback request for object X, page P, mode (read or write). Send callback reply after performing steps 3a and 3b to determine contents of reply:

3a. If P is in PLOCKS, remove P from PLOCKS and include list of OIDs for P objects that are in MOS in the callback reply.

3b. If mode is write, apply one of the following three cases: If no P object is in ROS, discard page P; if a P object is in ROS but X is not, discard object X; if X is in ROS, "refuse" to discard X and add X to PROMISES. Include description of action taken in the callback reply.

ACBL Server Description

We assume a multi-threaded server is used, where any necessary synchronization steps are performed to ensure correct synchronization between threads processing different client requests. In addition, to keep the description simple, we describe fetch processing and commit processing as if they were atomic steps. (Many interesting cases are therefore ignored; Section 2.3.2 provides an example of such a case.)

The server data structures include a page-based cache and a modified object buffer (MOB), as described in in Section 2.1. For each page P, Dir(P) is the (possibly empty) set of clients caching page P. C's presence in Dir(P) indicates that implicit read permissions have been granted to C for some or all of the objects on page P.

For each page P, either zero or one client holds a page-level write lock. For some pages the server allocates object-level locks, or *O-locks*. An O-lock is a read-write locking structure with an ordered wait queue; locks are granted in request order. O-locks are allocated (steps 2 and 5) and de-allocated (step 8) on a dynamic basis such that the the following invariant always holds: there is an O-lock for object X if and only if X has a writer or a write waiter. Thus, "X has an O-lock" always implies a writer or write waiter, while "X does not have an O-lock" always implies the opposite. If an O-lock is used to block a writer on a reader, the reader has been granted an *explicit read lock*. Clients in Dir(P) hold *implicit read locks* for any P object that has no O-lock.

Steps 2 and 3 describe fetch requests and fetch replies (respectively) but do not describe write lock request processing. A write lock request for object X on page P performs the same lock acquisition steps as a write fetch, and a page is not normally sent in the reply. However, if this request is added to X's wait queue behind a writer or write waiter then once the lock is granted, processing continues as if this were a write fetch request.

1. Piggy-backed discard of page P from client C.

Remove C from Dir(P).

2. Sever receives fetch for object X, page P, mode (read or write).

If P is not cached, initiate a disk read for P. The rest of step 2 can be performed in parallel with this read.

On fetch arrival, one of three cases holds: (2a) callbacks are required; (2b) blocking is required; (2c) a lock can be granted immediately. The details are as follows:

2a. Dir(P) contains clients other than C and there is no O-lock for object X. For each client C' other than C in Dir(P), send C' a callback containing X, P, and mode. On callback completion, if page write lock was held, it has been de-escalated and some object-level write locks are now held. If mode is write, any client that refused to discard object X now holds an explicit read lock on X. If a read or write lock now exists for X, continue with step 2b, otherwise 2c.

2b. **There is an O-lock for object X**. Add C's fetch request (including mode) to the wait queue of object X; trigger deadlock detection. If a lock is granted, continue with step 3; if an abort occurs, continue with step 4. (Blocking occurs in either case unless an immediate abort occurs.)

2c. **There is no O-lock for X and P is not write-locked.** If mode is write: if there is no O-lock for *any* P object, grant C a page-level write lock on P, otherwise allocate an O-lock for X and grant C a write lock on X. Continue with step 3.

3. Send fetch reply to C for fetch of object X on page P.

If P is not in cache, block until disk read for P completes. Apply all necessary MOB entries for P objects to page P.

3a. **There are no O-locks for P objects.** Send fetch reply to C with page P (no initial marks); reply indicates whether C holds page-level write lock on P. Add C to Dir(P).

3b. **There *are* some O-locks for P objects.** Send fetch reply to C with initial mark for each P object O such that O has an O-lock and C is not currently holding a lock on O. If C holds explicit read lock (due to queued write waiter) send "Add to PROMISES" directive in fetch reply. Add P to Dir(C).

4. Send abort reply to C for fetch of object X on page P.

(Deadlock detection discovered a cycle involving C; C's transaction was youngest in cycle.) Send abort reply to C with object discard list containing the OID of each object O where C holds a write lock on O and there is a write waiter. For each object O on page Q added to this list, remove C from Dir(Q). Release all of C's object-level locks and remove C's request from the wait queue of X. For each read lock release for object O on page Q, remove C from Dir(Q).

5. Callback reply from C for object X, page P, mode (read or write).

5a. If C holds page-level write lock on P, allocate object-level locks and make C the write lock holder for the list of P objects sent in callback reply, then release C's page-level write lock on P.

5b. If mode is write, reply indicates one of three actions taken by C. If action was page discard, remove C from Dir(P); if action was object discard, do nothing; if action was "refusal" to discard X, grant C an explicit object-level read lock on X (allocate locking structure for X if not already allocated). If this is last outstanding callback reply (for X/P) resume processing of request that generated the callback.

6. Commit request from C with MOS, object updates.

Release all object-level *read* locks held by C. For each read lock release for object O on page Q, remove C from Dir(Q). Force a commit record containing the object updates to a recoverable transaction log. Add the object updates to the MOB. Send a commit reply to C with object discard list containing the OID of each object O where C holds a write lock on O and there is a write waiter. For each object O on page Q sent in discard list, remove C from Dir(Q). Release all object-level and page-level *write locks* held by C.

7. Read-only commit notification from C.

Release all of C's explicit object-level read locks. For each lock release for object O on page Q, remove C from Dir(Q).

8. Object-level lock release for object X on page P.

(Steps 3b, 3c, 6 and 7 release object-level locks.)

- **8a.** If no lock holder remains after lock release and there are waiters, remove first request from X's wait queue and grant requested lock. If read lock granted and the new first waiter is a read request, remove this waiter and grant another read lock; repeat until queue is empty or first waiter is a write waiter. Server remembers which lock requests were just granted, if any, and resumes processing of the associated client requests after completing steps 8b and 8c.
- **8b.** If there are no writers or write waiters for X, de-allocate X's O-lock.
- **8c.** If *all* allocated O-locks for page P are write locks held by a single client C and no other client is caching P or attempting to cache P (*i.e.*, no in-progress fetches, no lock waiters, no Dir(P) entries for a client other than C) then grant a page-level write lock on P to client C and de-allocate all O-locks for page P.

Note: when a write waiter is removed from a wait queue due to an abort (step 4), it is sometimes possible to grant read locks, and a discard of the locking structure occurs if there is no writer or write waiter.

2.3.2 ACBL Discussion

Like AOCC, ACBL avoids per-object space overheads for concurrency control purposes. ACBL ensures that cached objects are always consistent with the committed state of the database, thus version checking is not required, and ACBL does not maintain per-object version information. Moreover, object-level locking structures are only used for fraction of the objects in active use by the clients.

AOCC and ACBL clients have similar complexity. The ACBL client must handle asynchronous callback requests, but apart from this factor the two client designs are actually very similar. However, the two server designs have very different complexity: the ACBL server is significantly more complex, due to the management of multiple locking granularities and to the asynchronous nature of the design. Since we ignored synchronization issues, the detailed description of ACBL given above is much simpler than the corresponding implementation in our simulator. In contrast, the AOCC server description is a much closer approximation to the corresponding simulator implementation.

In addition to ACBL's inherent complexity, there are a number of opportunities for optimization that improve performance but also complicate the implementation further. Here is one example of such a case. Suppose client C's write fetch request results in a page-level write lock on page P, but a fetch reply has not been sent yet because a disk read for P is still in progress. If another fetch request for a P object arrives at the server, the page-level lock should be "de-escalated" immediately. The alternative is to send the client a page-level write lock and then to immediately send a lock de-escalate request, which is clearly more expensive. (We included this optimization in the simulator implementation of ACBL. While we obviously did not implement all possible optimizations, we attempted to cover cases where it is clear that message savings can be achieved.)

In addition, all read-write locking schemes have some inherent problems that are not shared by optimistic schemes. We discuss three such problems here.

First, for a multi-server system, a locking scheme must implement distributed deadlock detection. This adds complexity to the implementation. More importantly, average blocking delays increase when distributed cycles occur; high-contention performance suffers in a multi-server setting.

Second, OODB applications are highly susceptible to deadlocks due to the frequent occurrence of reads that are later followed by writes, *i.e.*, many lock upgrade requests occur. It is well known that lock upgrades are a common source of deadlock. If two clients run the same transaction code and this code reads X and later writes X, each transaction will acquire a read lock and then both transactions will block attempting to obtain a write lock, creating a deadlock. (This is the most straightforward case; multiple-object cases also occur.)

In a relational database, queries that perform updates are distinguished from queries that do not; updates are "transparent" to the system, giving the system a chance to avoid the use of lock upgrades (at least within a single query). In contrast, most OODB applications are written in a declarative programming language (the most common case being an extension of C++). In this case there is no pre-declaration that writes will occur. Since an object's state is normally "observed" before it is updated, most writes are preceded by reads. While one solution to this lock upgrade problem is to allow application programmers to use special pragmas that pre-declare the need for a write lock, this approach places an unnecessary burden on the programmer.

A third problem is related to the lock upgrade problem. Since most writes are preceded by reads, a combined write lock and page fetch request (a "write fetch") will very rarely occur. In fact write fetches will be rare even if write accesses are not preceded by read accesses: since most accesses are reads, there is a high probability that the first access that "misses" on page P is a read. In particular, when object X is on the same page as another object Y that is "traversed" to "get to" X so that X can be modified, this traversal performs a read access for object Y prior to modifying X, causing a read fetch rather than a write fetch. Such a scenario is common. For example, X might be used as part of the internal representation of Y, and objects in such a relationship are often clustered on the same page.

In summary, there are three performance problems associated with read-write locking: (1) distributed deadlock detection is required for multi-server systems, and the occurrence of distributed cycles causes an increase in the average blocking delay; (2) lock upgrade requests are common, and are prone to deadlock; (3) updates to pages will almost always require both a fetch request and an independent write lock request. In contrast, for all optimistic designs (including AOCC) no deadlock detection is required. Moreover, the inability to anticipate that a write access will occur has *no* impact on performance.

As discussed in Chapter 4, the workload generator for our simulation studies uses a simple transaction model where each object accessed by a transaction is only used once, either as a read access or as a write access. This model is used for reasons of simplicity. Under this model, lock upgrades do not occur, while write fetches do occur. Thus, this model "favors" ACBL, and it is likely that our results over-estimate ACBL's performance when used with a real application workload.

2.4 Design Discussion

One of our design goals was to use the same mechanisms for both AOCC and ACBL when this was feasible, so that our comparison study could isolate the fundamental differences between these optimistic and callback locking designs. As a result, both schemes use an undo log to restore modified objects on abort, and both schemes use an adaptive rule for deciding whether to discard an object or its page from a client cache. Section 2.4.1 discusses the undo log, while Section 2.4.2 discusses adaptive discard policies.

The implementations of AOCC and ACBL used in our simulation studies both use an adaptive-granularity directory structure at the server. This directory is used to avoid generating unnecessary invalidations or unnecessary callbacks. Section 2.4.3 discusses the advantages and disadvantages of using this approach (for both AOCC and ACBL). Note that adaptive tracking of cache contents is not included in the descriptions given above. Describing how to *add* this feature is not hard, but including it in our summary descriptions results in too much detail.

2.4.1 Undo log

AOCC clearly gets more benefit from undo logging, as it incurs more restarts; a practical implementation of ACBL might not use undo logging. This raises two fairness question for our simulation study. If undo logging were removed from ACBL, would this significantly lower ACBL's client processing costs, or would it provide a significant amount of additional memory so that more pages could be cached at the client? If the answers to these questions were "yes," our performance study would have under-estimated ACBL's performance.

Undo log space overhead is low due to the use of object logging. Suppose the pages touched by one transaction use 25% of the client cache slots, 20% of these pages are modified, and the modified objects on an updated page represent an average of 20% of the page size. In this case the undo log state represents 1% of the client cache size.

Measurements of the Thor object-oriented database system [43] show that undo log maintenance has very low cost. For example, for the "T2b" traversal from the OO7 benchmark [8], the overhead for maintaining an undo log was less than 2.5% of non-commit latency for a "hot" traversal that used no fetch requests, and less than 0.5% for a traversal that did use fetch requests [13]. For the T2b traversal, roughly one in four object accesses is a write; we expect most workloads have low undo log costs compared to this traversal.

Given the low overhead of undo log maintenance, we decided not to introduce a CPU charge for copy-on-write in our simulation model. To verify that this decision does not unduly favor AOCC, we performed some simulation experiments that compare a normal ACBL setup with an AOCC setup that uses a higher "write think time" (the client CPU charge per write). As expected, there was little change in relative performance (less than 1%). Thus, our decision to use the undo log for both schemes has no significant impact on our relative performance results.

Note that there is another common approach to providing client-based undo: objects are copied on first use or first update from a shared cache to a private cache, thus unmodified object copies remain at the client (in the shared cache), and discarding a modified private copy of an object "restores" its state. Many commercial and research systems use per-application private caches, including systems that use optimism (see, *e.g.*, GemStone [6]) and systems that use locking (see, *e.g.*, SHORE [9]).

2.4.2 Adaptive Cache Maintenance

The fundamental adaptive cache discard rule is this: discard the page if the current transaction is not using it; otherwise discard the object. ACBL and AOCC both apply this rule to callbacks and invalidations. This adaptive rule is an integral part of ACBL's design; we "mirrored" this rule in AOCC's design so that the two schemes would perform similar client cache maintenance actions.

There is one difference between the designs: different approaches are used for commit and abort processing. For AOCC, if invalidations are processed at the commit or abort point of a transaction, invalidations cause object discards for pages that were accessed by the completed transaction, and cause page discards otherwise. Thus, AOCC always uses the adaptive rule, where it treats a transaction's accesses as "in use" during commit and abort processing.

In contrast, ACBL uses page discards for "deferred callbacks" that are processed on transaction completion (commit or abort). Discards for deferred callbacks are recorded by the client in the PROMISES set, while an additional list of discards is sent in commit and abort replies. Page discards are used because there is always an associated write waiter; a page discard may allow a page-level write lock to be granted. (A clever implementation of ACBL can detect cases where a page discard cannot possibly result in a page-level write lock being granted; object-level discards can be used for these cases.)

Our experiments show that the caching behavior of the two schemes is very similar. While there is a time lag when you compare aggressive cache maintenance (callbacks) and lazy cache maintenance (piggy-backed invalidations), this timing difference has little impact on overall caching behavior. For both schemes, clients normally use object discards for frequently accessed pages and page discards for other pages. The "steady state" content of a client cache is determined by the access pattern used at that client and the update patterns used at other clients.

It was important to "control" for caching behavior to perform our initial comparison study; using a different cache maintenance strategy for AOCC would cloud the analysis of our results. Now that we have performed a large set of experiments using this controlled approach, it would be reasonable to do a "companion study" that examines the use of several caching strategies for AOCC. Using page discards "whenever possible" is probably "too eager" a strategy; it seems better to retain recently-used pages and frequently accessed objects.

2.4.3 Adaptive Directory Information

If an object has been discarded from a cached page, we say the object has been "marked" as missing. Marks are used in the descriptions of AOCC and ACBL given above: client caches record marks as they discard objects from cached pages, and moreover the ACBL server sends "initial marks" for page P (for objects that the fetching client does not have permission to read). As described above, the AOCC and ACBL servers do not track the marks in client caches; the directory at a server only tracks which pages are cached at each client. In contrast, the implementations of AOCC and ACBL used in our simulation study do track object marks. This section briefly describes this mechanism and its implications for AOCC and ACBL performance.

Each server records marks as it learns of object discards, while the ACBL server also records the initial marks for pages sent in fetch replies. (AOCC always sends "full" pages.) Mark information is discarded when a page is discarded or when a page is re-fetched and a "full" version is sent to the client. The space used for tracking object marks is reasonable for our simulation environment. However, a practical implementation would need to limit the space allocated for this purpose. We believe a simple strategy would work well: for each client, the server should keep the N most recent marks, *i.e.*, the N most recent object-level discards. N only needs to be large enough to retain marks for the most frequently-updated objects; the space requirements for storing "recent marks" should be quite reasonable.

When object mark information is available, it is used as follows. Under AOCC, an invalidation for object X is not added to client C's invalid set if C is not caching X. Under ACBL, a callback for object X is not sent to client C if C is not caching X. Thus, unnecessary invalidations and callbacks are avoided. When mark information is not available and unnecessary invalidations or callbacks are used, they are "harmless;" they do not cause additional aborts or additional blocking. Tracking object marks is an optimization; it is not required for correctness.

Both schemes derive benefits from this approach. AOCC invalid sets are smaller, thus message sizes are smaller, the clients perform less invalidation processing on message reception, and the server uses a smaller invalid set when it performs a commit-time validation check. ACBL benefits even more since it avoids callbacks for pages that are read-write shared. For example, suppose a client has updated object X using an object-level write lock, and then performs another write access to X in a subsequent transaction. Assuming no other client is caching X, the server can grant another object-level write lock for X immediately, even if many other clients are caching X's page.

For both schemes, the omission of an invalidation or callback has another implication: the client that is not contacted will not discard the page. For ACBL, there is a potential problem case due to omitted callbacks: if a client is not actively using page P but has some "old marks" for P objects, it may not be sent a callback, in which case it will not discard P, and page-level write locks will not be granted to clients that *are* using the page.

This "old marks" problem case is easily eliminated. If the server only retains "recent marks," as suggested above, the server will not retain "old marks" and it will end up sending callbacks to clients that are no longer using a page with some marks on it. (The server will only know about marks for pages that the client has accessed recently, since marks are only generated for pages that a client is actively using.)

For our simulation study, where all marks are retained at the server, this "old marks" case is very unlikely to occur, and, if it does occur, the phenomenon will be short-lived. Our workload generator uses uniform random selection for choosing the updates on a page. After client C is no longer using a page, each new update to the page is random with respect to any marks at client C. Thus, with each update to page P it becomes increasingly likely that C has discarded the page due to a callback.

It is true that the existence of some marks on a page can add a small delay to the discard process. However, we believe that adding some hysteresis to the mechanism that "switches" back to the use of page-level locking is likely to be useful. If the page is not in use, the switch will occur. If the page *is* still in use, a small delay may prevent an "eager" page discard. When a page that is actively read-write shared by multiple clients switches to page-level locking, this switch can have significant costs associated with it: unnecessary callbacks are used, page discards occur; the page discards are soon followed by re-fetches of the page, and these fetches are delayed while a lock de-escalation is carried out.

The worst-case example for "eager" page discards is an ongoing false sharing scenario between two clients. Suppose clients X and Y are read-write sharing page P, where X is not reading objects that Y updates and *vice versa*. Clients X and Y never need to use a fetch for page P: one client never needs to fetch the updates performed by the other. If each client has marked the objects that are being updated by the other client and the server uses this information to avoid callbacks, no callbacks are ever used: write lock requests can be granted immediately at the server, and moreover page P is never discarded and then re-fetched. In contrast, if a P update by one client causes the other to receive a callback, this callback either adds latency to an object-level write lock grant and has no other effect, or it causes a page P discard. In the latter case, as soon as the client that discarded P uses it again, P is refetched; the fetch itself is expensive, and it may also be necessary to carry out a lock de-escalation prior to sending the fetch reply. (Eager discards for pages where an object is actually both read and written by multiple clients are ultimately less costly than the false sharing case, as fetches are required to move updated state from one client to another.)

2.5 Design Comparison

This section analyzes the expected relative performance of AOCC and ACBL. Sections 2.5.1 and 2.5.2 consider performance for workloads with low and high contention, respectively, while Section 2.5.3 discusses read-only transaction performance. The simulation studies presented in Chapters 4–6 validate the analysis presented here.

The average latency for a successful transaction commit can be modeled by the following formula:

$$\begin{aligned}
&\text{latency} = \text{success-latency} + \text{failure-latency}\\
&\text{success-latency} = \mathrm{E}_s + \mathrm{F}_s + \mathrm{L}_s + \mathrm{B}_s + \mathrm{C}_s\\
&\text{failure-latency} = \mathrm{E}_f + \mathrm{F}_f + \mathrm{L}_f + \mathrm{B}_f + \mathrm{C}_f
\end{aligned}$$

where

E = <u>E</u>xecution latency at client for local read and write accesses
F = <u>F</u>etch latency for fetch requests/replies, including disk read latency
L = <u>L</u>ock request latency for lock and callback requests/replies
B = <u>B</u>locking latency at server due to object-level conflicts
C = <u>C</u>ommit latency for commit requests/replies and processing at server

Note that we divide the costs associated with locking into two components. L is the message latency due to the use of locking; B is the latency due to waiting in an object-level lock queue at the server for a lock release. (When callbacks are used, blocking begins after all replies have been processed.) If ACBL uses the same write fetch request to both acquire a lock and fetch a page, the request and reply between the fetching client and the server fall under F, while callbacks fall under L. In both cases, any blocking that occurs falls under B.

AOCC's main advantage is that it has no L or B costs. ACBL's main advantage is that it has low failure costs.

2.5.1 Low Contention

This section considers low-contention performance for read-write transactions. Under low contention, an optimistic scheme incurs few aborts while a locking scheme incurs little blocking due to real data conflicts; the failure terms and B terms are small enough to ignore. This leaves the following latency breakdown:

$$\text{latency} = \mathrm{E}_s + \mathrm{F}_s + \mathrm{L}_s + \mathrm{C}_s$$

Since each scheme tends to execute a transaction just once, the number of accesses by each scheme is roughly the same; the same holds for number of fetches. Thus, the

E and F terms are roughly equal; the L and C terms determine relative performance. Both schemes have roughly the same C cost, thus the difference between the schemes is ACBL's L cost for lock and callback messages:

Read-Write Transaction
difference(ACBL-AOCC) = L_S cost for ACBL

For a lock request or callback request that succeeds immediately, which is the common case under low contention, processing the request is very fast: only main-memory data structures are used. Thus, ACBL's additional cost L_S is almost entirely due to message costs. Since both schemes use the same number of large messages (fetch replies and read-write commit requests), the difference in message *cost* can be approximated by comparing message *count*. In short, *message count predicts relative performance for low contention workloads.*

Low-Contention Message Count Model

Figure 2-1 presents a simple message count model for the page-level accesses performed by a transaction. For low contention, we expect page-level write locks are granted, thus we only need to consider page-level accesses to produce a message count. The figure also summarizes the message counts required for read-write commits.

	Message Count		**Round-Trip Count**	
	AOCC	ACBL	AOCC	ACBL
read hit:	0	0	0	0
read miss:	2	2	1	1
write hit:	0	2+2K	0	1 or 2
write miss:	2	2+2K	1	1 or 2
read-write commit:	2	2	1	1

Figure 2-1. Low Contention Message Count Model

AOCC uses a round-trip request to the server only to fetch a missing page and to request a commit. ACBL uses a round-trip request to get a page-level write lock on a cached page that is to be written; this request results in K callbacks, where K is the average number of clients contacted per write lock acquired; it grows with the number of clients, since more clients will be caching the same page. When K is zero, only 1 round-trip is used per write lock, since no callbacks are sent, otherwise 2 round-trip latencies occur. (Callbacks occur in parallel, thus a set of callbacks has one round-trip latency.)

The reason for showing both message count and round-trip count is that the actual messaging costs incurred by ACBL are not captured exactly by either count; callbacks are sent in parallel, but even if the server were to use a multicast, it must process the replies individually. (The server can send one message, but must receive K replies.) So the real cost is somewhere between the apparent cost as represented by the two counts. Regardless of which count we look at, we can clearly see that AOCC is always the same or better.

To make things more concrete, we apply this count model to a specific workload. Suppose a workload results in the following average page-level access pattern per transaction: 12 read-hits, 4 read-misses, 3 write-hits, and 1 write-miss. (This is a 75% page hit ratio at the client and a 20% page update probability; approximately what the HOTCOLD workload produces with perfect write clustering, *i.e.*, the most favorable case for ACBL.) In addition, each transaction uses a commit request and reply. Our model gives the following counts:

AOCC message count:	12
AOCC round-trips:	6
ACBL message count:	18+8K
ACBL round-trips:	9 or 13

With $K = 0$ (no callbacks required to obtain a write lock), ACBL uses 50% more messages (18 vs. 12) and 50% more round-trips (9 vs. 6). With $K > 0$, ACBL uses around twice the round-trips (13 vs. 6). The number of extra messages used by ACBL depends on the number of clients K that must be contacted (on average) to acquire a write lock. For $K = 2$, ACBL uses nearly 3 times the messages (34 vs. 12).

While AOCC has lower costs than ACBL, and is clearly the better scheme for low contention workloads, it is important to point out that ACBL is a well-designed locking scheme: other locking schemes have higher messaging costs. For example, an alternative to ACBL is a client-server implementation of standard two-phase locking, where the server maintains all locks and a round-trip is required for every new read or write access. Standard two-phase locking requires 2 messages (1 round trip) per page access, regardless of whether the page is cached. For the specific example above, standard two-phase locking would use 42 messages (21 round-trips). Thus, as long as K is not too large, ACBL is a better scheme than standard two-phase locking.

Write Clustering In the example above, the object writes performed by the transaction caused 4 of 20 accessed pages to be modified. In other words, if there were 40 objects writes, these writes were "clustered" onto just 4 of the 20 pages accessed. A much worse scenario could occur: the 40 writes could be spread across all 20 of the accessed pages, causing 20 page updates. In this case ACBL would require $22 + 20K$ messages, rather than $18 + 8K$. Thus, the quality of write clustering has a large impact on ACBL's message costs. In contrast, AOCC is relatively immune to the quality of write clustering. For the 20 page update example, AOCC's message count would remain the same: 12 messages would still be used. Note that page updates cause pages at other clients to be discarded; poor write clustering results in higher fetch costs for both schemes. The key point is that AOCC is relatively immune to write clustering as compared to ACBL.

For our comparison study, we chose to model three different write-clustering cases: good, average, and poor write clustering. The details are explained in Chapter 4. Most experiments use average write clustering, while Section 6.2.1 presents sensitivity analysis experiments that demonstrate the impact of varying the quality of write clustering.

2.5.2 High Contention

This section considers high-contention performance for read-write transactions. As contention increases, AOCC's failure costs increase, while ACBL's locking and blocking costs increase. ACBL also starts to have failure costs as contention leads to deadlock-induced aborts.

For each successful commit, both schemes must carry out a set of successful executions and a commit request, thus E_s and C_s costs are roughly equal. For transactions that perform a number of fetches, most AOCC aborts are detected early. As a result, the number of commit requests that fail is small. When the abort rate is factored in, extra commit-time costs are very small; C_f is not a significant factor[4]. For now we assume ACBL's failure costs do not matter; we discuss deadlock-induced aborts at the end of this section. The remaining costs are:

$$\text{relevant AOCC latency} = F_s + E_f + F_f$$
$$\text{relevant ACBL latency} = F_s + L_s + B_s$$

When transaction X blocks on transaction Y, the rest of Y's latency (until Y commits or aborts) is also incurred by X. To analyze these costs it is helpful to split the average blocking cost B_s into components E_b, F_b, L_b, and C_b. (For simplicity, Y's blocking costs are ignored.) E_b is average time per commit spent blocking while the "blocker" performs access executions, F_b is average time per commit spent blocking while the "blocker" performs fetches, and so on. If we replace B_s with these sub-costs and group terms by cost category, the remaining costs are:

$$\text{relevant AOCC latency} = E_f + (F_s + F_f)$$
$$\text{relevant ACBL latency} = E_b + (F_s + F_b) + (L_s + L_b) + C_b$$

In other words, both schemes have additional costs due to restarts or blocking for execution and fetch costs; ACBL has additional costs due to locking and blocking while commits occur that have no AOCC counterparts.

We first consider fetch costs. We give a back-of-the-envelope calculation here to show why fetch costs are normally roughly equal. To do this, it is necessary to choose abort and blocking rates. Our simulation results show that AOCC's abort rate (aborts/commit) is normally lower than ACBL's blocking rate (blocks/commit). This difference in rates is due to the fact that ACBL uses a symmetric conflict relation while AOCC uses an asymmetric conflict relation. AOCC allows concurrent activity

[4]If an early abort does not occur, the abort reply from the server helps to preload the client cache, avoiding future fetches. Thus, adding a commit request to an abort case removes one or more fetches.

for potential read-write conflicts, while ACBL must block when there is a potential conflict. Some of the additional concurrency allowed by AOCC results in aborts, but some results in commits. The commit cases determine the difference between ACBL's blocking rate and AOCC's abort rate. (This argument applies because AOCC keeps client caches "almost" up-to-date, thus most aborts are due to data contention between clients. An optimistic scheme that does not keep caches "almost" up-to-date incurs many more aborts due to out-of-date accesses, and its abort rate can be much higher than the blocking rate of a locking scheme.) For our example, we use an AOCC abort rate of 0.20 and an ACBL blocking rate of 0.25. For ACBL, when transaction A blocks on transaction B, we assume the average case is that B has already performed 70% of its accesses when this block occurs. We also assume fetches are uniformly distributed across a transaction's accesses, thus 30% of B's accesses and 30% of its fetches are performed while A blocks on B.

AOCC's fetch costs ($F_s + F_f$) remain low due to early abort detection, the natural preloading effect of failed executions, and the use of restore-on-undo to prevent page discards. The extra costs due to aborts are also only incurred for transactions that abort, thus the real cost is scaled by the abort rate. For example, suppose first-run transactions that commit use an average of 4 fetches, while transactions that first abort and then commit perform 3 fetches during the first-run failed execution and 2 fetches during the restart, for a total of 5 fetches. If the abort rate is 0.2 aborts/commit, the average fetch cost is $0.8(4) + 0.2(5) = 4.2$.

ACBL's fetch cost ($F_s + F_b$) are calculated in similar fashion. Suppose an ACBL transaction that does not block averages 4 fetches per commit. If a "blocker" has completed 70% of its fetches at the time blocking occurs, it performs 30% of its fetches, or 1.2 fetches, during blocking. If a transaction that blocks performs 4 fetches and waits for 1.2 other fetches to complete, its latency due to fetching is 5.2 fetches. With a blocking rate of 0.25, the average fetch cost is $0.75(4) + 0.25(5.2) = 4.3$.

While it appears that ACBL's extra latency of 0.3 fetches per commit is slightly higher than AOCC's extra latency of 0.2 fetches, an additional factor must be considered. ACBL's extra fetch latency is due to blocking; the number of fetches *executed* per transaction remains at the normal case of 4 fetches per commit. In contrast, AOCC's extra fetch latency is due to additional fetches: AOCC *executes* 4.2 fetches per commit. Some fraction of AOCC's extra fetches result in disk reads, thus AOCC has slightly higher disk utilization. As a result, AOCC's average fetch latency FL_{AOCC} is slightly higher than ACBL's average fetch latency FL_{ACBL}. When we scale the fetch counts by these costs, $4.2(FL_{AOCC})$ and $4.3(FL_{ACBL})$ are roughly the same.

AOCC's extra access cost E_f is somewhat higher than ACBL's extra access cost E_b. Unlike the fetch case, the accesses performed during a failed transaction do not reduce the number of accesses that must be performed during restart. Early abort detection does help reduce the total access count: the number of accesses does not double if an early abort occurs. Suppose transaction length is 200 and an early abort is detected at the 150 access point. With an abort rate of 0.2, the extra cost of these 150 accesses is $0.2(150) = 30$. For ACBL, if a "blocker" has completed 70% of its accesses at the time blocking occurs, it performs 30% of its accesses, or 60 accesses, during blocking. With a blocking rate of 0.25, the extra cost of these 60 accesses is $.25(60) = 15$. This difference between the schemes has a relatively small impact. Local access costs are not as relevant as the other costs in the model, for two reasons. First, local accesses are fast. Second, local accesses do not consume server resources, while the other costs in our latency model do involve use of the server.

We have now discussed all of AOCC's costs, and argued that they are roughly equal to components of ACBL's cost. The remaining ACBL components represent additional overhead that has no AOCC counterpart:

$$\text{extra ACBL latency} = (\mathrm{L}_s + \mathrm{L}_b) + \mathrm{C}_b$$

ACBL's normal lock message costs (L_s) increase with an increased update frequency (more write lock requests) and with an increase in read-write sharing of pages (more pages require object-level locking, which is more message-intensive than page-level locking). In addition, while a blocked transaction waits, the "blocker" is using lock requests and callbacks (adding latency L_b). The use of lock requests and the frequency of blocking both rise together. In summary, ACBL pays a high price for the use of lock requests and callbacks and then pays some fraction of this cost yet again due to blocking on other transactions.

ACBL also incurs blocking latency due to waiting while another transaction carries out the first part of a commit request: a blocked transaction waits on the "blocker" while a commit request (or read-only commit notification) is generated at the client and sent to the server; this waiting ends when the server releases the read or write lock causing the blocking. (The commit reply sent to the "blocker" is not part of the latency of the "blockee.")

Our performance results in later chapters will show that under very high contention, ACBL's performance drops dramatically relative to AOCC's performance. One of the reasons for this is that ACBL starts to incur high costs due to deadlock-induced aborts. Each ACBL abort is very expensive, due to wasted lock acquisition costs. An ACBL transaction that aborts has "wasted" all of the time that it spent acquiring write locks, as these locks are released on abort. In addition, the lock requests and callbacks used for write lock acquisition are wasted, and these messages consume a critical shared resource, namely server CPU.

Compared to an ACBL abort, an AOCC abort is very "cheap." Since restarts mostly hit in the client cache, restarts run much faster than first-run executions. The low fetch costs result in a relatively low additional load on the server. Early abort detection also reduces the server cost associated with aborts. As a result, much of the resource consumption due to an abort occurs at the client. We assume a client C only runs one transaction; this transaction incurs all of the extra latency due to wasted CPU cycles at C. If three transactions are active at client C and one aborts, then these three transactions have added latency due to wasted client CPU cycles at C. The key point is that the large majority of all transactions in the system are running at clients other than client C; these transactions are only penalized for the server resources consumed by an abort that occurs at C.

This point is very important but rarely discussed: compared to single-site and parallel database systems, aborts due to optimism are much less expensive in client-server systems. Client-server systems run only a few transactions at a time per client, while these other systems often have hundreds of active transactions per node. An abort in these other systems therefore adds latency to hundreds of transactions. Moreover, in a client-server system the main memory of the client is dedicated to one or a few transactions, making it feasible to retain all of the data touched by a transaction; this makes fast restarts possible. In a system with hundreds of transactions per node, the memory must be divided among all these transactions, making it less likely that the data of all active transactions can fit in main memory.

As we discuss in Section 6.3, several conditions must hold for ACBL to outperform AOCC: AOCC must have a high abort rate; restarts must perform a significant number of "new" accesses that miss in the client cache; AOCC must drive the server disks near their saturation point (while ACBL must not drive either resource to saturation). The first two conditions cause AOCC to perform many additional fetches. The third condition causes AOCC's average fetch latency FL_{AOCC} to be significantly higher than ACBL's average fetch latency FL_{ACBL}. If AOCC performs more fetches but the server disks are *not* near their saturation point, AOCC's "blocking" costs due to resource contention are not as high as ACBL's blocking and abort costs due to data contention.

2.5.3 Read-Only Transactions

A read-only transaction can incur aborts or blocking during its execution. For high contention, ACBL blocking times must be compared to AOCC's extra costs due to restarts. If AOCC restarts add low additional fetch costs, its aggregate cost for carrying out a read-only transaction will be lower than ACBL's aggregate cost; the reasoning is as given above.

For low contention, we can compare expected performance by considering message costs for a successful commit. Both AOCC and ACBL will use fetches for cache misses, while neither scheme uses additional messages (ACBL does not use write lock requests or callbacks). As a result, low contention performance through the commit point is equivalent.

When a read-only transaction reaches its commit point, an ACBL client allows an immediate commit[5], while an AOCC client uses a commit request to ensure that a consistent state of the database was read by the transaction. Thus, AOCC incurs one extra round-trip compared to ACBL. We study the impact of this extra request in Section 6.2.2. The result is an intuitive one: when the execution of the transaction up to the commit point has low latency compared to a round-trip message exchange, ACBL can outperform AOCC by a significant margin; when the execution has high latency compared to a round-trip exchange, the difference in relative performance is small.

[5]Supporting an immediate commit requires that the client has a guarantee that its read locks will not be unilaterally discarded by the server until some future time t; this time window (or lease [29]) is advanced as long as the client and server remain in contact.

While we describe read-only commit requests for AOCC and also use them in our simulation study, in fact AOCC does not need to use them in a single-server system. At the server, updates are applied in commit order, and transaction-consistent invalidation messages are generated due to these updates. Once a client has processed all invalidations generated through time T_{reply} (the time that the last fetch reply was sent to the client) it has a consistent cache with respect the committed state of the system at time T_{reply}. If a read-only transaction is not aborted due to invalidations, it can be "serialized" at this point in time; an immediate commit can occur at the client.

This immediate commit case does not hold for a multi-server system: processing invalidation messages sent from several different servers does not result in a global consistency property for the client cache. Therefore, supporting immediate commit for a multi-server optimistic scheme is an area of future work; Section 7.2 discusses this problem.

Chapter 3

Related Work

There is a rich literature of work on concurrency control. Concurrency control was originally invented for centralized single-version database systems; single-version systems maintain only one copy of each data item. Fundamentally, there are two kinds of concurrency control, pessimism (locking) and optimism. The seminal paper on standard two-phase locking is by Eswaran *et. al.* [21]. The seminal paper on optimism is by Kung and Robinson [38]. Bernstein *et. al.* and Gray and Reuter [30] both provide good overviews of concurrency control and recovery issues, including later work on distributed database concurrency control.

Optimistic schemes can be classified [33] into *forward* and *backward* validation schemes. Backward validation schemes such as AOCC validate a transaction against already-committed transactions, while forward validation schemes validate a transaction against active transactions.

Multi-version concurrency control can be used for both locking schemes [59] and optimistic schemes [6, 2, 39, 41] to isolate read-only transactions from read-write transactions. Multi-version schemes ensure that read-only transactions always commit, but have higher space overheads than single-version schemes.

An additional approach to concurrency control, not studied in this thesis, is timestamp-ordering. Timestamp schemes choose a serial order for transactions prior to their commit and use a combination of optimistic and pessimistic methods to try to commit the transactions in this order. Timestamp schemes are summarized in [4].

All of these mechanisms can be applied to a client-server OODB. For example, the GemStone [6] and Jasmine [39] OODBs use multi-version optimistic schemes.

Below we focus on work directly related to this thesis: Sections 3.1–3.3 review three studies of single-version client-server schemes. These studies, by Franklin and Carey [23, 24], Wang and Rowe [58, 57], and Wilkinson and Neimat [61], examine nine different concurrency control schemes that use inter-transaction caching. These schemes use coarse-granularity (page-level) conflict detection. (AOCC and ACBL use fine-granularity conflict detection, as described in Chapter 2.) Figure 3-1 presents a brief description of each scheme. In our review of the studies, a more complete description for each scheme is provided when the scheme is first discussed.

In addition to comparing different inter-transaction caching schemes, each of the studies uses a non-caching version of standard two-phase locking (Basic Two-Phase

Scheme	Description
C2PL: Caching 2-Phase Locking	Server-based pure locking scheme. Combined version check and lock request on first access to cached page.
CBR: Callback-Read	Callback locking scheme. Read locks cached with pages.
CBA: Callback-All	Callback locking scheme. Read and write locks cached with pages.
O2PL: Optimistic 2-Phase Locking	Client-based deferred locking scheme. Immediate local lock acquisition; remote write lock acquisition deferred until commit.
No-Wait Locking	Server-based asynchronous locking scheme. Synchronous lock acquisition on fetch, immediate asynchronous lock request on first access to cached page.
No-Wait Locking with Notification	No-Wait Locking augmented with explicit update messages sent immediately after each read-write commit.
Certification	Backward-optimistic scheme. Version check on first access to cached page.
Cache Locks	Server-based deferred locking scheme. Synchronous read lock acquisition on fetch; other read and write lock acquisitions deferred until commit. Piggy-backed invalidations used for cache maintenance and early abort checking.
Notify Locks	Backward-optimistic scheme. Explicit update messages sent immediately after each read-write commit. All validation performed at client: update messages are used for early abort checking and also (via multi-phase handshake with server) for commit-time validation.

Figure 3-1. The Nine Concurrency Control Schemes Discussed

Locking, or B2PL) as a "baseline" for measuring the benefit of inter-transaction caching. In each case the results show that retaining pages across transaction boundaries provides performance benefits when workloads exhibit some inter-transaction locality of reference. We do not discuss the B2PL experiments below.

A final section discusses shared-nothing parallel database systems that use data replication. Concurrency control for such systems requires cache maintenance, thus these systems are related to client-server systems. Section 3.4 summarizes the differences between such database systems and client-server systems, and discusses the impact of these differences on concurrency control tradeoffs.

3.1 Franklin and Carey

Franklin and Carey [23, 24] studied a number of interesting design tradeoffs. We summarize the key tradeoffs, and also compare the semi-optimistic O2PL (Optimistic

2-Phase Locking) scheme used in this study with AOCC.

3.1.1 C2PL vs. CBR vs. CBA

Three pure locking schemes are included in the study. The tradeoff studied across these three schemes is the value of caching locks with cached pages across transaction boundaries. C2PL (Caching 2-Phase Locking) caches pages but does not cache locks; CBA (CallBack-All) caches all locks held at the commit point across a commit; CBR (CallBack-Read) caches only read locks across a commit, where write locks are downgraded to read locks as part of commit processing.

CBR was described in Section 2.2. CBA is like CBR but also caches write locks across transaction boundaries. If a fetch occurs while a write lock is cached, the lock must be called back. The two schemes often have similar performance, but CBA is more sensitive to shared access frequency. Write lock caching is found to be truly useful only for a page that is "private" to one client, where other clients do not cause callbacks for a cached write lock.

C2PL is a caching version of basic two-phase locking (B2PL). C2PL caches pages across transaction boundaries, but does not ensure that they are valid. On each initial read or write access to a new page, a fetch or lock request is used. Version numbers are kept with pages, and a lock request for a cached page will cause the page to be updated if it is invalid. This scheme has a fixed message count, regardless of the number of cache hits or the number of updates: a fetch request or a lock request is used on every page access. The callback schemes use few messages compared to C2PL when updates are infrequent and locality of reference is good, thus they perform better for this case. Poor locality causes all schemes to use many fetches per transaction, making their performance roughly equal. High contention causes the callback schemes to have high message costs relative to C2PL, due to the use of many callbacks; C2PL performs better for this case.

3.1.2 O2PL vs. CBR

Franklin and Carey also study a deferred locking scheme, O2PL (Optimistic 2-Phase Locking). O2PL is essentially a "deferred callback" version of CBR. Each client C uses a local lock manager, and local read and write locks are acquired by the transactions running at C as they perform page accesses. Unlike CBR, other clients are not forced to discard pages that are updated at C; there can be concurrent readers and writers of a page. O2PL "optimistically" assumes that transactions at other clients will not conflict with the transactions at client C.

When transaction T running at client C requests a commit, the server is now responsible for carrying out the "deferred callbacks" for T: messages are sent to all clients other than C that are caching pages updated by T, asking these clients to remove the pages. As with CBR callbacks, a local transaction that is reading a page will block a page discard request until it completes. Unlike CBR, a write-write conflict causes an immediate abort of one of the transactions, since both transactions have already updated the page. If all pages are discarded, transaction T can commit. If any abort replies are returned instead, T aborts.

The study compares O2PL to CBR to examine the issue of locking vs. deferred locking. O2PL has slightly better performance under low contention. O2PL uses fewer messages than CBR for low contention, and message count predicts relative performance, thus explaining this low-contention result. As contention increases, O2PL's performance degrades rapidly, while CBR is more robust with respect to contention level. As a result, CBR has much better high contention performance.

O2PL's poor high-contention performance is not due to the use of optimism alone, but rather to the use of both optimism *and* locking. Under high contention, a pure locking scheme has high blocking latency but a relatively low abort rate, while a pure optimistic scheme has a high abort rate but performs no blocking. In contrast, O2PL simultaneously incurs both high blocking latency *and* a high abort rate. This combination produces very poor performance. Any blocking performed by an aborted transaction is "wasted" blocking: write permissions that were acquired are dropped, and must be acquired again during restart. Restarts are therefore very expensive: the same blocking delays that occur during a first-run execution also occur during a restart.

The combined effect of wasted blocking and non-wasted blocking can easily be seen in the results reported by Franklin and Carey: under high contention, O2PL does *more* blocking than CBR or C2PL. A scheme that performs more blocking than standard two-phase locking has little in common with an optimistic scheme that performs no blocking. For this reason, we prefer to use the term "deferred locking" for schemes such as O2PL.

All of the deferred locking schemes described in this chapter can simultaneously incur a high abort rate and high blocking times, and therefore all have poor high contention performance. The schemes do differ in one respect, however: some schemes acquire read and write locks in roughly the same order that the read and write accesses occur, while other schemes acquire read locks immediately and defer write lock acquisition to the commit point. The latter approach has the following problem for high contention workloads: deadlock cycles are more likely.

O2PL is an example of the latter type of scheme: local read and write locks are acquired immediately, while remote write locks are acquired at the commit point. For a scheme such as CBR where all locks are acquired immediately, if read-write blocking occurs, a read request blocking on a writer is as likely as a write request blocking on a reader. Under O2PL, a remote write request blocking on a local reader becomes more probable, while the opposite case becomes less probable. Such an inequality makes a deadlock cycle more likely. (Consider two overlapping transactions X and Y, where X reads something Y writes and *vice versa*. By delaying the write lock requests, it becomes more likely that each write lock request will end up blocking on a local read lock held by the other transaction.)

Note that it is *inter-client* deadlocks that become more likely by deferring remote write lock acquisition. Each O2PL client is responsible for detecting local deadlocks, while the server periodically collects all local waits-for graphs at the clients and checks for inter-client deadlock cycles. Distributed deadlocks add additional latency to all transactions involved in the deadlock, due to the delay that occurs before the cycle is detected. Although O2PL and CBR are similar designs, CBR clients send immediate block notifications to the server, and the server can perform all deadlock detection. While this approach has a higher message overhead, it is likely that deadlock detection is more important under high contention than saving messages. However, the opposite is true under low contention, where message costs are more important than deadlock detection.

In summary, we have identified three reasons for O2PL's high blocking times compared to C2PL and CBR: O2PL has a higher abort rate and therefore performs more wasted blocking, adding to its aggregate blocking time; O2PL is more likely to incur deadlocks compared to CBR and C2PL; O2PL uses distributed deadlock detection, while CBR and C2PL do not.

3.1.3 Invalidate vs. Propagate

For O2PL, when a committing transaction is about to update page P, all clients caching P are contacted as part of the commit. Therefore, O2PL can either cause these clients to discard page P, or it can install the new state of page P in the client caches. This tradeoff is known as the invalidate vs. propagate tradeoff. Page discards can be used prior to making a commit/abort decision, while page updates can only be performed once a commit decision has been forced to a transaction log at the server. Thus, the page update mechanism must carry out a two-phase protocol with the clients, while the page discard mechanism only requires one phase.

Franklin and Carey found that invalidation is almost always the better approach. They found one special-case workload where update proves to be more useful: a producer/consumer workload. (One client "produces" new updates and other clients "consume" these updates using only read accesses). They describe a dynamic heuristic that switches to using updates for this workload while using invalidations for the other workloads. This dynamic scheme only uses updates on commit; page discards are always used for aborts.

3.1.4 O2PL vs. AOCC

O2PL is an example of a scheme that trades off better low contention performance for worse high contention performance. The reasons for its poor high contention performance are described above. Since AOCC has better high contention performance than a callback locking that is itself an improvement over CBR, while O2PL has worse performance than CBR, relative high contention performance is clear.

With respect to low contention, message count predicts relative performance, and AOCC has a lower message count: it does not use one or two rounds of messages to synchronize with clients caches as part of its commit process. Moreover O2PL's message count is based in part on the number of clients that are caching shared pages, while AOCC's count is not; AOCC would show a larger improvement over O2PL for higher client cases.

3.1.5 Cache Maintenance Strategies

Another tradeoff discussed in the Franklin and Carey study is the impact of the cache maintenance strategy on *effective cache size*. Note that invalid pages are "useless" since new state is always fetched when an invalid page is accessed. Thus, if 10% of a client cache holds invalid pages, the effective cache size is 90% of the actual size. Franklin and Carey compare C2PL to both CBR and O2PL to demonstrate the impact of effective cache size.

C2PL uses a *passive* cache maintenance strategy: it detects and updates an invalid page only when necessary, *i.e.*, when an invalid page is accessed by a transaction. CBR and O2PL use a *proactive* cache maintenance strategy: pages are removed *before* they can become invalid. CBR is an *eager* proactive scheme: it uses callbacks to remove pages as soon as possible (prior to allowing a write). O2PL is a *lazy* proactive scheme: it integrates cache maintenance actions with commit processing, thus it performs page removal or page update as late as possible (for a proactive scheme). (The term "cache maintenance actions" covers both removal and update actions.)

One benefit of proactive cache maintenance is a higher effective cache size. Both CBR and O2PL "prune" the client cache of "useless" pages, providing free space for newly fetched state. LRU discards are rarely used, and valid pages are therefore rarely discarded. In contrast, C2PL can discard a valid page while invalid pages that are wasting cache space are retained. Franklin and Carey found that for workloads with low locality of reference, C2PL's passive strategy results in a poor cache hit ratio when compared to a proactive strategy as used by CBR and O2PL.

Avoidance-Based Strategies

Strategy	Description
Eager proactive	Synchronous consistency actions prior to allowing a write. (ACBL, CBR, CBA)
Lazy proactive	Consistency actions integrated with synchronous commit protocol. (O2PL)

Detection-Based Strategies

Strategy	Description
Eager reactive	Consistency actions triggered by a commit are sent in explicit messages. (NWL-Notify, Notify Locks)
Lazy reactive	Consistency actions triggered by a commit are piggy-backed on other messages. (AOCC, Cache Locks)
Passive	Consistency action only occurs when transaction accesses invalid cached state. (C2PL, NWL, Certification)

Figure 3-2. Cache Maintenance Strategies

AOCC uses a third maintenance strategy, *reactive* cache maintenance: pages are not discarded proactively prior to the commit of an update, but they are discarded soon after such a commit. (AOCC *reacts* to commits, which trigger cache maintenance actions.) As with the proactive schemes, a reactive scheme frees up space in the client cache, resulting in few LRU discards. Thus, reactive and proactive cache maintenance strategies result in very similar effective cache sizes.

In addition to AOCC, three schemes described below also use a form of reactive cache maintenance: Cache Locks uses piggy-backed invalidations, as in AOCC, while NWL-Notify and Notify Locks both send explicit update messages after each read-write commit completes. Thus, a commit can either trigger explicit message sends or the generation of messages that are to be piggy-backed on other messages sent to clients; we use the terms *eager reactive* and *lazy reactive* for these two cases.

Franklin and Carey refer to proactive schemes as *avoidance-based* schemes and to other schemes as *detection-based* schemes [23]. Avoidance-based schemes always have consistent caches, while detection-based schemes do not. (Detection-based schemes must be able to *detect* invalid pages). For locking schemes, including deferred locking schemes, this distinction is useful: avoidance-based locking schemes do not need to use lock requests for read accesses. For other performance issues, however, we find it useful to refine these categories by using the five cache maintenance strategies defined in this section. For example, of the five strategies, only passive cache maintenance can result in a low effective cache size. The five strategies are summarized in Figure 3-2, where we also show their relationship to the avoidance/detection distinction.

Note that each of the five strategies is different with respect to message use for cache maintenance. Eager proactive schemes use synchronous message exchanges with clients prior to each write. Lazy proactive schemes use synchronous message exchanges during a read-write commit. Eager proactive schemes send explicit asynchronous notification messages after each commit. Lazy proactive schemes piggy-back cache maintenance actions on other messages sent to clients. Passive schemes do not generate cache maintenance actions on either a per-write or per-commit basis; no attempt is made to keep caches from becoming arbitrarily out-of-date, and all maintenance actions occur due to accesses to invalid pages.

3.2 Wang and Rowe

Wang and Rowe [58, 57] studied a number of client-server schemes.

3.2.1 C2PL vs. CBA

Like Franklin and Carey, Wang and Rowe compare C2PL and CBA to determine the value of caching locks with pages. Wang and Rowe summarize this tradeoff as follows: CBA has better performance when there is locality of reference in the workload and contention is low; the two schemes have roughly equal performance when contention is low and locality of reference is also low; C2PL has better performance when contention is high. These general results are the same as the Franklin and Carey results for this tradeoff.

3.2.2 NWL vs. C2PL

NWL (No-Wait Locking) is the same scheme as C2PL, with two exceptions. First, while accesses to already-cached pages cause a lock request (with cached version number) to be sent to the server, as in C2PL, the client does not wait for a reply, but rather "optimistically" assumes the version is correct and a lock can be granted. Second, the server does not use lock-granting replies; it only sends a message in response to an "optimistic" lock request if the request fails and the transaction must abort.

NWL uses passive cache maintenance: cached pages are allowed to become arbitrarily out-of-date. An invalid page is only discovered when an access to this page causes an abort, and aborts cause page discards. The result is both a very high abort rate due to out-of-date accesses and high fetch costs during restart due to re-fetching discarded pages. (A key lesson from this study is that passive cache maintenance and optimism do not work well together.) The NWL abort mechanism as described by Wang and Rowe discards all pages accessed "optimistically" by an aborted transaction; it appears that all pages that were accessed out of the client cache rather than fetched during transaction execution are discarded. If this description is correct, an NWL abort unnecessarily discards a number of valid pages along with the invalid pages.

It initially seems counter-intuitive that NWL does not outperform C2PL under low contention, since NWL does not wait on lock replies. However, due to the high abort rate and the page discard mechanism just described, aborts add significant additional messages per commit. For example, NWL's message count is nearly the same as C2PL's message count when a 20% write probability is used.

In addition to high fetch costs during restarts, NWL's use of read-write locking contributes to its high restart costs. Since all accesses that do not cause an abort acquire a lock at the server, active transactions can block on other active transactions. Thus, both the blocking rate and the abort rate increase as contention increases. As discussed for O2PL, a scheme that can simultaneously incur both a high blocking rate and a high abort rate will have poor high-contention performance. (Gerson implemented no-wait locking for the Statice system [60]. His implementation switches to two-phase locking for "hot" pages, thus attempting to avoid a high abort rate [27]. No performance study has examined this adaptive variant of NWL.)

3.2.3 NWL vs. NWL-Notify

Wang and Rowe also experimented with NWL-Notify, a variant of NWL that uses eager reactive cache maintenance. As soon as an update commits, explicit notification messages containing page updates are sent to all clients currently caching the updated pages. The use of reactive cache maintenance significantly reduces the number of aborts due to out-of-date page accesses. However, the results show that NWL-Notify has very high server CPU costs due to sending updates; these costs offset the advantage of a lower abort rate, and NWL-Notify is not an improvement over NWL. Both variants of NWL perform poorly compared to the pure locking schemes used in the study (CBA and C2PL).

3.2.4 C2PL vs. Certification

Finally, a small comparison study is presented prior to presenting the results for the schemes discussed above. A pure optimistic scheme, Certification, is compared to C2PL. Wang and Rowe find that there is no difference between the schemes for a read-only workload or a very small number of clients (where contention is very low), but otherwise C2PL outperforms Certification; they therefore conclude that a purely optimistic scheme should not be used for their main study.

As with the other schemes discussed in this chapter, Certification discards modified pages on abort and re-fetches these pages during restart. In addition to this standard problem, there is design flaw specific to the Certification design that explains Certification's poor performance.

Certification is a purely optimistic scheme that uses the original Kung/Robinson certification check to validate commit requests [38]. As with NWL, passive cache maintenance is used. Wang and Rowe decided that the use of passive cache maintenance made it likely that out-of-date state would be accessed, thus a pre-access version number check is used prior to allowing a cached page to be accessed. Therefore, a round-trip exchange with the server is used on every initial page access.

The use of pre-access checks "throws away" one of key advantages of using optimism: an optimistic scheme can perform both read and write accesses on cached pages *without* using a round-trip exchange. Due to these checks, Certification has exactly the same message costs as C2PL when there are no aborts, and *higher* message costs when even a small number of aborts occur. (Note that Certification does not do early abort detection: a single restart doubles the number of messages used when compared to a first-run execution that commits.) Given the message costs of the two schemes, it appears that C2PL should outperform Certification except when Certification's abort rate is very low (or zero), in which case the two schemes should have roughly the same performance. This analysis is consistent with the results presented by Wang and Rowe.

3.3 Wilkinson and Neimat

Wilkinson and Neimat [61] performed the first client-server concurrency control study to examine schemes that use inter-transaction caching. Two such schemes are described, Cache Locks and Notify Locks.

The simulation results presented by Wilkinson and Neimat are hard to interpret: a different client cache model is used for the two schemes. One clear result is that sending explicit updates to clients on each commit is too expensive for general-purpose use. (This result is confirmed by Wang and Rowe in their comparison of NWL and NWL-Notify.) Given the different cache models and the fact that one scheme used very expensive update messages while one did not, no additional conclusions can be drawn with certainty with respect to other tradeoffs present in the two designs. Nevertheless, the two schemes are interesting with respect to examining concurrency control design issues. Thus, we discuss the two designs below. In addition, since each of these schemes has design elements that overlap with AOCC's design, we also compare the two designs to AOCC.

3.3.1 Cache Locks vs. Notify Locks vs. AOCC

Cache Locks is a deferred locking scheme that uses server-based locking. All lock requests for accesses to cached pages are deferred; read and write sets are sent with a commit request so that the server can attempt to acquire locks for these accesses. As in AOCC, lazy reactive cache maintenance is used: invalidations are piggy-backed on fetch and commit replies, and are used for both cache maintenance and early abort detection.

There are three key differences between Cache Locks and AOCC. First, for a commit request, Cache Locks uses a version number check at the server to detect any out-of-date accesses that were not caught by early abort detection at the client. AOCC uses invalid sets for this check, resulting in much lower space overheads. Second, Cache Locks incurs blocking and deadlocks while AOCC does not. Third, Cache Locks does not send updates in abort replies or use an undo log to restore modified pages.

Notify Locks is a backward-validation optimistic scheme that uses eager reactive cache maintenance: updates are sent to clients after each read-write commit. These updates are used for cache maintenance and early abort checking, as in Cache Locks and AOCC. Notify Locks and AOCC are similar in that they are both backward-validation schemes: commit-time validation compares the read set of a committing transaction to the updates of already-committed transactions. However, the two schemes use a very different implementation of this validation step. AOCC sends a transaction's read set to the server in a commit request, to allow the server to validate this set. Notify Locks does not send read sets in commit requests. Instead, it relies on the early abort checking performed at the client to validate a committing transaction's read set. Since the explicit update messages contain committed writes, the client can check its read set against these writes. The problem with this approach is that the full set of updates that must be used to validate a committing transaction's read set may not have arrived at the client prior to the generation of a commit request. In other words, the early abort checking done at the client did not necessarily do *all* of the required checking.

The Notify Locks scheme solves this problem by detecting the presence of unacknowledged update notifications. If unacknowledged updates exist, the server uses a special round-trip exchange with the client to determine whether these updates caused an abort at the client. Once this initial check passes, Notify Locks acquires write locks at the server; these locks are used to coordinate the transaction with other concurrent commit requests. During write lock acquisition, further update notifications can be sent to the client requesting the commit, thus after write locks are acquired another round-trip exchange with may be required. Therefore, Notify Locks can pay a high synchronization cost at commit time in order to avoid sending the read set of a transaction in a commit request. The overhead of sending a read set is (normally) not high enough to justify the Notify Locks approach.

Note that the write locks held at the server during commit processing are dropped *prior* to sending explicit update notifications to other clients, thus client caches can become out-of-date. If these write locks were retained until all these updates were acknowledged, Notify Locks could ensure that client caches are always be up-to-date and that transactions always read up-to-date state. In other words, while Notify Locks as designed using reactive cache maintenance, it could be changed to use proactive cache maintenance.

3.4 Parallel Database Concurrency Control

Shared-nothing parallel database systems are multiple processor systems where each processor has its own memory and disks (see, *e.g.*, [18]). For these systems, data is partitioned across the processors, and access to the data "owned" by a processor normally involves running a sub-transaction at that processor. Some shared-nothing systems allow processors to cache copies of non-local state indefinitely, *i.e.*, to "replicate" the data across more than one processor. In this case, the concurrency control scheme must adopt a cache maintenance strategy for the replicated data.

Client-server schemes that use inter-transaction caching also replicate data at the clients and must use a cache maintenance strategy. Thus, it is natural to look to studies of shared-nothing concurrency control schemes for insights relevant to client-server concurrency control tradeoffs. (For example, the client-server O2PL scheme discussed in Section 3.1 was studied because Carey and Livny found that a parallel shared-nothing version of O2PL showed some promise [12].)

However, the nature of these two types of systems is sufficiently different that results for one system do not apply to the other. In this section we briefly present the important differences and discuss their impact on concurrency control tradeoffs. In the following comparison, we use "CS" for "client-server" and "PD" for "shared-nothing parallel database."

1. PD systems execute many transactions at each node. These transactions share the memory, CPU and disk resources of the node. In contrast, a CS client executes only a few transactions at a time; the client memory and CPU are dedicated to a small number of transactions. This difference has an impact on restart costs. First, a CS client cache can normally retain the data used by active transactions, while a PD node may not have sufficient main memory to hold the pages touched by a large number of transactions. Thus, CS restarts have low fetch costs, placing a small load on the critical shared resource, the server. In contrast, PD restarts can have high disk costs; disks are an important shared resource. Second, note that an abort wastes CPU time at a node and

adds latency to the active transactions at that node. A CS abort adds latency to the active transactions at one client; these transactions represent a small fraction of all active transactions in the system. A PD abort adds latency to the many transactions active at a node, representing a much larger fraction of all transactions in the system. Thus, it is more important to avoid a high abort rate in a PD system.

2. For PD systems, we expect that most data conflicts occur between transactions running at the same node, as transactions are placed at nodes according to their expected data accesses, and moreover there are many transactions per node. For CS systems, we expect most data conflicts are inter-client conflicts. There is little or no concurrent activity at a client, and there is no attempt to place transactions according to expected accesses. All data accesses are remote accesses; a client does not "own" any data.

   Immediate local lock acquisition as used by O2PL is likely to reduce the abort rate for PD systems, but unlikely to do so for CS systems. At the same time, deferring remote write lock acquisition until commit is more likely to result in either blocking or aborts for CS systems as compared to PD systems.

3. A PD system has relatively few nodes compared to a CS system, and it is typical for a replicated data item to be placed at a small number of nodes (*e.g.*, 4). In a CS system, a large number of clients can be caching the same data item. For a PD system, the two-phase commit process used to coordinate transaction commit involves each node that executed a sub-transaction. This set of nodes often includes most or all of the nodes that are caching replicated items accessed by one of the sub-transactions. For a CS system, a distributed commit requires the involvement of participating servers; clients do not need to be contacted unless proactive cache maintenance is used during commit. The PD version of O2PL can piggy-back deferred write requests on the required two-phase commit protocol, while the CS version of O2PL must contact a potentially large set of additional nodes (participating clients) during phase one of the commit protocol.

In summary, locking and proactive cache maintenance have lower costs for a PD system than they do for a CS system, while transaction restarts have higher costs for a PD system than they do for a CS system. Some designs will work well in both types of systems, while other designs will only work well in one system or the other.

Chapter 4

Experimental Framework

To use simulation to study different client-server concurrency control schemes, it is necessary to model all of the following elements: client, server, disk, database, and transactions. We refer to the server, network, disk, and database models collectively as the *system model*, while the *workload model* captures the way that transactions run against the database, and the nature of these transactions.

Each model has a set of *parameters*, to allow us to vary, *e.g.*, the speed of the network, or the sharing pattern exhibited by transactions. With respect to the system model, we have chosen two groups of system parameter settings that we refer to as the CURRENT and FUTURE settings. These settings are meant to model present-day systems and near-future systems, respectively. Section 4.1 describes our system model and summarizes the CURRENT and FUTURE settings.

With respect to the workload model, we have chosen six groups of workload parameter settings that produce transactions with different sharing patterns and different contention levels. We refer to each such group of settings as a *workload*. Section 4.2 describes our workload model and the six workloads that we use in this dissertation.

This chapter summarizes our models and settings, giving the reader all details required for understanding our experimental setup. The interested reader can refer to Appendix A for a more detailed discussion of how we chose our CURRENT and FUTURE settings.

Simulator Genesis

As part of our research, we developed a new discrete-event simulation library, including modules for the resources discussed in this chapter (client, server, network, disk) and a module for workload generation. These modules are written in `C++`.[1]

The design of our system model is based in part on the the page-based simulation work of Franklin and Carey [23, 24] and the adaptive-granularity locking work of Carey, Franklin, and Zaharioudakis [10] (CFZ). As discussed in Section 4.2, our choice of workloads is motivated by the workloads used in these other studies.

[1] The core discrete-event simulation engine was jointly developed with Sanjay Ghemawat, while the higher-level modules were developed by the author.

Prior to the publication of the CFZ work we implemented a simulation system that allowed us to study both purely page-based and purely object-based concurrency-control schemes. After deciding that adaptive-granularity locking was the right locking scheme to use in our comparison study, we changed our client cache model to the model described in Chapter 2 and implemented both ACBL and AOCC on top of the resulting simulation framework. Section 4.3 concludes this chapter by discussing the differences between our own simulation framework and that used in the CFZ work.

4.1 System Model and Settings

Figure 4-1 summarizes the set of parameters that make up the system model, along with the settings that we use for these parameters. The first and second columns give the settings for the CURRENT and FUTURE system configurations, respectively. CURRENT system results are used in Chapter 5, where we present most of our key findings. Chapter 6 describes experiments that cover other regions of the overall parameter space. It includes experiments that use the FUTURE settings, as well as a set of sensitivity analysis experiments that examine the impact of varying a particular parameter (or pair of related parameters). The third column in Figure 4-1 gives the parameter ranges used in the sensitivity analysis experiments.

The following subsections describe different components of the system model and give the CURRENT and FUTURE parameter settings for each case.

4.1.1 Database

Each simulator run described in this thesis accesses a small subset of the database pages. We call this set of accessed pages the *working set* of the simulator run. The six workloads described in the next section have working set sizes between 1250 and 1300 pages. Our results apply to larger working set sizes: this relatively small size is used to make our simulations tractable. The absolute size of the working set is less important than the ratio of client or server cache sizes to total working set size. Cache size parameters are discussed later in this section.

We assume that objects are small, and that multiple objects fit on a page. For simplicity, we use fixed object and page sizes. The working set is thus described by the *object size*, *objects per page*, *page size*, and *working set size* parameters. This data model is sufficient for our purposes, as it allows us to study a concurrency control scheme's ability to handle small objects and fine-granularity sharing.

This set of parameters allows for per-page overhead: the product of *object size* and *objects per page* can be less than *page size*. For both the CURRENT and FUTURE systems, we use 100 byte objects, 40 objects per page, and 4 KB pages, thus each page has 96 bytes of overhead.

Database Parameters			
Parameter	CURRENT	FUTURE	Chapter 6 Exp.
Object size	100 bytes	100 bytes	—
Page size	4 KB	4 KB	—
Objects per page	40	40	—
Working set size (pages)	1250–1300	1250–1300	—
Server Parameters			
Parameter	CURRENT	FUTURE	Chapter 6 Exp.
Server cache size	50% of WS	50% of WS	10%–100% of WS
Modified object buffer size	50% of WS	50% of WS	–
Server CPU speed	50 MIPS	200 MIPS	30–400 MIPS
Register / Unregister	300 instr.	300 instr.	—
Validation time per object	0–300 instr.	0–300 instr.	—
Cost of deadlock detection	0 instr.	0 instr.	—
Cache lookup	300 instr.	300 instr.	—
Client Parameters			
Parameter	CURRENT	FUTURE	Chapter 6 Exp.
Number of clients	1 – 24	1 – 24	—
Client cache size	25% of WS	25% of WS	5%–50% of WS
Client CPU speed	25 MIPS	100 MIPS	15-200 MIPS
Cache lookup	300 instr.	300 instr.	—
Read think time	50 instr./byte	50 instr./byte	25–1000 instr./byte
Write think time	100 instr./byte	100 instr./byte	50–2000 instr./byte
Inter-trans. think time	0 instr.	0 instr.	—
Network Parameters			
Parameter	CURRENT	FUTURE	Chapter 6 Exp.
Network bandwidth	80 Mbps	160 Mbps	4–800 Mbps
Fixed network cost	6000 instr.	3000 instr.	6000–250 instr.
Variable network cost	7168 instr./KB	2048 instr./KB	7168–128 instr./KB
Disk Parameters			
Parameter	CURRENT	FUTURE	Chapter 6 Exp.
Disk setup cost	5000 instr.	5000 instr.	—
Slow disk bandwidth	3322 μsecs/KB	2580 μsecs/KB	5000–500 μsecs/KB
Fast disk bandwidth	1288 μsecs/KB	990 μsecs/KB	1900–190 μsecs/KB
Number of server disks	4	8	—

Note: WS = Working set size

Figure 4-1. Summary of System Parameter Settings

4.1.2 Processors

A client or server is modeled as a simple processor with a microsecond granularity clock. This clock advances as events running at the processor make "charges" against it. Charges in our model are specified using instruction counts, and are converted to μsecs based on the speed of the processor; the client and server speeds are specified in MIPS, or millions of instructions per second. For the CURRENT system, client processors run at 25 MIPS and the server at 50 MIPS. For the FUTURE system, client processors run at 100 MIPS and the server at 200 MIPS.

Events queued at a processor are executed in start-time order, where event E's start time is the earliest time it should run. *E.g.*, for each message sent to the server, the network module computes a message arrival time and adds a message-reception event to the server's queue with start time equal to the computed time. Each processor's time advances as charges are made. Time at processor P also advances between event executions if the earliest start time in *any* of the processor queues is later than P's current time. An event never begins executions earlier than its requested start time, but it can begin later than its start time if the processor is heavily utilized.

A voluntary-suspension model is used; we did not feel the need to implement a preemptive scheduler. All the client and server events that we use have short run-lengths; an event yields the processor at any point that a significant delay would occur (such as the need to wait for a disk operation to complete).

There are a number of parameters used to specify the instruction charges for actions that occur at the client or server. Some of these charges are described in the "client" and "server" subsections below, while charges related to network or disk usage are discussed in the "network" and "disk" subsections. For cache lookup, which occurs at both client and server, we always use a 300 instruction charge.

4.1.3 Server

For schemes that either lock (unlock) objects or pages, or that register (unregister) the fact that a client is caching (has stopped caching) a particular object or page, a charge is used at the server for each lock, unlock, register, or unregister. If both locking and registration are used, as in ACBL, then only a single charge is used for a combined lock/register or unlock/unregister pair. We use 300 instructions for cache lookup and register/unregister for both the CURRENT and FUTURE systems.

AOCC is charged for performing commit-time validation: each read-set entry is searched for in the client's invalid set, and the cost of each search is based on the size of the invalid set. The following charges are used for both the CURRENT and FUTURE systems. For small invalid sets, the cost is 10 instructions per invalid set entry; for large invalid sets, the cost is a fixed 300 instructions. These charges model an adaptive approach where a linear search is used for small invalid sets and a hash table lookup is used for large invalid sets. (Note that there is no validation cost if the client's invalid set is empty.)

To ensure that our comparison study is not biased against locking schemes, we intentionally underestimate those charges that only apply to lock-based approaches. Thus, while our model has a parameter for deadlock detection, we always use a zero setting for this parameter (*i.e.*, deadlock detection is “free”).

The final two server parameters specify the server cache size and the *modified object buffer* size as a fraction of the working set size. For both the CURRENT and FUTURE systems, these data structures are sized at 50% of the working set size. The use of the MOB is described in Section 2.1.

4.1.4 Client

The number of clients in a given system is a system model parameter. We ran experiments using 1–24 clients.

Each client has a cache whose size is specified as a percentage of the overall working set size. For both the CURRENT and FUTURE systems, client cache sizes are 25% of working set size. These caches are page-based, but support object lookup, and individual objects can be marked as unavailable, as described in Chapter 2. We charge a cache lookup (300 instructions) for each object access, for each processed callback request (for ACBL), and for each processed invalid set entry (for AOCC).

For each read or write access that the client performs, the client CPU is charged a fixed number of instructions, as specified by the *read think time* and *write think time* parameters, respectively. (These think times are specified in instructions/byte; the total instructions charged thus depends on the object size.) For both the CURRENT and FUTURE systems, we use 50 instructions/byte for reads and 100 instructions/byte for writes. Objects are 100 bytes, thus we use 5,000 and 10,000 instructions per read and write access.

The model also has a *transaction think time*, a CPU charge that can be used to add a delay between the successful completion of one transaction and the start of the next. This parameter was set to zero for all experiments reported in this thesis.

4.1.5 Network

Rather than using a detailed model of a particular network, such as Ethernet, FDDI, or ATM, our network model is more abstract. We divide 1-way message latency into three components: processor cost for sending the message, transmission across the “wire”, and processor cost for receiving the message.

The processor-based cost of sending or receiving a message is modeled as a linear function of the message size. For both message send and receive, the appropriate processor is charged a fixed number of instructions, plus a variable number of instructions per KB. Since these fixed and variable system parameters are specified in terms of instructions, network latencies will go down if processor speeds are increased. (This effect occurs in real systems; software overheads are a large fraction of total message latencies.)

The wire itself is modeled as a shared FIFO queue with fixed bandwidth (specified in Mbps). The “wire time” for a message is based on the message length and the bandwidth. Note that message latency for an unloaded network and unloaded sender and receiver is just the sum of the send CPU time, the wire time, and the receive CPU time. Even in this "unloaded system" case, a round-trip message exchange has a significant cost (*i.e.*, compared to the cost of locally accessing an object), and avoiding synchronous round-trip requests can lead to significant savings.

This network model is a first-order approximation of a shared-resource network, such as an Ethernet or a token-passing ring in which all communication uses a single shared resource.

We ignore the second-order effects associated with network contention; our model fails to approximate a real network as it nears saturation. Fortunately, when we run our chosen workloads using the CURRENT and FUTURE system parameters, the network does not saturate.

For the CURRENT system, network speed is 80 Mbps, fixed CPU send/receive cost is 6000 instructions, and variable CPU cost is 7168 instructions/KB (7 instructions/byte). For the FUTURE system, network speed is 160 Mbps, fixed CPU send/receive cost is 3000 instructions, and variable CPU cost is 2048 instructions/KB (2 instructions per byte).

Under this network model, charges assigned to the sending and receiving processors are not allowed to overlap any of the wire latency. In real systems, a large message can start flowing out over the wire before the sender has fully processed the entire message, and at the other end the receiver can start reading the message before all of the message has crossed the wire. This partial mismatch between real systems and our network model is discussed in Appendix A, where we describe our reasons for choosing the above network parameter settings.

4.1.6 Disk

The database pages are stored on server disks; the number of disks is a system parameter. Pages are statically assigned to disks using a simple mapping that is not tied to any specific workload. (Pages are assigned to disks using the simple formula `page_number modulo number_of_disks`.)

The server is charged a fixed number of instructions each time it initiates a disk read or write access. This charge should reflect all the CPU costs associated with a single disk access. Once the charge completes, a disk access is "initiated." The disk itself is modeled as a shared FIFO resource; accesses use up this resource in the order they are initiated.

Like a processor, a disk is a time-based resource; each access is scheduled at its initiation time, and makes a "charge" that advances the disk's clock. When this charge completes, a completion event is scheduled at the server at the disk's current clock time.

Disk reads due to fetch requests are random with respect to the current disk head location, while disk operations used to install committed updates are intelligently scheduled by the installation thread (see Section 2.1). Therefore, we use two bandwidth parameters: slow and fast. The slow bandwidth represents the average bandwidth one expects when a series of random page-sized reads is issued, while the fast bandwidth represents the bandwidth one expects when a series of page accesses is issued by an intelligent scheduling algorithm that selects accesses from a large pool of possible accesses.

Appendix A discusses disk scheduling algorithms. It also describes how we chose the following settings for the CURRENT and FUTURE systems. We use the same CPU setup cost for both systems: 5000 instructions. (This is the same charge used in [10].) Of course, FUTURE CPU speeds are 4 times faster, thus FUTURE disk setup times are 4 times faster. The CURRENT system has four disks, where each disk has a slow access time of 3322 μsecs/KB and a fast access time of 1288 μsecs/KB. The FUTURE system has both more disks and faster disks: there are eight disks, where each disk has a slow access time of 2580 μsecs/KB and a fast access time of 990 μsecs/KB.

4.2 Workload Model and Settings

This section describes our workload model and the six workloads used in our simulation experiments. Section 4.2.1 describes the parameters of our general workload model, while Section 4.2.3 describes the six workloads used in this thesis.

4.2.1 Workload Model

Each transaction consists of a sequence of read and write accesses, as determined by the *workload generator*, a run-time component of the simulator. This generator is configurable, allowing us to simulate a number of interesting workloads that exhibit different sharing patterns and different levels of contention. Figure 4-2 shows the full set of parameters for the workload generator.

Each client executes one transaction at a time. Starting with the first access in the transaction sequence, it performs each access in turn, until either an abort occurs or all accesses have been performed (in which case it attempts a commit). If a transaction aborts, it is *restarted*: the client begins transaction execution again, starting with the first access of the same access sequence. (The access sequence can change during a restart execution; see the description of the *restart change probability* parameter below.) If the transaction commits, the workload generator is used to create a new transaction, and the client then executes this new sequence of accesses.

Clustering and Write Probabilities

We assume that the objects stored in the database have been placed onto pages by some *clustering process* that uses information about common access patterns or the links between objects to determine groups of objects that are likely to be accessed together [3, 14, 19, 51, 54, 55]. Clustering is important for performance; the objects on a page are read from disk as a group and are sent to a client as a group. We model the *quality* of clustering by specifying a range of cluster sizes. The average cluster size determines the average number of objects accessed per page.

To construct a transaction, the workload generator goes through an iterative process, where each step adds a new cluster of accesses to the access sequence. A cluster

Workload Generator	
Parameter	**Units**
Transaction accesses	(range)
Pages in shared-1 region	pages
Pages in shared-2 region	pages
Pages per private region	pages
Total working set size	pages
Shared-1 access prob.	percentage
Shared-2 access prob.	percentage
Private access prob.	percentage
Other-region access prob.	percentage
Shared-1 cluster write prob.	percentage
Shared-2 cluster write prob.	percentage
Private cluster write prob.	percentage
Other-region cluster write prob.	percentage
Shared-1 object write prob.	percentage
Shared-2 object write prob.	percentage
Private object write prob.	percentage
Other-region object write prob.	percentage
One cluster per page	boolean
Shared-1 cluster size	(range)
Shared-2 cluster size	(range)
Private cluster size	(range)
Other-region cluster size	(range)
Restart change prob.	percentage
Percent forced read-only	percentage

Figure 4-2. Workload Parameters

is chosen by first picking the page to access and the cluster size K; K object accesses are then selected randomly from the page. (Selecting the page and the cluster size are discussed in detail below.)

Normally, the generator does not select the same page twice when generating a transaction; the default is to use at most one cluster per page. This behavior can be overridden by setting the *one cluster per page* parameter to "false"; in this case a page can be selected more than once, and each time it is selected a new cluster size K is chosen and K new accesses are added to the transaction sequence.

The generator never picks the same object twice, even when multiple clusters per page are allowed. This invariant simplifies concurrency control scheme implementations. For locking schemes, it ensures that an object-level lock upgrade is never required due to a read access followed by a write access for the same object. Lock upgrades are a source of additional deadlocks and aborts; under a workload model that allows multiple accesses to the same object, ACBL would have lower high-contention performance.

After the access type and page are chosen, the workload generator decides whether write accesses will be allowed for the new cluster, using the *cluster write probability*. If writes are allowed, then for each access the workload generator decides whether the access is a write, using the *object write probability*. The *net object write probability* is the product of the cluster write probability and the object write probability. For example, with a cluster write probability of 50% and an object write probability of 20%, the net object write probability is 10%.

This two-level approach to specifying write probabilities can be used to model the fact that the clustering process that places objects on pages may cluster objects that tend to be read-only on some pages, while placing objects that tend to be updated on other pages. A low cluster write probability means that the clustering process has done a good job grouping objects according to write access patterns; only a few of the pages accessed by a transaction will be updated.

Regions and Access Types

Each workload uses a set of pages referred to as its *working set*. This set is modeled as a sequence of pages, where each page contains a fixed number of objects. A set of four workload parameters is used to divide the working set into disjoint *regions*. Each region is a contiguous sequence of pages; region boundaries fall on page boundaries. A number of different regions can be specified. There are two named regions, called *shared-1* and *shared-2*; there is a parameter for each region specifying the number of pages assigned to it. (Assigning a zero size to a region omits the region from the workload). In addition, there is a parameter for specifying per-client *private regions*. If this parameter has non-zero value K, then there is a set of private regions of K pages each. The Nth client is said to *own* the Nth private region. Finally, the total number of pages in the working set is also a parameter. Thus, in addition to the shared-1, shared-2, and per-client private regions, there can be a final contiguous sequence of pages not assigned to any of these regions; we call this set of pages the *leftover region*.

Our model identifies four types of object accesses for a given client C: *shared-1* and *shared-2 accesses* are simply accesses to these named regions; *private accesses* are accesses to C's own private region; *other-region accesses* are accesses to any other region: these accesses can go to another client's private region or to the leftover region, but cannot go to shared-1, shared-2, or C's own private region.

As shown in Figure 4-2, each access type has a set of parameters associated with it. In addition to specifying how the total object accesses should be distributed across the four access types, one must specify three things for each access type: a cluster size range, a cluster write probability, and an object write probability. Thus, the different access types can have different cluster sizes, or different read-write characteristics.

Generating a Transaction

To produce a new access sequence for a transaction, the workload generator first chooses a transaction length; the length is chosen in uniform fashion from a range that is specified by minimum and maximum lengths. The generator then goes through a series of cluster-adding steps, adding the accesses for one cluster at a time until the desired number of accesses has been generated.

Adding a Cluster For each type of access, there are parameters for specifying an access probability and a cluster size range (a minimum and maximum number of object accesses per cluster). The access probabilities are specified as a percentage of the total accesses; the sum of the shared-1, shared-2, private, and other-region access probabilities must be 100%.

Given these parameters, four *access weights* are computed for the four access types. For access type A, let P_A be the object access probability and C_A be the average cluster size from A's cluster size range. Access weight W_A is equal to P_A/C_A. The workload generator randomly chooses the access type of the next cluster to add to the transaction using a random choice that is weighted by the four access weights.

The access weights are only necessary because the access breakdown is specified in terms of desired *object access* probabilities, while we add entire clusters at a time. For example, suppose we have a workload that only uses two access types, shared-1 and shared-2, where the specified *object* access probabilities are 50% for each region, the average shared-1 cluster size is 10, and the average shared-2 cluster size is 20. If the workload generator simply selects shared-1 clusters half the time and shared-2 clusters half the time, we end up with twice as many shared-2 object accesses as shared-1 object accesses. Using the access weights of 50/10 and 50/20 for shared-1 and shared-2, respectively, shared-1 clusters are selected twice as often, on average, which produces the desired 50–50 object access breakdown.

Choosing an access type defines one or more regions within the working set that can be accessed. A page from one of these regions is selected in uniform fashion. (As discussed above, normally a page is only selected once per transaction, but this behavior can be overridden.) The access type's cluster size range is used to choose a cluster size S. S objects are then randomly selected from the page and added to the transaction sequence. For each access, if writes are allowed, a random read/write choice is made based on the object write probability.

We have now described the workload model and the details of transaction generation. Two additional workload generator parameters can be used to modify this behavior.

Restart Change Probability

When a transaction fails and restarts, on re-execution it normally uses the same sequence of accesses. However, this tends to favor AOCC, since AOCC does a good job "pre-loading" the client cache with objects that will be accessed again on restart. In a real system, even if the same transaction code is re-run when a failed transaction is restarted, the objects that are accessed may be different from the first execution. For example, if the code traverses a data structure, that structure may have changed since the first (failed) execution of the transaction. To allow us to examine the impact of such changes, we added the *restart change probability* parameter to the workload generator.

It is easiest to describe the effect of the restart change probability in an operational manner. The simulator keeps a version number for each object at the server, which is incremented each time the object is updated by a committed transaction. It also keeps version numbers in client caches, indicating which version of the object is cached at the client. Moreover, it records a version number sequence at a client as the client executes a transaction: for each object accessed, the cached version number is appended to the sequence. (Note that these version numbers are used by the simulator, not by the concurrency control schemes.)

The version number sequence from a failed transaction is retained when the transaction restarts. During the re-execution, after completing an object access, the current cached version number for the accessed object is compared to the retained version number from the previous execution (the one that failed). If the object has changed its value since the last time it was accessed, a random choice is made, weighted by the restart change probability, as to whether the remaining accesses in the transaction sequence should be replaced with new accesses. If the decision is to do a replacement, the workload generator replaces the remaining accesses with new accesses; a suffix of the transaction sequence is changed, but the transaction length is kept the same, and the workload generator still ensures that there are no duplicate object accesses in the full sequence. Execution then proceeds with the accesses in the new suffix.

If the random choice does not end up causing a suffix replacement, the client continues with the next access in the original transaction sequence. If another object access occurs where the object has changed its value, another random choice is made; there can be multiple points during transaction re-execution where a suffix replacement is possible. Since each choice is random (weighted by the restart change probability), the original sequence may end up getting used, even if some objects have changed. Once a suffix replacement has occurred, no further version number checking is done; only one suffix replacement can occur per abort.

Percent Forced Read-Only

The *percent forced read-only* parameter is a “knob” that can be used to vary the read-only/read-write mix. Except for one experiment in Chapter 6, this knob is not used in this thesis (the parameter is usually set to zero). If this parameter is set to X%, the workload generator randomly “forces” X% of all transactions to be read-only. A forced-read-only transaction is generated in the same way as other transactions, using the process described above, except that the cluster and object write probabilities are simply ignored, and all accesses produced are read accesses. Thus, the forced-read-only transactions exhibit the same average length and access pattern as the normal transactions they are mixed with.

4.2.2 Motivating the Six Workloads

We chose six different workloads for our experiments. This section describes our reasons for choosing these workloads; the next section describes each workload.

For reasons of continuity, we use workloads that others have used to study client-server systems. In particular, the client-server concurrency control studies of Franklin and Carey [10, 23, 24] are well known in the database research community; four of our workloads are derived from the workloads used in their studies: UNIFORM, PRIVATE, HOTCOLD, and HICON. These workloads began as page-based workloads [23, 24]. While transactions now consist of object accesses, we have preserved the same sharing patterns and the same number of average page accesses per transaction.

These four workloads capture important sharing patterns, as we describe below. One access pattern that is not present is the use of a small uniformly shared region that is (1) not a significant fraction of the overall working set size, and (2) not as frequently accessed as the remainder of the working set. Such a region represents a small set of data structures that are important to the overall application but are not the most frequently accessed objects. Some examples include: work queues; the upper levels of tree-structured data structures; an aggregate value such as the overall number of parts used in a design project.

We decided to model two new shared sets that are infrequently accessed, which we call SMALL and TINY:

- SMALL is a 50 page uniformly shared region that is accessed 10% of the time. The net object write probability for SMALL objects is 20% or less, resulting in moderate contention over this region.

- TINY is a 1 page uniformly shared region that is accessed only 2% of the time; TINY is a 40 object region. The net object write probability for TINY objects ranges from 50% to 100%, resulting in high contention over this region.

Each regions is added to one of the above workloads, creating two new workloads.

The first new workload is called SMALL+HOTCOLD: the 50 page SMALL region is added to HOTCOLD. We chose to add SMALL to HOTCOLD because it creates a moderate contention workload with both uniform and skewed sharing patterns. Real applications will exhibit both these sharing patterns; we consider SMALL+HOTCOLD to be the best representative of a "real" application. For this reason, SMALL+HOTCOLD is our standard choice for the sensitivity analysis experiments presented in Chapter 6; the other workloads are also used in a number of the Chapter 6 experiments.

The second new workload is called TINY+PRIVATE: the 1 page TINY region is added to PRIVATE. TINY contains "hot spot" objects (objects under high contention). We chose to add TINY to PRIVATE to isolate the impact of hot spot accesses. Without TINY, PRIVATE has no contention (no blocking and no aborts). Under TINY+PRIVATE, all blocking and all aborts are due to TINY's hot spot objects.

4.2.3 Workload Descriptions

This section describes the workloads in detail; after discussing some elements that are common to all the workloads, it presents each workload in turn. The parameter settings for the workload generator used to produce the six workloads are summarized in Section 4.2.4.

Common Elements

All of the workloads (except for UNIFORM) have access patterns that use one access type 80% of the time; the remaining 20% of the accesses are sometimes just one type of access, and are sometimes broken down further (if more than two access types are used). This skewed access pattern is common in many database studies; an 80–20 skew seems to be the *de facto* skew for concurrency control studies.

However, there are two kinds of 80–20 workloads. For earlier studies that were not client-server studies, such as studies of single-server databases or multi-server databases with shared disks, all clients in the system were treated uniformly: 80% of all accesses went to a single shared region. HICON is an example of this form of 80–20 workload. In contrast, with client-server systems, we are interested in the effects of inter-transaction caching at the clients. Thus, workloads where each client accesses its own "private" region 80% of the time became interesting. HOTCOLD, PRIVATE, SMALL+HOTCOLD, and TINY+PRIVATE are all examples of this form of 80–20 workload.

Most of the workloads use a cluster size range of of 5–15 (the average cluster size is 10), and a single cluster per page is the default. (The except is TINY+PRIVATE, where a cluster size of 2 is used for accesses to the TINY region, and more than one such cluster is allowed.) Thus, on average, one quarter of the objects on a page are accessed when a page is touched, and the number of pages touched is equal to 1/10 of the average transaction length. Accessing 10 out of 40 objects means that the clustering algorithm is doing a fairly good job. Better clustering is generally hard to achieve. For example, a study by Tsangaris and Naughton found that most practical clustering algorithms provide poor clustering [55], while algorithms that provide good clustering were prohibitively expensive.

Default Write Probabilities. For regions that are updated, the default net write probability is 10%. Average write clustering is used for this default, *i.e.*, cluster write probability is 50% and object write probability is 20%). Two regions use the reverse case: a 20% cluster write probability 50% object write probability are used for the "cold set" accesses used by the HICON workload and for the private accesses used by the TINY+PRIVATE workload. In each case, these regions were used as low-update "cold" regions that are paired with high-update "hot" regions (the "hot set" in HICON and the TINY set in TINY+PRIVATE, respectively). We used a low cluster write probability to produce a very low page update frequency for the cold regions. There is just one region with a default net write probability greater than 10%: the TINY region has a default net write probability of 50% (100% cluster write probability, 50% object write probability).

These default write probabilities are used for the main experiments reported in Chapter 5. For all six workloads, we also studied many other combinations of cluster and object write probabilities. The impact of varying the cluster and object write probabilities is explored in Section 6.2.1.

UNIFORM Workload

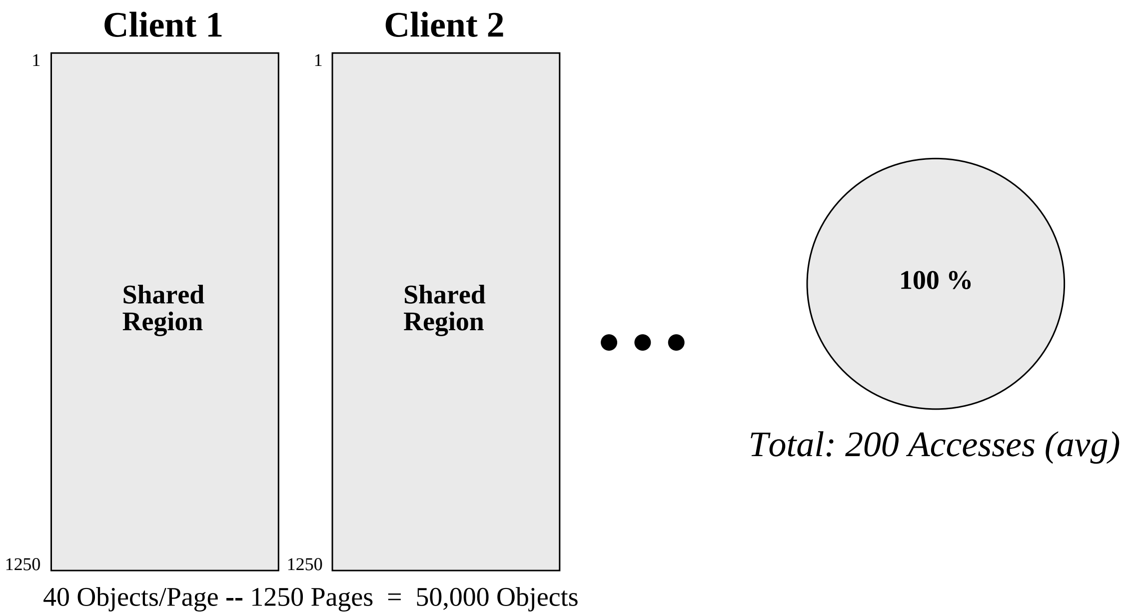


Figure 4-3. UNIFORM Workload

The UNIFORM workload is an object-based version of the page-based UNIFORM workload from [23, 24]. Its access pattern is depicted in Figure 4-3, which shows that every client views the working set in the same way: as a single 1,250 page shared region. In this workload transactions average 200 object accesses (20 pages are touched). Although clustering within pages is used, across a large number of transactions, all objects in the working set are equally likely to be accessed.

UNIFORM is used to examine how concurrency control schemes perform when there is little or no locality of reference across transaction boundaries.

PRIVATE Workload

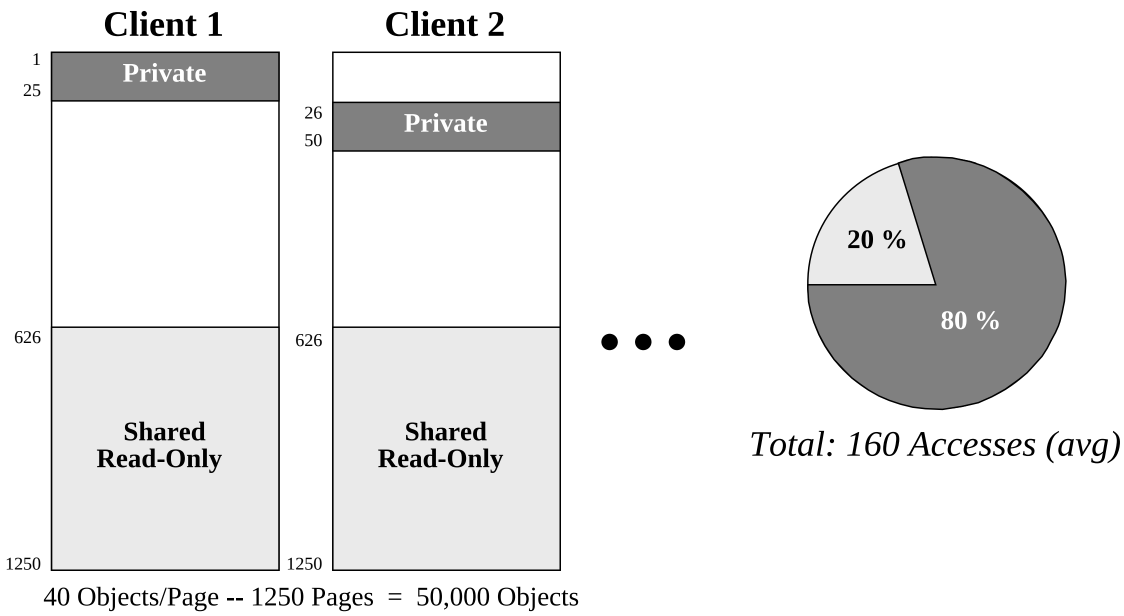


Figure 4-4. PRIVATE Workload

The PRIVATE workload is an object-based version of the page-based PRIVATE workload from [23, 24]. In this workload transactions average 160 object accesses (16 pages are touched). Each client has its own 25 page private region that it accesses 80% of the time; the other 20% of the time, it reads objects from a large (625 page) read-only region shared by all the clients. This access pattern is depicted in Figure 4-4.

PRIVATE was designed to model an environment where the users of the application "divide" the work in advance, in a way that avoids conflicts. One example would be a large CAD system, where users agree in advance to work on disjoint parts of the design; each user updates a disjoint region, while all the users access a shared library of components in read-only fashion.

HOTCOLD Workload

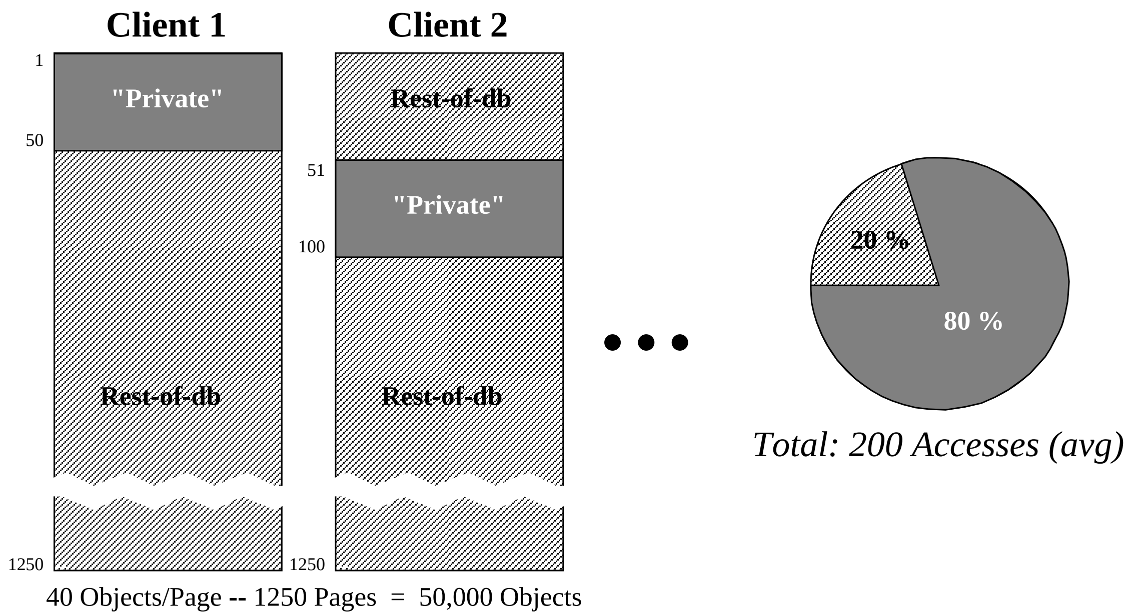


Figure 4-5. HOTCOLD Workload

The HOTCOLD workload is an object-based version of the page-based HOTCOLD workload from [23, 24]. Under HOTCOLD clients mostly access their own data but sometimes access each other's data. The access pattern of HOTCOLD is depicted in Figure 4-5. Transactions average 200 object accesses (20 pages are touched). There are 1,250 pages in the working set. Each client has its own 50 page "private" region; any remaining pages make up a "leftover" region whose size depends on the number of clients. *E.g.*, with 20 clients, 1,000 pages are used for private regions, leaving a 250 page leftover region. Each client performs private accesses 80% of the time and other-region accesses 20% of the time. An other-region access goes anywhere except a client's own private region: it can go to another client's private region, or it can go to the leftover region.

HOTCOLD was designed to test the performance of client-server concurrency control schemes that do inter-transaction caching. Each client has an affinity for its own objects, so inter-transaction caching should provide significant performance benefits. The other-region accesses cause moderate read-write sharing between clients; HOTCOLD is useful for examining low-contention performance.

SMALL+HOTCOLD Workload

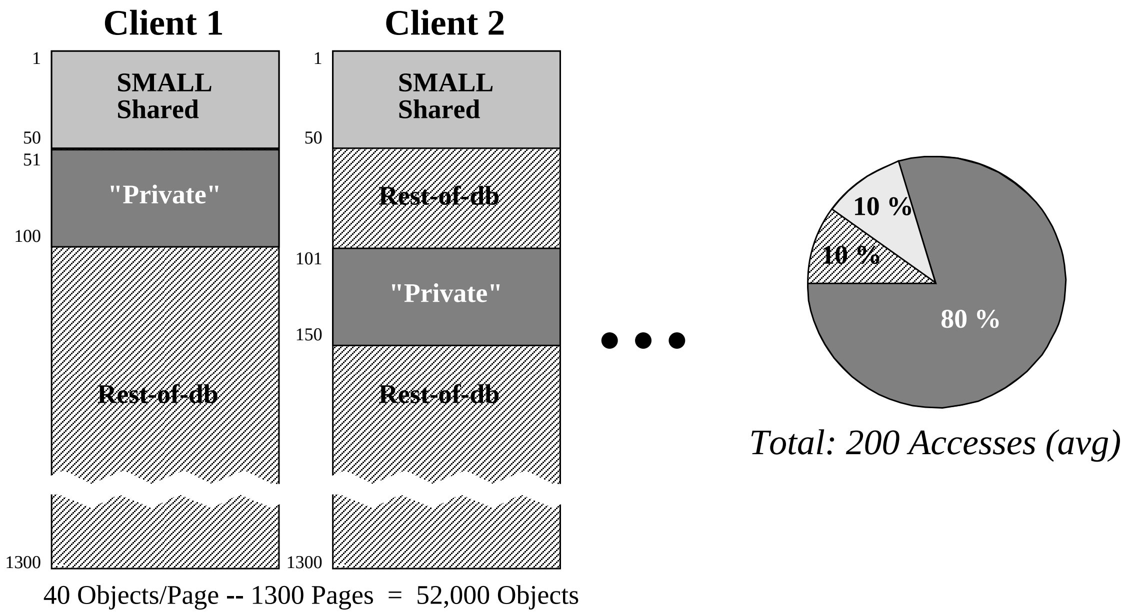


Figure 4-6. SMALL+HOTCOLD Workload

SMALL+HOTCOLD is like HOTCOLD except the SMALL region, a 50 page uniformly-shared region, has been added. This workload is depicted in Figure 4-6. Transactions average 200 object accesses (20 pages are touched). 10% of client accesses are SMALL accesses, 80% are private, and 10% are other-region accesses. Other-region accesses go to anywhere in the working set except SMALL and the client's own private region. Like HOTCOLD, SMALL+HOTCOLD has a leftover region whose size depends on the number of clients. Note that more accesses go to SMALL than to the leftover region; there should be little contention over the leftover region, and moderate contention over SMALL.

Of the six workloads, SMALL+HOTCOLD has the richest mix of sharing patterns. For this reason, SMALL+HOTCOLD is the primary workload used for sensitivity analysis (described in Chapter 6).

The SMALL region is a good challenge for caching client-server concurrency control schemes. The caching of SMALL pages should provide performance benefits. However, the update frequency for SMALL pages will increase with the number of clients, thus the actual benefit derived from caching will depend on the effectiveness of cache maintenance (*e.g.*, piggy-backed invalidations as used by AOCC) or cache consistency (*e.g.*, callbacks as used by ACBL).

HICON Workload

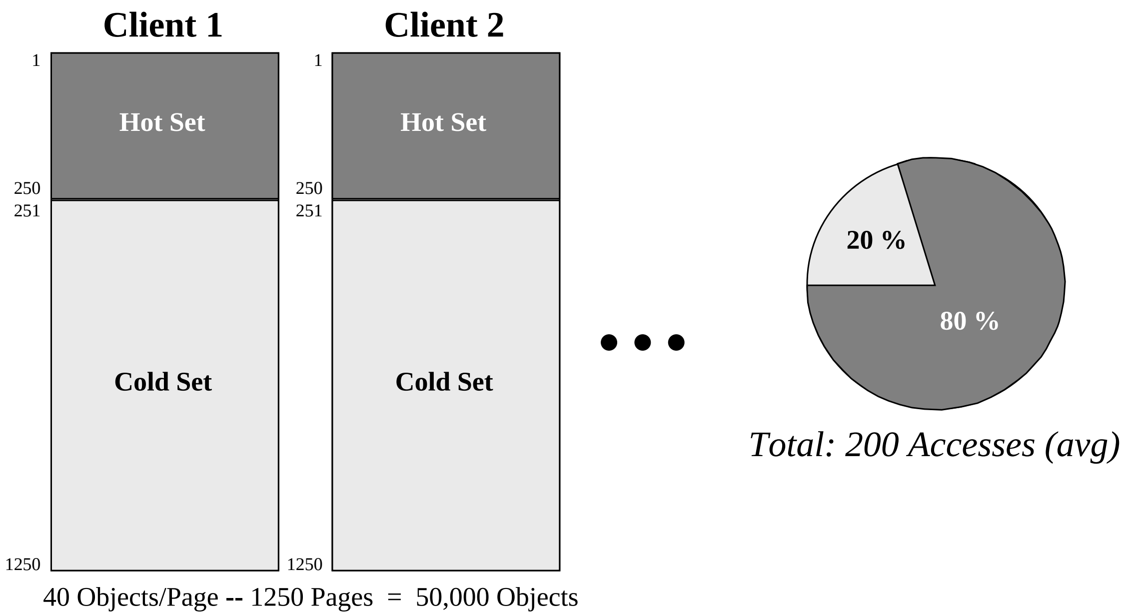


Figure 4-7. HICON Workload

The HICON workload is an object-based version of the page-based HICON workload from [23, 24]. Skewed 80–20 workloads such as HICON have been used in many database studies, such as studies of shared-disk database systems. We use it to study the effects of data contention, for the case where the contention occurs over a large region due to a large number of accesses hitting this region.

HICON's access pattern is depicted in Figure 4-7, where a 1,250 page working set has been split into a 250 page "hot set" that is accessed 80% of the time and a 1,000 page "cold set" that is accessed 20% of the time. The "hot set" will clearly have more contention over it, which is why we label it "hot". Transactions average 200 object accesses (20 pages are touched). Both the hot set and the cold set in HICON are uniformly shared by the clients.

Note that the next workload, TINY+PRIVATE, was designed to have high contention due to a *small* number of accesses to a TINY hot set. In contrast, under HICON *most* of the accesses go to a larger "hot set" and are a potential source of conflict.

TINY+PRIVATE Workload

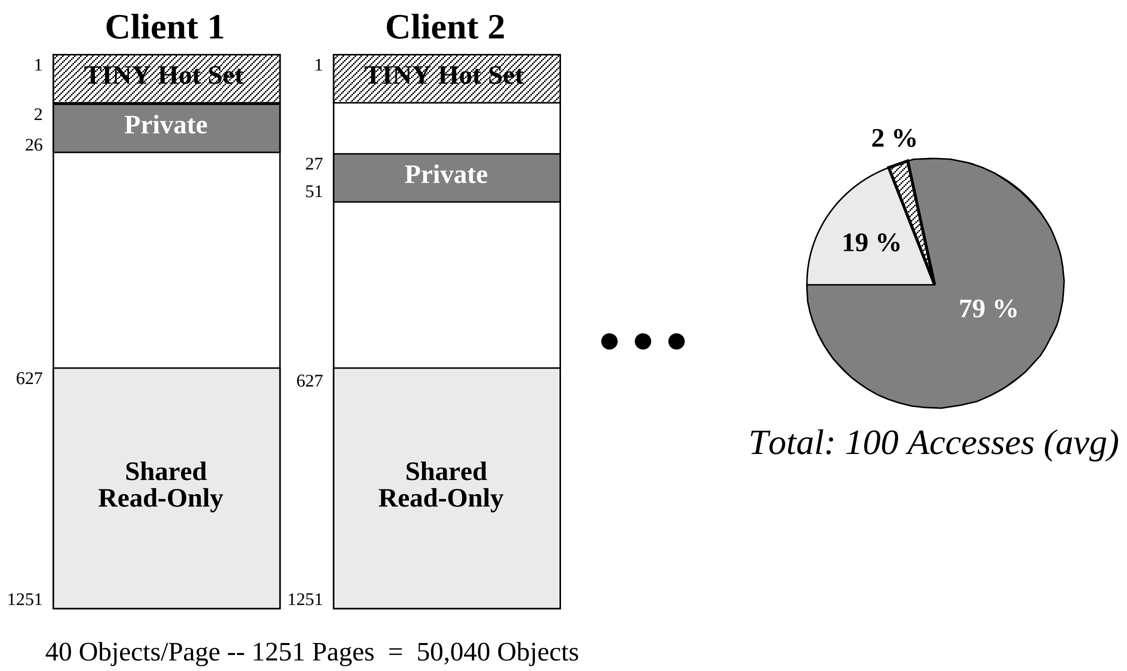


Figure 4-8. TINY+PRIVATE Workload

The HICON workload just described accesses its hot set most of the time; a large set of objects is updated frequently by all the clients. This leaves the inverse case, where clients mostly access their own private regions, and only infrequently access a small "hot set" of objects under high contention. For example, a count of the number of parts in an automobile design might be updated by all the clients, even though each client is updating a different part of the overall design. This insight leads to the TINY+PRIVATE workload.

TINY+PRIVATE is the same as PRIVATE, except a 1 page TINY region has been added. (TINY must be very small to be "hot" because there are few accesses to it.) The resulting access pattern is shown in Figure 4-8. Where PRIVATE has 80% private accesses and 20% read-only accesses, TINY+PRIVATE has 79% private accesses, 19% read-only accesses, and 2% TINY accesses.

TINY+PRIVATE transactions average 100 accesses; because they average only 2 TINY accesses, a cluster size of 2 is used for these accesses. An average cluster size of 10 is still used for other accesses. For the TINY region, a 100% cluster write probability is used; the default object write probability is 50%. For the private pages, a net object write probability of 10% is always used.

TINY+PRIVATE was designed to examine how concurrency control schemes perform when there are hot objects that are accessed rarely. Since the only contention is over the TINY region, this workload isolates the effects of contention over these hot objects.

4.2.4 Summary of Workload Generator Settings

Parameter	**UNIFORM**	**HICON**	**PRIVATE**	**TINY+PRIVATE**
Transaction accesses	180–220	180–220	140–180	90–110
Pages per private region	—	—	25	25
Pages in shared-1 region	1,250	250	—	1
Pages in shared-2 region	—	1,000	625	625
Total working set size	1,250	1,250	1,250	1,251
Private access prob.	—	—	80%	79%
Shared-1 access prob.	100%	80%	—	2%
Shared-2 access prob.	—	20%	20%	19%
Private cluster write prob.	—	—	50%	20%
Shared-1 cluster write prob.	50%	50%	—	100%
Shared-2 cluster write prob.	—	20%	0%	0%
Private object write prob.	—	—	20%	50%
Shared-1 object write prob.	20%	20%	—	50%
Shared-2 object write prob.	—	50%	0%	0%
One cluster per page	true	true	true	false
Private cluster size	—	—	5–15	5–15
Shared-1 cluster size	5–15	5–15	—	2–2
Shared-2 cluster size	—	5–15	5–15	5–15

Figure 4-9. Parameter Settings: UNIFORM, HICON, PRIVATE, TINY+PRIVATE

Parameter	**HOTCOLD**	**SMALL+HOTCOLD**
Transaction accesses	180–220	180–220
Pages per private region	50	50
Pages in shared-1 region	—	50
Total working set size	1,250	1,300
Private access prob.	80%	80%
Shared-1 access prob.	—	10%
Other-region access prob.	50%	10%
Private cluster write prob.	50%	50%
Shared-1 cluster write prob.	—	50%
Other-region cluster write prob.	20%	50%
Private object write prob.	20%	20%
Shared-1 object write prob.	—	20%
Other-region object write prob.	20%	20%
One cluster per page	true	true
Private cluster size	5–15	5–15
Shared-1 cluster size	—	5–15
Other-region cluster size	5–15	5–15

Figure 4-10. Parameter Settings: HOTCOLD, SMALL+HOTCOLD

Figure 4-9 shows the workload generator parameters for the UNIFORM, HICON, PRIVATE, and TINY+PRIVATE workloads. These workloads do not use other-region accesses, so the other-region rows are not shown. Figure 4-10 shows the workload generator parameters for the HOTCOLD and SMALL+HOTCOLD workloads. These workloads do not use the shared-2 region, so shared-2 rows are not shown. In both cases the cluster and object write probabilities are the default probabilities. As discussed above, a number of different combinations of cluster and object write probabilities were used with each workload; the results of varying these parameters are reported in Section 6.2.1.

In addition to the parameters shown in these two figures, there are two other workload parameters, *restart change probability* and *percent forced read-only*. Chapter 6 includes a sensitivity analysis experiment for these parameters (each is varied from 0% to 100%).

The default forced read-only setting is 0%; the ability to "force" transactions to be read-only is not normally used. The default restart change probability is 50%. This setting means that during re-execution of a failed transaction, if one of the objects accessed by the transaction has a new value, there is a 50–50 chance that this will cause the remaining accesses in the transaction to be replaced with new accesses.

Most concurrency control studies use the same access pattern on restart (the "perfect restart" assumption). This approach favors optimistic schemes. Our use of a 50% restart change probability results in a fairer comparison between optimism and locking.

4.3 Related Experimental Framework

This section discusses how our simulation framework differs from the framework used by Carey, Franklin, and Zaharioudakis (CFZ) in their study of adaptive-granularity callback locking schemes [10].

Four of our workloads (PRIVATE, HOTCOLD, UNIFORM, and HICON) are based on workloads with the same names used both in the CFZ study and also in several earlier client-server studies [23, 24].

Our workload generator has a number of features that are extensions of the CFZ generator. We control both the net write probability and the clustering of writes onto pages; CFZ only control net write probability, thus writes are uniformly distributed across all accessed pages. We control the likelihood that restarts result in a changed access pattern; CFZ uses the "perfect restart" assumption. Finally, we can control the mix of read-only and read-write transactions in a way that is orthogonal to the write probabilities used for the read-write cases.

With respect to common resource settings, we use more up-to-date settings for our CURRENT and FUTURE systems. *E.g.*, our systems have more available disk bandwidth at the server and faster client and server CPU speeds. The rationale for our choice of resource settings is summarized in Appendix A. In addition, note that Chapter 6 includes experiments that demonstrate the impact of varying all of the core system parameter settings.

With respect to the system architecture that the two schemes run on, our simulated system has two new data structures: an object-based undo log at the clients, and an object-based modified object buffer (MOB) at the server. Section 2.1 describes these structures and discusses the advantages of using a MOB-based server design. In addition, Section 2.4.1 discusses experiments that demonstrate the low overheads associated with undo logging. Based on these experiments, we decided not to introduce CPU charges for the undo log.

Chapter 5

Main Experimental Results

This chapter presents our main experimental results. It examines the relative performance of AOCC and ACBL for all six workloads, using the CURRENT system settings and default workload settings (as described in Chapter 4).

For each workload, we examine the results across a range of 1–24 clients. The overall result is that AOCC outperforms ACBL across all six workloads, where the difference in performance increases with number of clients across most or all of the 1–24 client range. At a high enough number of clients, the throughput of each scheme "levels off" due to saturation or blocking effects. In each case, AOCC peaks at a higher throughput, where its peak occurs at the same client point as ACBL's peak or at a higher number of clients. These results show that AOCC scales better with number of clients.

The next chapter, Chapter 6, examines the impact of varying different parameter settings. The experiments in Chapter 6 demonstrate that the relative results and insights presented in this chapter hold across a wide range of settings.

Chapter Organization

Section 5.1 gives on overview of how we interpret our simulation results: it discusses the graphs used in this chapter and the approach we take to analyzing the simulation results for each workload. The next six sections (Section 5.2 through Section 5.7) present simulation results for each of the six workloads, in the following order: PRIVATE, HOTCOLD, SMALL+HOTCOLD, UNIFORM, HICON, and TINY+PRIVATE. Section 5.8 briefly summarizes the key insights presented in this chapter.

5.1 Interpreting the Simulation Results

Result Graphs

The following are general points regarding results graphs:

- All graphs use the following convention: AOCC results are plotted using a *solid* line, with *circles* marking the data points; ACBL results are plotted using a *dashed* line, with *boxes* marking the data points.
- To compute a single point for a graph, we use a set of values taken from consecutive *batches* (time intervals within a simulation run). All batches covered at least 5000 commits. A mark (circle or box) is placed at the average value, while a vertical error bar centered on this mark shows the size of 95% confidence interval. (Error bars are not visible when the confidence interval is smaller than the circle or box used to mark the average value.) Confidence intervals for TINY+PRIVATE throughput are all within 4% of average, and throughput confidence intervals for the other workloads are all within 2% of average. This level of confidence is more than sufficient for our comparison goals.
- In this chapter, the X axis always gives the number of clients in the system, while the Y axis gives the range of values for the specific metric shown in the graph. Thus, moving from left to right, each graph shows the impact of adding additional clients to the system. (We show results for 1–24 clients.)
- Graphs have Y-axis labels that give the units for the metric reported. Graph *titles* (given below each graph) give short descriptions of the metric shown. For example, the *Message Count* graph shows the averages number of messages sent per successful commit; the Y axis is labeled *messages per commit*.

Like the Message Count graph, most graphs report average values *per successful commit*. The following graphs have this form: *Message Count* (messages per commit); *Commit Requests* (commit requests per commit); *Client Requests* (client requests — other than commit requests — per commit); *Page Replies* (replies to client requests that include a version of a page, per commit); *Server Requests* (callback requests sent by the server per commit); *Blocking Rate* (blocks per commit); *Abort Rate* (aborts per commit); *Access Count* (object accesses per commit); *Lock Waiting* (the component of overall transaction latency due to locking actions, in ms per commit); *Wasted Work* (the component of overall transaction latency due to executing aborted transactions, in ms per commit); *Wait + Waste* (the combination of the previous two metrics, in ms per commit).

For the *Lock Waiting* graph, any lock acquisition latency that occurs at the server, including waiting due to the use of callbacks and waiting due to actual blocking, is counted as lock waiting time. In addition, message latency for request/response pairs is counted as lock waiting time, but only if this exchange does not involve a page reply. For the case where a page is fetched from the server, the latency due to the page fetch is not counted: any concurrency control scheme, including a scheme that does not use locking, would incur a round-trip page-fetching cost for a missing page. However, if an ACBL client acquires a write lock for state that is *already* cached at the client, the lock request and reply are strictly used for locking purposes, and the latency due to these messages is counted as locking waiting. (Essentially, the locking waiting metric is a measure of the latency due to actions that are *not* performed by an optimistic scheme such as AOCC.)

It is interesting to see how much of the total lock waiting time is due to *wasted lock waiting*, *i.e.*, to lock waiting performed by aborted transactions. Thus, the Lock Waiting graph plots both total lock waiting time (in *black*) and wasted lock waiting time (in *gray*).

For the *Wasted Work* graph, the entire execution time of an aborted transaction counts as *wasted work*.

The wasted work metric is an estimate of the cost of using optimism, and lock waiting is an estimate of the cost of using locking. Neither metric is a perfect measure of concurrency-control costs[1], but these measures are sufficient for producing a comparison of the nature of these costs; the *Wait + Waste* graph gives such a comparison. This graph reports the combined effect of locking and aborts. Wait + Waste is always equal to the wasted work cost for AOCC, since it has no lock waiting cost. For ACBL, Wait + Waste is computed by adding the wasted work and lock waiting metrics and then subtracting wasted lock waiting time, so that the overlap between the two metrics is not "double counted."

In addition to the per-commit graphs above, the following graphs are used: the *Throughput* graph gives overall system throughput in *commits per second*; the *Server Utilization* graph gives average server CPU utilization (the percent of total available CPU instructions that are utilized); and the *Disk Utilization* graph gives the average disk utilization across all the server disks, where each disk's utilization is given by the percentage of the total batch time that the disk used to carry out read and write requests.

Finally, for each Throughput graph there is a corresponding *Percent Improvement* graph. This graph reports on the *difference* between AOCC and ACBL throughput values as a percentage of the smaller value. We use a bar graph format for this graph, where a bar is drawn for each of the following client values: 1, 2, 4, 8, 12, 16, 20, and 24. Suppose 'AOCC' and 'ACBL' stand for the AOCC and ACBL throughput values for a system with C clients. The bar for the C client case is drawn as follows:

- If $AOCC > ACBL$, percent improvement is given by $(AOCC - ACBL)/ACBL$ and is shown as a bar rising above the X axis.

- For $ACBL > AOCC$, percent improvement is given by $(ACBL - AOCC)/AOCC$ and is shown as a bar dropping below the X axis.

Note that the smaller throughput value is always used as the denominator; *percent improvement* indicates the gain in performance achieved if one switches from using the slower scheme to using the faster scheme. We use the direction of the bar (above or below the X axis) to show whether it is AOCC or ACBL that is the better scheme.

[1] For example, pages fetched during an aborted transaction might be accessed during a restart, or even by some future transaction, thus not all work performed during an abort is "wasted." At the same time, fetches that are *not* useful can add to the server load and lengthen average client request times for all client requests, including those performed during successful transaction executions. Unless the access pattern of a transaction is very likely to change on restart, useful fetches are more likely than non-useful fetches; we expect the wasted work metric to be an under-estimate of abort costs.

In this chapter, AOCC is always the better scheme, thus all bars are above the X axis. (Chapter 6 reports on some experiments that have Percent Improvement graphs with bars that drop below the X axis.)

Examing Simulation Results

We use total system throughput in commits per second as our main metric for comparing the performance of the two schemes. Since we use a closed simulation model (each client is always executing a transaction), throughput and latency are inversely related: the scheme that has better throughput also has lower average latency.

The reader should keep in mind that our simulator was not designed to predict the performance for a specific database system, but rather to explore the relative performance of the two schemes, using a range of different system configurations. Thus, we are interested in the relative positioning (as well as the "shapes") of the AOCC and ACBL throughput curves, while specific throughput values are less interesting. (If one scheme achieves 100 commits per second while the other achieves 200 commits per second, the doubling of performance is more interesting than the values "100" and "200.")

For each workload we present two discussions, *Main Results* and *Peak Throughput Results*. The first discussion presents the general reasons for AOCC's better performance and for the increasing performance gap as clients are added to the system. The second discussion presents the reason that each scheme's total system throughput approaches or attains some peak value.

Note that while our throughput graphs show a degrading throughput beyond the client point where the peak value is achieved, it is possible to design an *admission control* policy for a database system that attempts to keep the system performing at near-peak performance at client points that are higher than the "ideal" number of clients. [11] A discussion of how one would implement admission control for either AOCC or ACBL is beyond the scope of this thesis. However, we can still consider the impact of adding such policies to the two schemes: each scheme would have a curve that rises to its peak throughput value, falls to some near-peak value, and then remains at roughly the same value through 24 clients (and beyond). Thus, if admission control were used, the performance gap between the two schemes would be (roughly) constant above the client point where both schemes have peaked, and the size of the performance gap would be roughly the size of the gap between the two peak values.

Since a peak vs. peak throughput comparison is interesting, the *peak vs. peak percent improvement* is given at the top of each Percent Improvement graph (centered above the bar graph). The maximum throughput values in the 1–24 client region are used to compute this value. For certain workloads, one or both of the throughput peaks occurs beyond the 24 client point. For these workloads, since AOCC's throughput is always rising faster than ACBL's throughput at 24 clients, the actual peak vs. peak percent improvement is larger than the peak vs. peak percent improvement given in the Percent Improvement graph.

5.2 PRIVATE Workload

In the PRIVATE workload, each client has its own 25 page private region that it both reads and modifies; in addition, the PRIVATE working set includes a 625 page read-only region. On average, 80% of a client's accesses go to its own private region, and 20% go to the read-only region. (Note that in HOTCOLD and SMALL+HOTCOLD, one client can access another client's private region; this is not the case for PRIVATE.)

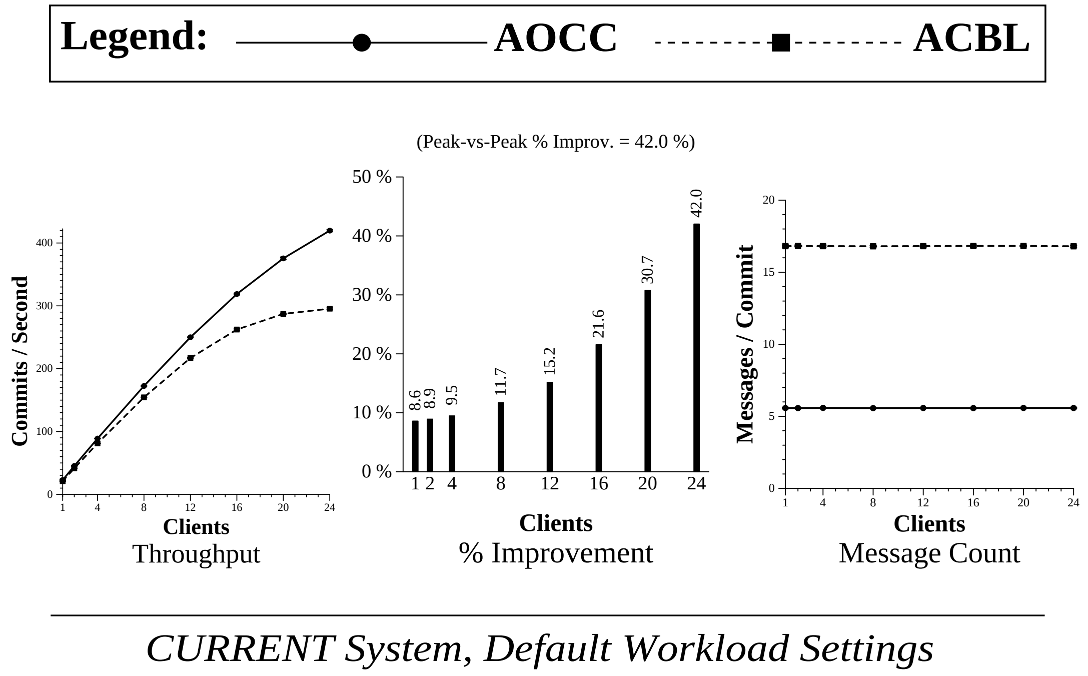


Figure 5-1. PRIVATE: Throughput, % Improvement, Message Count

Since PRIVATE has no data contention, there are no abort costs or blocking costs; messaging costs alone determine the relative performance of the two schemes. Figure 5-1 gives the Throughput, Percent Improvement, and Message Count graphs for PRIVATE. The Message Count graph shows that messages per commit is flat across the 1–24 client range, where ACBL always uses 11.2 more messages per commit than AOCC. AOCC has lower messaging costs and higher throughput. The Percent Improvement graph shows that AOCC outperforms ACBL by 8.6% at one client and by 42% at 24 clients. (Due to the scale of the Y axis, low-client throughput differences are not visible in the Throughput graph. *E.g.*, at one client, the throughput values for AOCC and ACBL are 22.9 and 21.1 commits per second, respectively.)

The performance gap between the two schemes increases as clients are added, where this gap widens at a faster rate in the 12–24 client range than in the 1–12 client range. We first discuss the general reasons for AOCC's better performance and for the widening performance gap as clients are added to the system. We then discuss peak throughput issues and the reason for the more rapid change in percent improvement for the higher client range.

Main Results

The message count difference of 11.2 messages per commit shown in Figure 5-1 is easily explained. ACBL uses write lock requests for the updates that are performed on private pages, and (since there is no contention) these requests always result in page-level write locks. Thus, ACBL uses one write lock request/write lock reply pair for each page that is updated, while AOCC does not use these locking messages. For the default PRIVATE settings, an average of 5.6 private pages are updated per commit, producing an average of 11.2 additional messages.

The PRIVATE workload demonstrates an important point: the cost of using additional messages per commit depends on the number of clients in the system. As clients are added, even though the gap in message *count* per commit is a constant 11.2 messages, there is a widening gap in message *cost* per commit, due to increasing contention for use of the server CPU.

Each client request has server CPU costs, including the costs for receiving the request and sending the reply. As more clients compete for use of the server CPU, the average delay incurred at the server (per client request) increases. When the server CPU is not close to its saturation point, the increase in delay due to resource contention is linear with respect to both message count and number of clients. If there are C clients, the added cost of using K extra requests per commit is proportional to CK. (Once the server CPU is near its saturation point, the added cost of an extra message is worse than linear with respect to CK.)

Peak Throughput Results

Figure 5-2 gives the Server Utilization and Disk Utilization graphs for PRIVATE. Consider first each scheme's relative use of the total available CPU and disk resources: at any client point C, AOCC's server utilization is always higher than its disk utilization, and the same is true for ACBL. Server CPU is thus the *critical resource* for both schemes: as the server CPU is driven to saturation, system throughput levels off and eventually peaks. (The terms *level off* and *peak* are shorthands for "asymptoticly approach a peak throughput value" and "reach a peak throughput value," respectively.)

Since server CPU is the critical resource for both schemes, and ACBL saturates this resource faster than AOCC (as shown in the Server Utilization graph), AOCC will peak at a higher number of clients than ACBL. We can this see from the Throughput graph in Figure 5-1: ACBL's throughput levels off (and nearly peaks) by 24 clients, while AOCC's throughput does not.

When one scheme levels off at a client point where the other scheme's throughput is still rising, the performance gap between the two schemes increases rapidly from the point where the first scheme starts to level off to the point where the second starts to level off. For PRIVATE, we see such a rapid change in percent improvement starting

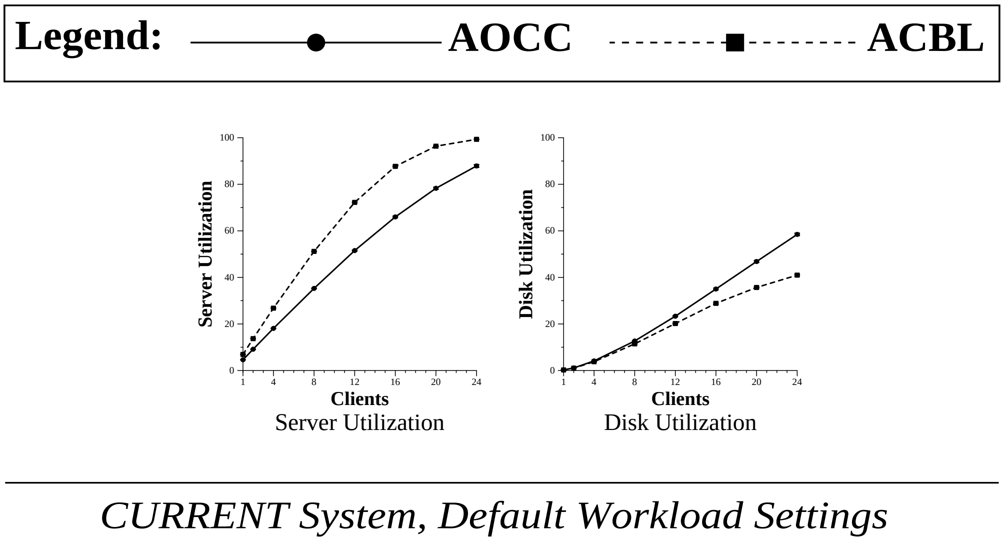


Figure 5-2. PRIVATE: Server and Disk Utilization

at roughly 16 clients; this rapid increase will stop somewhere beyond the 24 client point used in our experiments, at the point where AOCC's throughput starts to to level off.

Neither scheme peaks by 24 clients, thus we cannot compute the peak vs. peak percent improvement. Since AOCC's throughput is rising more rapidly than ACBL at 24 clients, the actual peak vs. peak improvement is higher than the 42% improvement reported in the Percent Improvement graph. (The Percent Improvement graph reports on the difference in the two maximum throughput values for the 1–24 client range).

Both schemes use roughly the same number of disk operations *per commit*, but this is not apparent in the Disk Utilization graph. AOCC's lower messaging costs allow it to commit more transactions *per second*, thus AOCC performs more disk operations *per second*; this results in higher disk utilization for AOCC, as shown in the Disk Utilization graph.

5.3 HOTCOLD Workload

The working set for the HOTCOLD workload is 1250 pages in size. This working set is divided into a set of 25 regions (50 pages per region), where each client in the system is the "owner" of one *private region*; the rest of the regions are *unclaimed regions*. (With 2 clients in the system, there are 2 private regions and 23 unclaimed regions; with 24 clients in the system, there are 24 private regions and just one unclaimed region.)

On average, each client accesses its own private region 80% of the time; its other accesses are *other-region* accesses that go to any of the other 24 regions of the working set. An other-region access by client C_i can either access an unclaimed region or the

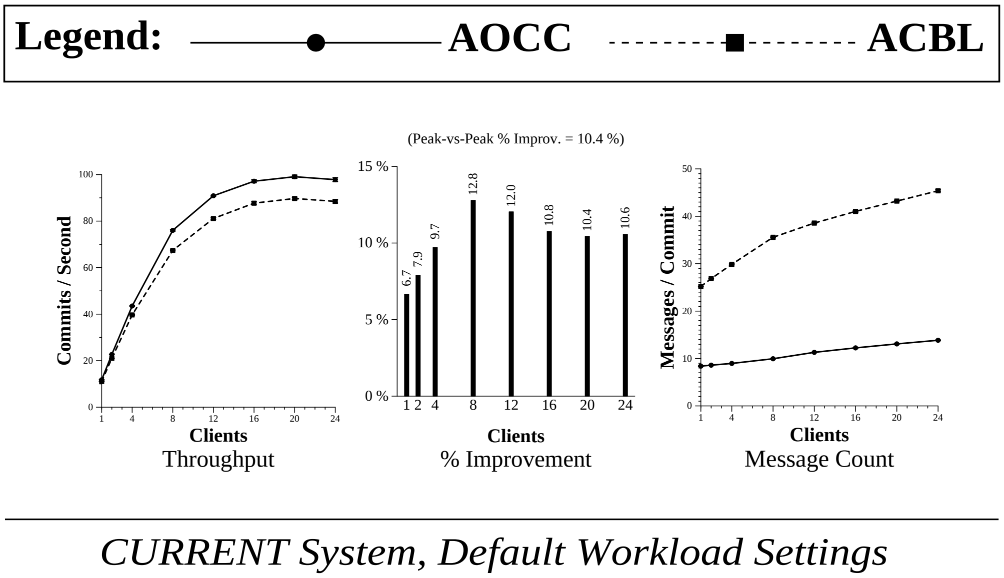


Figure 5-3. HOTCOLD: Throughput, % Improvement, Message Count

private region of some other client C_j. We call the latter case an *invasion* of client C_j's private region; such invasions are the major reason for data conflicts under HOTCOLD. (Data conflicts can also occur if two clients both happen to access the same unclaimed region using other-region accesses, but such conflicts are rare compared to conflicts due to invasions of private regions.)

Figure 5-3 gives the Throughput, Percent Improvement, and Message Count graphs for HOTCOLD. AOCC again has lower messages per commit and higher throughput. Unlike the PRIVATE workload, however, messages per commit for both schemes rises as clients are added to the system. Moreover, ACBL's message count rises faster than AOCC's message count, causing the percent improvement to climb faster under HOTCOLD than it does under PRIVATE. (From 1–8 clients, PRIVATE percent improvement rises by 3.1% while HOTCOLD percent improvement rises by 6.1%.)

We first discuss the general reasons for AOCC's better performance and for the widening message gap; we then discuss peak throughput issues.

Main Results

Figure 5-4 shows the Blocking Rate, Abort Rate, and Access Count graphs. The level of data contention increases linearly with number of clients in the system, resulting in a nearly linear blocks per commit for ACBL and and aborts per commit for AOCC. (Restarts have a lower abort rate than first-run executions, thus AOCC aborts per commit is not perfectly linear.)

ACBL's increasing blocking rate means that the average waiting time due to blocking increases. It also (indirectly) implies more object-level locking is used, and

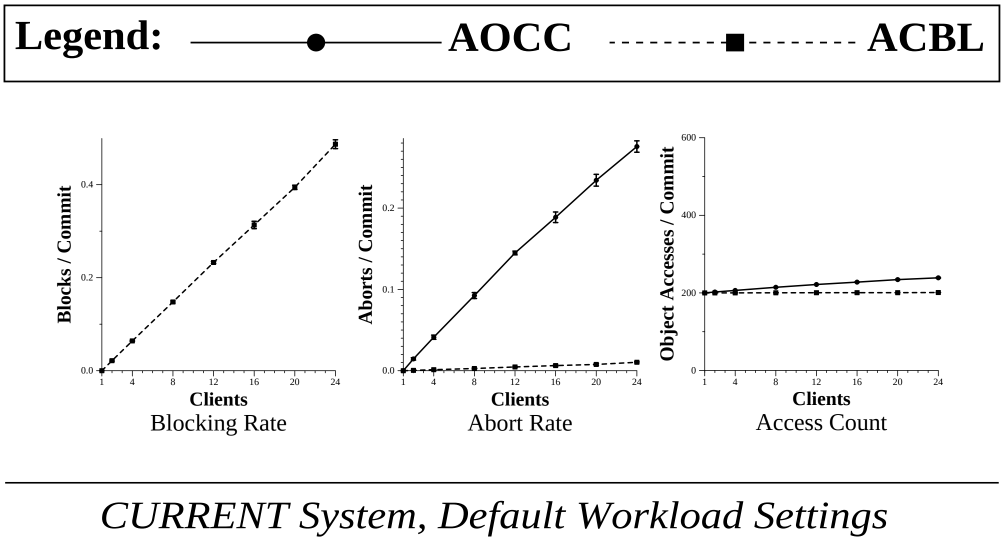


Figure 5-4. HOTCOLD: Blocks, Aborts, Accesses (per Commit)

thus more lock and callback requests are used. (Since ACBL only blocks at the object level, the presence of blocking implies the use of object-level locks.)

AOCC's increasing abort rate means that more object accesses occur per commit, as accesses performed during a failed transaction are "wasted." The Access Count graph in Figure 5-4 shows that AOCC's accesses per commit rises from 200 to 240 as we move from 1 to 24 clients. In contrast, ACBL remains at a flat 200 accesses per commit, due to its low abort rate. (At 24 clients ACBL has reached 201 accesses per commit.)

Since an abort rate of 0.28 adds 40 accesses to the average transaction length, the average length of a *failed* execution is roughly 143 accesses. This average is below 200 accesses per commit because AOCC clients perform *early abort detection* using the piggy-backed invalidations that arrive in fetch replies. Early abort detection saves unnecessary commit requests, and also some unnecessary accesses. (Since failed transactions average 143 accesses, 57 accesses are saved per abort, on average.)

An increasing abort rate can also imply the use of additional commit requests and additional fetch requests (and the corresponding replies). For HOTCOLD, however, such an increase does not occur; we discuss the reasons below.

While the blocking rate and abort rate metrics for HOTCOLD do rise to non-trivial values, blocking and abort costs are not the primary component of transaction latency; for the default settings, HOTCOLD is not a high-contention workload. Therefore, message cost is still the primary determinant of relative performance. We now examine the message behavior of the two schemes in detail.

Figure 5-5 gives the message breakdown graphs: Commit Requests, Client Requests, Server Requests, and Page Replies.

Since large messages have higher send and receive costs compared to small messages, it is possible that one scheme could send more total messages but still have a

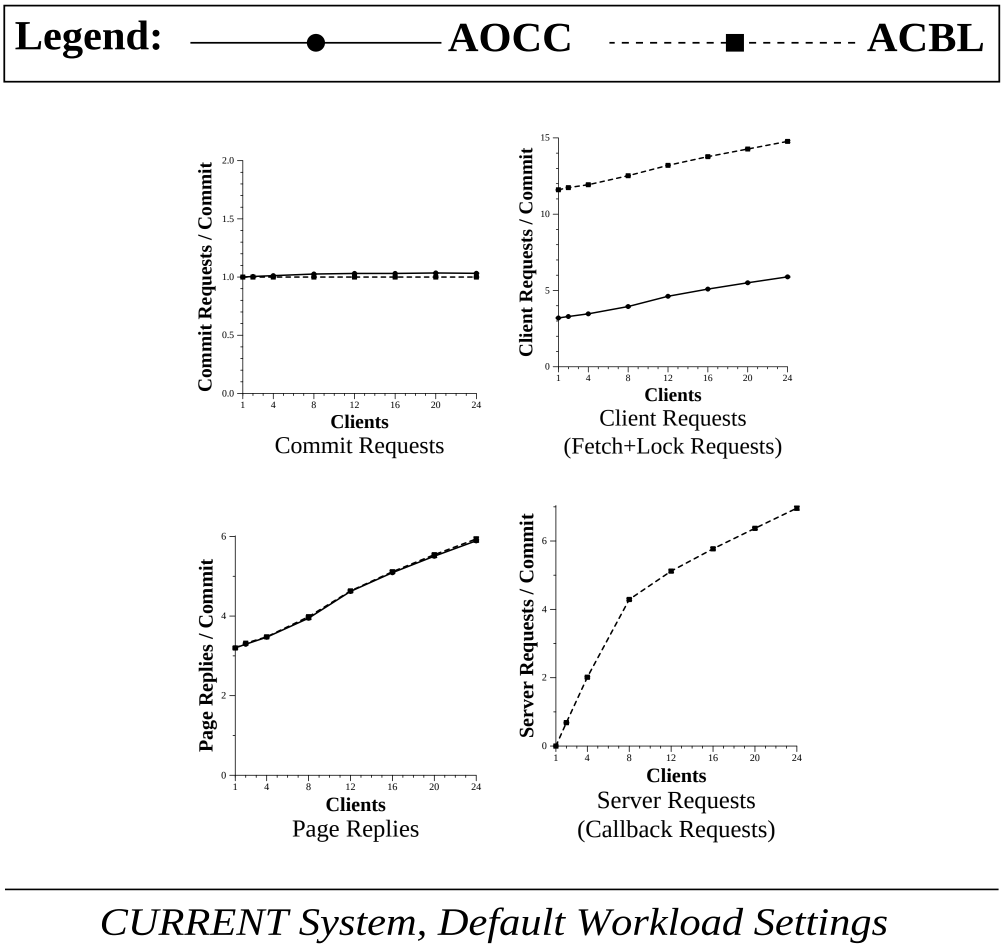


Figure 5-5. HOTCOLD: Message Breakdown

lower messaging cost. Thus, it is important to compare the schemes with respect to their use of the two largest message types, commit requests and page replies. (Commit requests are large because they contain modified object state to be installed at the server. Page replies are large because they contain 4 KB pages.)

We can see from the Commit Requests and Page Replies graphs in Figure 5-5 that the two schemes use roughly the same number of large messages. Even though AOCC's abort rate rises to 0.28 aborts per commit, its commit requests per commit remains near 1.0. Under HOTCOLD, the large majority of AOCC aborts occur at the clients; the need to abort is almost always detected prior to reaching the commit point, and very few commit requests are sent for transactions that end up aborting. (ACBL's commit requests per commit remains near 1.0 because its use of locking keeps its abort rate very low.)

One might think that the extra accesses performed during restart by AOCC would mean additional fetches, and thus additional page replies. However, the Page Replies graph shows that this is not the case: both schemes use roughly the same number of page replies per commit, where this number increases with additional clients. The reason for this phenomenon is that both schemes have very similar caching behavior, even though AOCC transactions sometimes abort and ACBL transactions do not.

Most accesses performed by a client are to its own private pages. A private access results in a fetch only when the accessed object or its containing page has been modified by some other transaction, causing the object to be marked or the containing page to be dropped. Under both schemes, when some invaliding client I modifies objects on a page owned by another client C, this usually results in exactly one fetch:

- Under AOCC's optimistic approach, either a transaction running at client C will end up aborting due to this update, or it will not. The first case occurs if an out-of-date version of an object written by client I is read from C's cache. In this case the updated page is fetched during restart. The second case occurs if objects updated by I are removed from C's cache before a transaction at C happens to read an out-of-date version. In this case some transaction at client C will *eventually* access one of these updated objects, and a fetch will occur.

- Under ACBL's locking approach, it is possible that a transaction running at client C will block on transaction I due to one of I's updates; in this case C is resumed once I commits, and C will fetch the page updated by I. It is also possible that no transaction at C will block on the updater at I, either because the updater at I is the one that ends up blocking, or because no blocking occurs. In either of these non-blocking cases, the updated state is removed from C's cache either when a callback is received or when a committing C transaction processes its "promises" to mark objects as unavailable. As above, some transaction will *eventually* access one of the removed objects and a page fetch will occur.

In summary, an update by an invading client causes a single fetch by the owning client at some point after the update has occurred. An abort can occur under AOCC, but this does not change the number of fetches it uses to obtain the new object state.

The above reasoning relies on the fact that the owners of private pages access these pages more frequently than these pages tend to be updated by "invading" clients. The skew in access frequency makes it unlikely that multiple updates to a given private page P by "invaders" will occur during a time interval that spans two accesses to page P by P's owner, and thus each update by an invader triggers a single fetch by the owner. (The UNIFORM workload does not have such a skew; see Section 5.5.)

Note that aborts *would* result in additional page replies per commit if AOCC did not use undo logs at the clients. The state stored in an undo log is used to undo any uncommitted updates performed on objects updated locally by an aborted transaction (except for those objects removed from the cache due to piggy-backed invalidations). If undo logs were not used, updated state would be removed on an abort, rather than restoring it. This would result in more fetches per commit: aborts would cause frequently accessed state (such as private state) to be removed from client caches.

It is clear that restoring a private page is better than dropping it, given the high frequency of private accesses. Whether it is useful to restore an infrequently-accessed page (*e.g.*, an unclaimed page modified by an other-region access) depends on the application's restart behavior. For the experiments reported in this chapter, a 50% restart change probability is used; some pages restored on abort will not be accessed during restart. If we used a "perfect restart" assumption, *all* restored pages would be used immediately; this would magnify the benefit of using undo logs.

So far we have shown that aborts cause AOCC to use more object accesses than ACBL, but not more commit requests or page replies. As a result, AOCC aborts are very inexpensive for this workload. While average transaction latency at 24 clients is 245.3 ms per commit, the total wasted work time is only 56.6 ms per commit. Of this 56.6 ms, only 3.2 ms is due to the additional "think time" charged to the client CPU (for 32 extra read accesses and 8 extra writes). Almost all of the measured wasted work time is due to fetches that occur during aborted transactions. As we argue above, most such fetches are in fact useful fetches.

While AOCC's abort costs are low, ACBL's locking-induced costs are not. As contention increases, ACBL transactions spend more time blocked at the server waiting on locks. However, since HOTCOLD is not a high-contention workload, this blocking time is relatively small. The most significant locking cost incurred by ACBL is the time spent sending and receiving messages as part of acquiring locks.

As shown in the Client Requests graph in Figure 5-5, ACBL uses more client requests per commit than AOCC. It also uses an increasing number of callback requests per commit, as shown in the Server Requests graph, where AOCC uses no such requests. As with PRIVATE, the gap in client requests is due to the use of lock requests. Unlike PRIVATE, both schemes show an increasing number of client requests per commit as clients are added to the system. More clients means a higher update rate, resulting in more object marking and more page dropping at each client cache during a given client transaction's execution. As more data is removed from caches, it becomes more likely that fetches will be required, thus both schemes show an increase in both client requests and page replies.

In combination, the extra client requests and callback requests result in a message gap that increases as clients are added to the system. AOCC's messaging advantage over ACBL increases with the number of clients in the system.

Finally, it is interesting to examine the "knee" that occurs in the Server Requests graph at roughly 8 clients. (This change in slope also causes the total message count to have a "knee," as can be seen in the Message Count graph in Figure 5-3.) This knee occurs because the rate of growth in the number of callback requests sent by the server is different for the low-client and high-client regions.

For the low-client region, 20% of the accesses are other-region accesses, and these usually go to unclaimed pages, since most of the 25 regions in the working set are unclaimed. Pages in unclaimed regions are rarely updated, thus the expected number of clients caching an unclaimed page grows as clients are added to the system. A growth in the number of cachers translates to a growth in the number of callbacks required to obtain a write lock.

For the high-client region, most of the 25 regions are owned by some client. 80%

of the accesses go to a client's own private region, and most of the remaining accesses go to some other client's private region. The pages in a private region are frequently updated by their owner; the system is constantly driven towards a state where each client is the only cacher of its private pages. If C owns private page P, an access to P by invading client I will typically cause either one or two callbacks, depending on whether P is updated by client I. For an update, client C is first sent a callback due to I's write lock request. For either a read-only or update access by client I, an update by C will eventually cause a callback to be sent to I.

More than two callbacks per "invasion" can occur when there are overlapping accesses to C's private page P by two or more invaders, or when the system switches to using object-level locking for page P, due, *e.g.*, to client C refusing to drop P from its cache. However, the number of callbacks per invasion remains low. There is a growth in callbacks per commit, but this is mainly due to the fact that the number of invasions per commit rises with the number of clients in the system.

Peak Throughput Results

Figure 5-6 gives the Server Utilization and Disk Utilization graphs for the HOTCOLD workload. AOCC uses more available disk bandwidth than available server CPU per commit; the disks are the critical resource for AOCC. ACBL's use of the two server resources is nearly balanced, with CPU usage a little higher than disk usage; both resources are critical resources for ACBL. The two schemes have similar per-commit disk costs, thus the difference in relative use of the two server resources is due to ACBL's higher messaging costs.

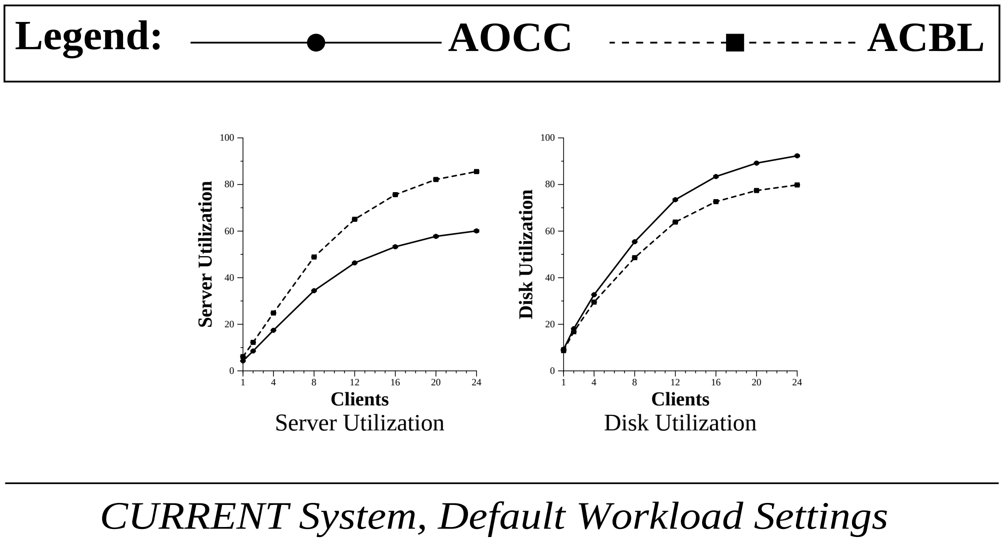


Figure 5-6. HOTCOLD: Server and Disk Utilization

Both schemes drive their critical server resource(s) towards saturation at a faster rate than they do under PRIVATE. Both have a system throughput that levels off and peaks at 20 clients. Due to the similar saturation rates, the percent improvement (shown in Figure 5-3) is relatively stable above the 12 client point, remaining close to the peak vs. peak percent improvement of 10.4%.

5.4 SMALL+HOTCOLD Workload

SMALL+HOTCOLD modifies the HOTCOLD workload by adding the SMALL region, a 50 page uniformly shared region, and by switching half of the other-region accesses to SMALL accesses. (Each transaction averages 80% private accesses, 10% other-region accesses, and 10% SMALL accesses.) SMALL+HOTCOLD averages 200 object accesses and 20 object updates per transaction.

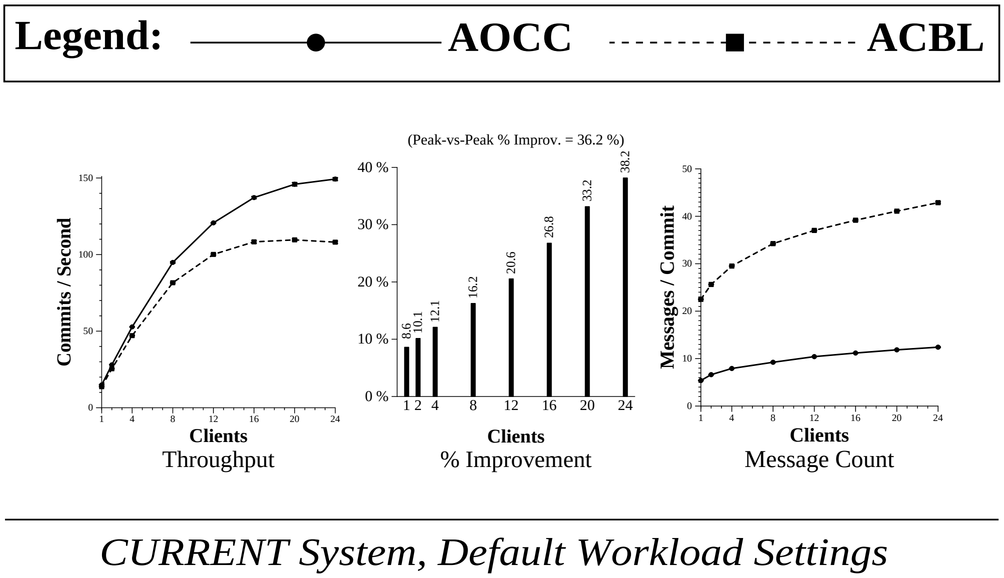


Figure 5-7. SMALL+HOTCOLD: Throughput, % Improvement, Message Count

Of the six workloads, SMALL+HOTCOLD has the richest mix of sharing patterns. It includes both skewed sharing (one client accesses a region more frequently than other clients) and uniform sharing (all clients are equally likely to access a region). For this reason, SMALL+HOTCOLD is used in the majority of the sensitivity analysis experiments presented in Chapter 6.

Figure 5-7 gives the Throughput, Percent Improvement, and Message Count graphs for SMALL+HOTCOLD. As with HOTCOLD, AOCC has a lower message count than ACBL, and the gap in message count increases as clients are added to the system. AOCC's throughput peaks later and higher than ACBL's throughput, where the AOCC peak has not been reached by 24 clients. The result is that percent improvement increases across the entire 1–24 client range.

We first discuss general results, with a focus on the impact of changing half of the other-region accesses found in HOTCOLD to SMALL accesses. We then discuss peak throughput issues.

Main Results

The overall results are similar to the HOTCOLD results, except the performance gap for this workload widens at a slighter faster rate as clients are added to the system. Consider the 1–8 client region, where saturation effects do not yet play a major role in the throughput results. For HOTCOLD, percent improvement rises form 6.7% to 12.8% in this region (see Figure 5-3), while for SMALL+HOTCOLD percent improvement rises from 8.6% to 16.2%.

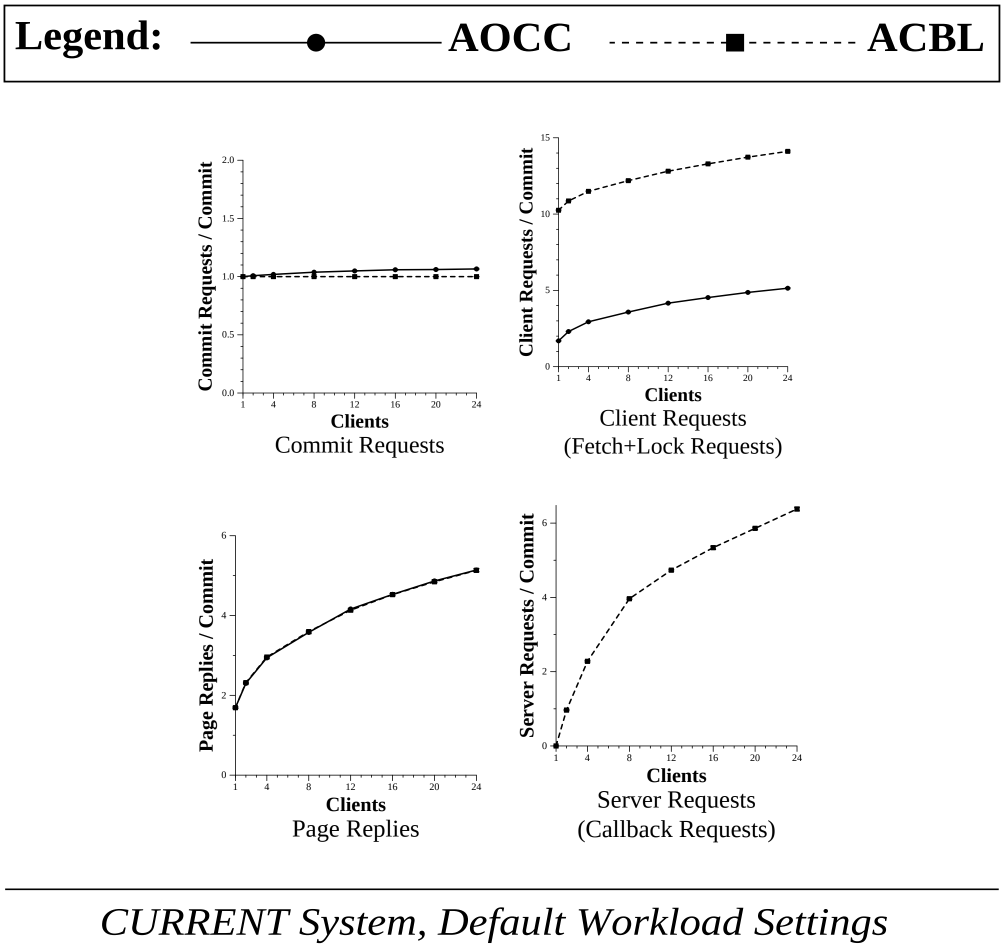


Figure 5-8. SMALL+HOTCOLD: Message Breakdown

Since this workload has replaced some of the other-region accessed performed in HOTCOLD with accesses to the SMALL region, the changes are due based on differences in SMALL accesses vs. other-region accesses:

- The level of contention for SMALL accesses is roughly double the level of contention for other-region accesses. However, we have only changed the contention (probability that a conflict occurs) for 10% of a transaction's accesses: abort and blocking rates are higher, but not significantly higher, than under HOTCOLD.

- At a low number of clients, SMALL accesses tend to hit in client caches, since clients access these shared pages more often than they are updated. In contrast, other-region accesses often require a page fetch, even at a low number of clients. As we add clients to the system, however, more clients are modifying the same set of SMALL pages, and thus the rate at which SMALL pages are dropped from client caches increases. The result is that more fetches occur for SMALL accesses, and there is no longer a difference in client hit ratio when we compare SMALL accesses to other-region accesses. Thus, message costs start slightly lower for SMALL+HOTCOLD than for HOTCOLD, but become roughly equal as clients are added to the system.

- While both other-region accesses and SMALL accesses cause cache misses and thus page fetches, and the fetch rate equals out eventually (as just discussed), there is still a cost difference between these two kinds of cache misses. When a miss occurs on a SMALL page, this page is always present in the server cache; misses occur frequently enough on the set of 50 SMALL pages that they remain cached. On the other hand, for a client miss on an unclaimed page or a private page, the fetch operation at the server may have to read the page from disk, since not all private and unclaimed pages are cached (they don't all fit in the server cache). Thus, while SMALL accesses can cause as many fetches as other-region accesses, they do not cause disk reads. The result is that SMALL+HOTCOLD transactions perform fewer disk operations per commit than HOTCOLD transactions.

The most important difference discussed above is the difference in disk cost. Because SMALL+HOTCOLD requires less disk bandwidth per commit than HOTCOLD, both AOCC and ACBL show higher throughput. In addition, while the message counts are not very different between the two workloads, the fact that AOCC uses fewer messages than ACBL has more of an impact on the gap in performance under SMALL+HOTCOLD, since messaging costs are a larger fraction of overall transaction latency. This explains why we see a wider performance gap (higher percent improvement numbers) for this workload.

Peak Throughput Results

Figure 5-10 gives Server and Disk Utilization graphs for SMALL+HOTCOLD. While AOCC saturated the disk under HOTCOLD, the relatively lower disk costs for SMALL+HOTCOLD cause AOCC to drive the server CPU and server disks towards saturation at roughly the same rate; both are critical resources for AOCC under this workload. ACBL's high messaging costs cause the server CPU to be the critical resource, as was the case with HOTCOLD.

Thus, both schemes have the server CPU as a critical resource. AOCC drives this resource towards saturation at a slower rate, due to its lower messaging costs,

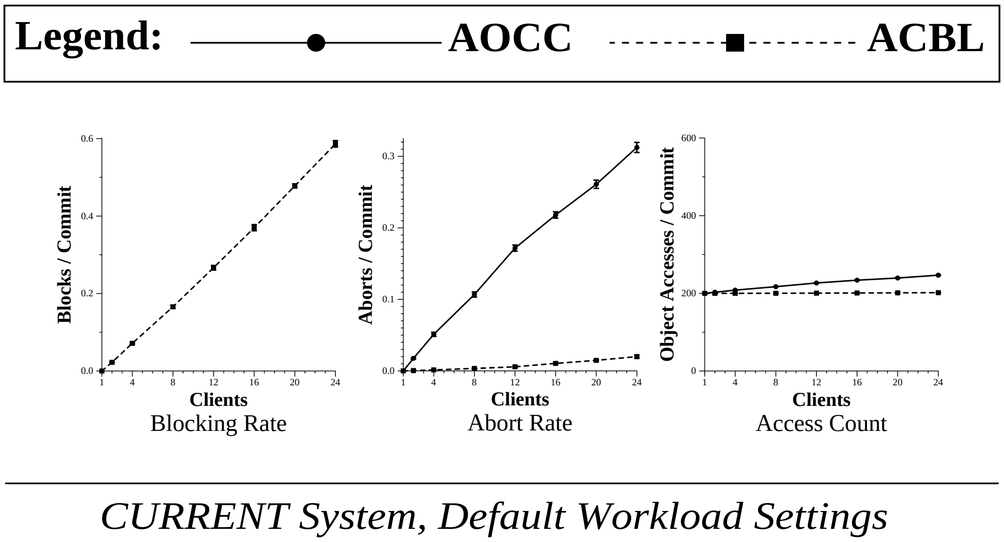


Figure 5-9. SMALL+HOTCOLD: Blocks, Aborts, Accesses (per Commit)

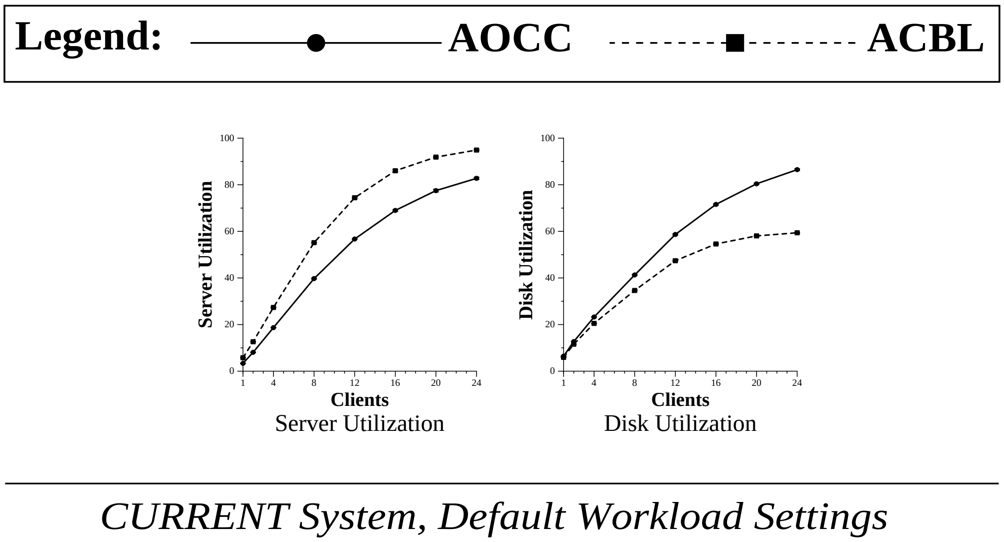


Figure 5-10. SMALL+HOTCOLD: Server and Disk Utilization

causing it to peak at a higher number of clients. ACBL reaches a peak throughput at 20 clients, while AOCC's throughput has not yet peaked by 24 clients. Given AOCC's high server and disk utilization at 24 clients, it is likely that AOCC will peak shortly after the 24 client point. The peak vs. peak improvement will thus be somewhere near the peak vs. peak value of 36.2% that we computed using the 24-client AOCC value and the 20-client ACBL value.

5.5 UNIFORM Workload

As with HOTCOLD and SMALL+HOTCOLD, UNIFORM averages 200 accesses and 20 object updates per transaction. Every page in UNIFORM's 1250 page working set is equally likely to be accessed. (Across many transactions, every object on every page is equally likely to be accessed.)

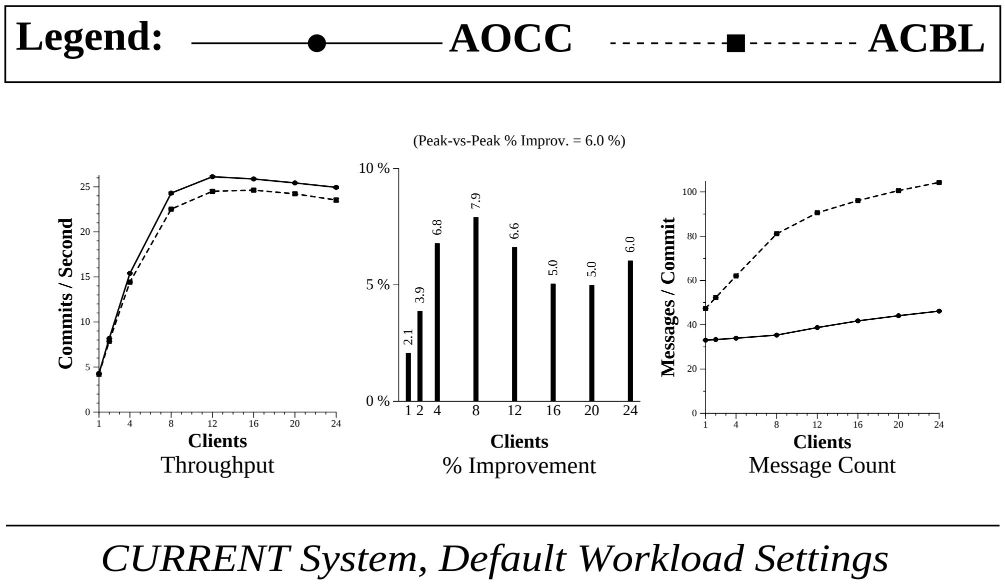


Figure 5-11. UNIFORM: Throughput, % Improvement, Message Count

Unlike UNIFORM, the other five workloads have an *access skew*: 80% of a client's accesses go to a region of the working set that is small enough to fit in the client cache. Except for PRIVATE, clients do not retain all frequently accessed objects in their caches: updates by other clients cause object marks and page drops that remove some of these objects. Nevertheless, many of these objects are cached at any given time, resulting in a better client hit ratio for these workloads than for UNIFORM. The other workloads thus use fewer fetch requests and fewer disk reads per commit. (At a high number of clients, HICON begins to perform *more* fetch requests per commit than UNIFORM, due to the frequency of object marks and page drops for frequently accessed pages. However, these pages are always cached at the server; HICON does not perform nearly as many disk reads as UNIFORM, even when it uses more fetches than UNIFORM.)

Aborts cause AOCC to use use slightly more fetches and disk operations than ACBL, as we show below. This difference is small compared to the total fetches and disk reads used, thus both schemes have a high shared overhead due to UNIFORM's high client cache miss ratio. Given this high shared cost, any differences in other costs (such as differences in the use of messages) have less relative impact on throughput than they do for the other workloads; AOCC's percent improvement over ACBL is lower under UNIFORM than under the other workloads.

Figure 5-11 gives the Throughput, Percent Improvement, and Message Count graphs for UNIFORM. AOCC has a lower message count, and the gap in message count grows as clients are added to the system. AOCC has better throughput than ACBL, where the performance gap is relatively small, as just discussed.

Percent Improvement rises from 2.1% to 7.9% from 1–8 clients; in this region saturation- and thrashing-related costs do not dominate the results. UNIFORM has higher data contention than the previous three workloads, resulting in a higher AOCC abort rate. We first examine why AOCC still outperforms ACBL even with this higher abort rate; we then discuss the saturation/thrashing results for the client region beyond 12 clients, where the percent improvement essentially stabilizes near the peak vs. peak improvement of 6%.

Main Results

Figure 5-12 gives our usual message breakdown for UNIFORM: the Commit Requests, Client Requests, Server Requests, and Page Replies graphs are shown. Figure 5-13 shows the Blocking Rate, Abort Rate, and Object Access Count graphs. We first use the graphs in these two figures to discuss the impact of aborts on AOCC.

The Abort Rate shown in Figure 5-13 shows that AOCC's abort rate rises to over 0.7 aborts per commit at 24 clients. The message breakdown graph shows the impact of these aborts on the two largest message types, commit requests and page replies. Commit requests per commit remains at one across the entire client range; AOCC and ACBL use roughly the same number of commit requests. Since invalidations are piggy-backed on fetch replies, each reply is an "opportunity" for a client to detect that its current transaction should be aborted and restarted. The relatively high number of fetches per commit under UNIFORM means that most aborts occur at the clients, thus the commit requests per commit for AOCC remains close to 1.0.

While AOCC does not use more commit requests per commit, it does use more page replies. The page replies for both AOCC and ACBL increase gradually as clients are added to the system, and AOCC begins to use more page replies per commit than ACBL as its abort rate rises. At 8 clients, the AOCC abort rate is 0.2 and this causes AOCC to use an additional 0.6 page replies per commit. At 24 clients, the 0.7 aborts per commit cause AOCC to use an additional 2.4 page replies per commit. Thus, on average an AOCC abort results in a little over 3 extra page fetches. These fetches cause additional disk operations per commit, making AOCC's disk costs per commit higher than ACBL's disk costs per commit.

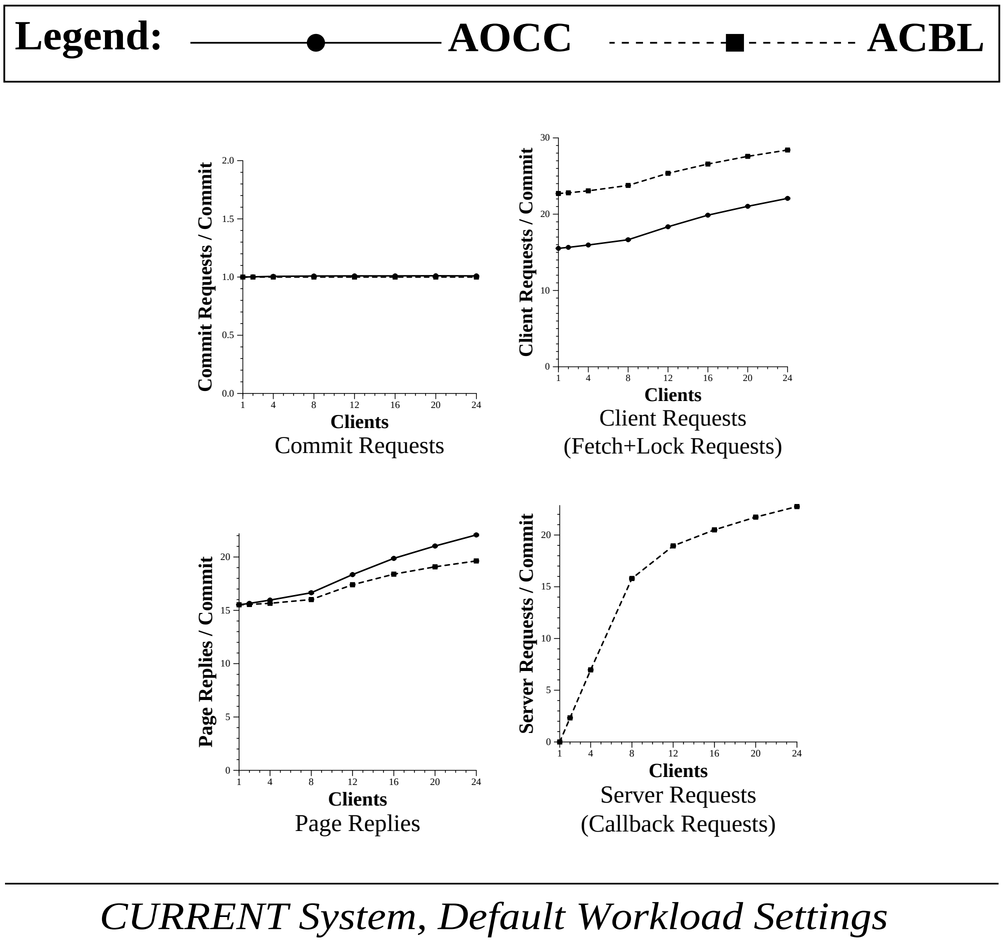
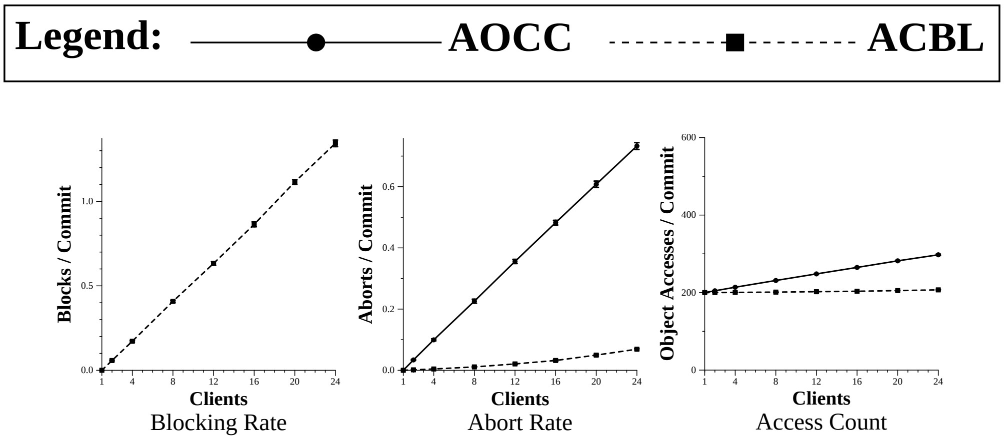


Figure 5-12. UNIFORM: Message Breakdown

Abort-induced fetches are a direct result of our use of a 50% restart change probability. Roughly half of all aborts result in a changed access pattern during restart, and the pages accessed during restart thus include some pages that were not accessed by the previous (failed) execution. These new page accesses have a good chance of not being cached at the client, given the uniform access pattern. In contrast, consider the case of a HOTCOLD transaction that aborts and restarts. Even if the access pattern changes, it is still the case that 80% of the new accesses will go to the client's private pages, and these pages are likely to be cached at the client. In general, if a workload includes private regions, such that each client mostly accesses its own region of the working set, restarts cause relatively few additional page fetches. The UNIFORM and HICON workloads do not have this property, and thus aborts cause relatively more page fetches under these workloads.

In addition to causing new page fetches, aborts also result in a higher average number of object accesses at the client per successful commit. The Object Access Count graph in Figure 5-13 shows this result: while ACBL's access count remains near



CURRENT System, Default Workload Settings

Figure 5-13. UNIFORM: Blocks, Aborts, Accesses (per Commit)

the average transaction length of 200 accesses across the entire client range, AOCC's access count climbs to 298 accesses per commit at 24 clients. Additional accesses cause additional work for the client CPU's. This extra work is relatively inexpensive compared to, *e.g.*, the cost of an additional fetch that incurs an extra disk read. AOCC does not saturate the client CPU's due to the execution of additional object accesses. Moreover, note that extra work performed at a client has less impact than extra work performed at the server. Extra work at client C slows down one transaction: the transaction executing at C. Extra server work slows down *all* transactions in the system, since all transactions are contending for use of this shared resource.

ACBL incurs three costs that grow as clients are added to the system: messaging costs, blocking costs, and abort costs (wasted work). We discuss each in turn.

ACBL's use of lock requests and callbacks causes its overall message count to rise considerably faster than AOCC's overall message count, as we see in the Message Count graph in Figure 5-11. Figure 5-12 gives the message breakdown. Note that ACBL's client requests per commit rises faster than AOCC's client requests per commit, and moreover the ACBL server's use of callbacks increases with added clients (while AOCC does not use such messages).

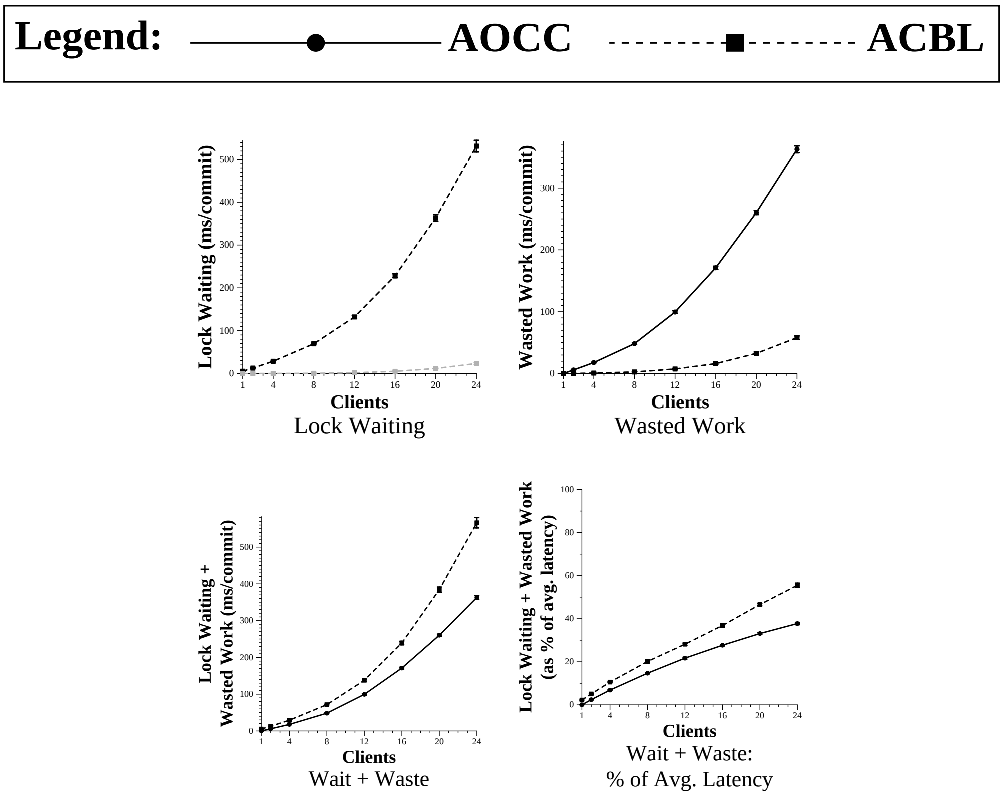
AOCC and ACBL both use an increasing number of fetches per commit as clients are added, due to the growth in the number of object marks and page drops that occur during the lifetime of a single transaction. On top of this shared cost, AOCC also sends some additional fetches and page replies due to restarts. The additional fetches used by AOCC can be determined by examing the Page Replies graph. Note that the gap in page replies per commit is relatively small, compared to the gap in total client requests per commit as shown in the Client Request graph, where ACBL uses considerably more client requests overall.

ACBL's client request count grows faster than AOCC's client request count due to its use of write-lock requests. At a small number of clients, page-level locks are almost always granted, and the number of write lock requests is based on the number of pages that are modified. As clients are added to the system, more lock requests result in object-level write locks, since it becomes more likely that some other client is using the page containing the object to be write-locked. The use of object-level locking prevents unnecessary blocking, but does result in additional messages as compared to page-level locking: when ACBL switches to using object-level locking for a given page, multiple updates to this page require multiple write-lock requests.

The Server Request graph shows that the ACBL server's use of callback requests also rises as clients are added to the system. This graph is rather interesting, as there is a "bend" or "knee" to this graph at around eight clients. At a low number of clients, the number of callbacks per commit grows with the number of clients in the system: with more clients in the system, a given object that must be "called back" is more likely to be cached by some client other than the lock requester. However, as we approach the higher client range, more and more updates occur, causing more object marking and page dropping, and the average number of clients caching any given object or page eventually starts to *decrease* with added clients. Thus, fewer callbacks are used per lock request. However, as pointed out above, more lock requests are used per commit as clients are added. The total callbacks per commit thus continues to rise even when the number of callbacks per lock request is decreasing, but the rate of growth slows compared to the small-client rate of growth, thus explaining the "knee" in the callbacks per commit curve.

Finally, note in Figure 5-13 that ACBL's abort rate approaches 0.07 aborts per commit at 24 clients. (As data contention increases, deadlocks are more likely, and ACBL's abort rate thus rises with number of clients.) ACBL's abort rate is one tenth of AOCC's abort rate; one might expect that the impact is very small. However, ACBL aborts are potentially very expensive (compared to AOCC aborts). Like an AOCC abort, an ACBL abort causes some additional fetches and additional object accesses. *Unlike* an AOCC abort, execution of additional accesses results in additional lock requests per commit, and thus additional lock waiting time per commit. The cost of an ACBL abort thus depends heavily on the average lock waiting time. Since more aborts occur as contention increases, and lock waiting times also increase with increased contention, ACBL's abort-related costs can grow rapidly with rising data contention.

So far we have shown that AOCC uses more disk operations per commit due to an abort-induced "fetch gap" that slowly increases with the number of clients. In contrast, ACBL uses more server CPU per commit due to a much higher "message gap" that widens significantly with the number of clients. Moreover, Figure 5-13 shows that ACBL's blocking rate increases linearly with the number of clients, thus the average lock waiting time will increase as clients are added to the system. There is also a small abort rate for ACBL that may have some impact on its performance. This summary is derived from our *count-based* metrics, such as number of messages per commit, number of page replies per commit, *etc*. A key point of this thesis is that such count-based metrics are useful for understanding performance, but *time-based* metrics are often required to gain a fuller understanding of the behavior of different



CURRENT System, Default Workload Settings

Figure 5-14. UNIFORM: Lock Waiting and Wasted Work

concurrency control schemes.

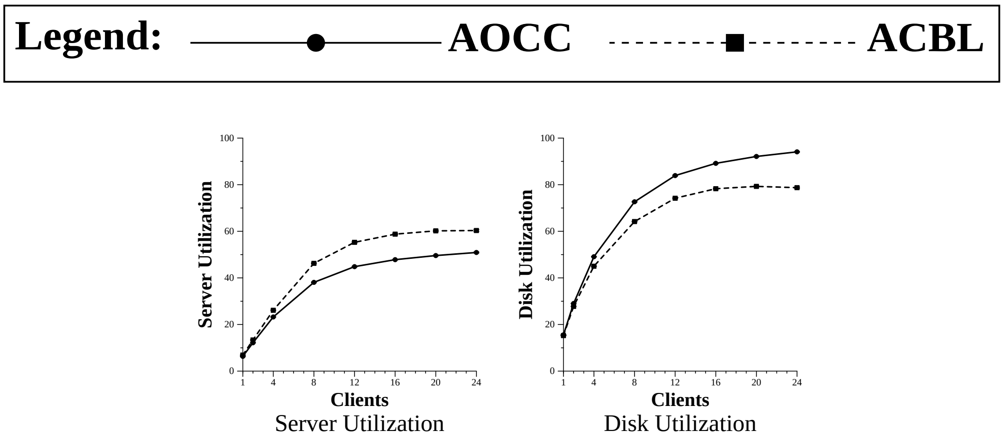
Figure 5-14 presents three such time-based metrics: Lock Waiting, Wasted Work, and Wait + Waste. The Lock Waiting graph shows average time in ms performed by ACBL to acquire locks, where this time includes both message delays due to lock requests and callbacks and blocking time due to data conflicts. The gray line in this graph shows the fraction of this time that is due to *wasted lock waiting* (lock waiting performed by aborted transactions). For UNIFORM, very little of the total lock waiting is wasted work.

The Wasted Work graph shows the average time per successful commit used to execute aborted transactions. AOCC has higher wasted work cost than ACBL, as we would expect. However, as we can see in the Wait + Waste graph, the combination of lock waiting and wasted work for ACBL is higher than the wasted work performed by AOCC. This combined metric allows us to compare the total concurrency-control costs incurred by each scheme: it shows that AOCC's abort-induced costs are lower than the combination of ACBL's locking-induced and abort-induced costs. (Wait + Waste is computed by adding lock waiting and wasted work and then subtracting the overlap; wasted lock waiting is not "double counted.")

In this case we could simply compare lock waiting to wasted work to see which scheme has lower concurrency control costs. (ACBL's lock waiting rises to 532 msec at 24 clients, while AOCC's wasted work rises to 363 msec at 24 clients.) However, ACBL's abort costs become non-trivial by 24 clients, and the Wait +Waste graph thus provides a better picture of the overall concurrency control costs incurred by each scheme.

Peak Throughput Results

Figure 5-15 gives the Server Utilization and Disk Utilization graphs for UNIFORM. Due to the high client and server miss rates under UNIFORM, both schemes consume more of the available server disk resources than server CPU resources. AOCC's higher fetches per commit (due to aborts) results in the use of more disk operations per commit, and AOCC also commits more transactions per second. These factors cause AOCC to drive the server disks towards saturation at a faster rate than ACBL. AOCC's peak throughput occurs at 12 clients, where its server disk utilization has risen to 84%. While an extra client gives the system the opportunity to perform more concurrent transactions (and thus to commit more transactions per second), it also causes *each* client transaction to incur more disk latency per commit, due to higher resource contention. Resource contention over the disks increases both because there are more clients vying for use the disks and because the increased abort rate results in an increasing number of disk operations per commit (in the 12–24 client region). This cost/benefit tradeoff balances out at roughly 12 clients, where we observe a peak throughput, and throughput degrades beyond this point as disk latency per commit continues to rise.



CURRENT System, Default Workload Settings

Figure 5-15. UNIFORM: Server and Disk Utilization

ACBL's throughput peaks due to a different tradeoff. Each additional client provides the potential for more commits per second, but it also results in more lock waiting time for *each* transaction in the system, due to both rising blocks per commit (more waiting time at the server) and to the increase in messaging costs (more messages are used, and the cost of using messages rises with a rising level of resource contention over the server CPU). As we saw in Figure 5-14, ACBL's lock waiting time rises rapidly with the number of clients in the system. This tradeoff balances out at roughly 16 clients, where ACBL hits its peak throughput value. In this case neither of the server resources has saturated. ACBL transactions spend an increasing amount of time blocked at the server as more clients are added to the system, and transactions do not consume resources while they are blocked. Note that ACBL's server CPU utilization has leveled off by 24 clients, while its disk utilization is actually dropping by 24 clients. Thus, ACBL is clearly suffering from "too much blocking" at a high number of clients.

Both schemes could benefit from an admission control policy for this workload. If ACBL allows "too many" clients into the system, this results in a level of data contention (and blocking) that results in less total work than can be achieved using a smaller number of clients. Similarly, if AOCC allows "too many" clients into the system, this results in a high level of disk resource contention that produces less total work than can be achieved at a smaller number of clients. Admission control would allow both schemes to have a total throughput that levels off near the peak throughput, rather than degrading beyond this point. If both schemes used an admission control scheme, the peak vs. peak gap would be stabilize at the current gap of roughly 6%.

While ACBL peaks later than AOCC, its throughput does not change significantly between the two peaks (from 12 to 16 clients). Both schemes have a drop in throughput beyond their peak values due to rising lock waiting (for ACBL), disk latency, and other costs. As the throughput values "degrade" at roughly the same rate, the percent improvement becomes relatively stable at 16 clients and beyond, remaining near the peak vs. peak percent improvement of 6%.

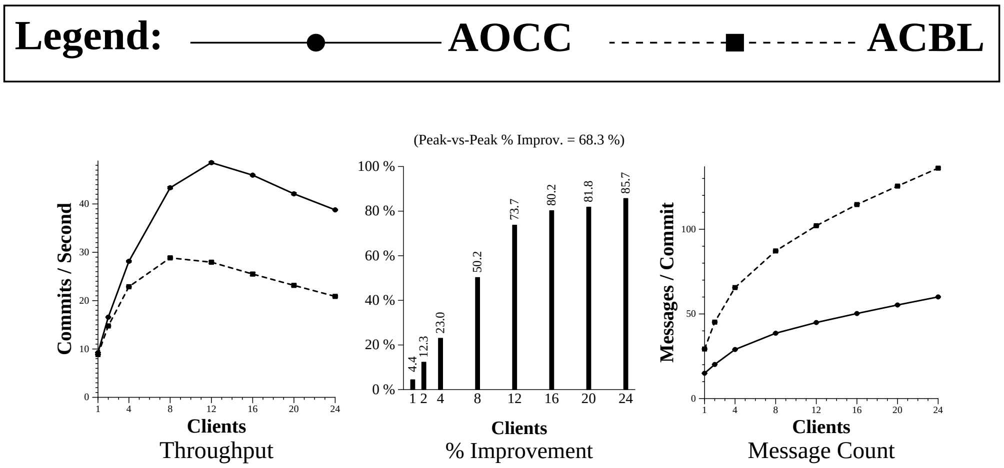
5.6 HICON Workload

The HICON workload divides its 1250 page working set into a 250-page *hot* (high-contention) region and a 1000-page *cold* (low-contention) region. 80% of all accesses go to the hot region, while the remainder go to the cold region. As with HOTCOLD, SMALL+HOTCOLD, and UNIFORM, HICON averages 200 accesses and 20 object updates per transaction.

Main Results

Figure 5-16 gives the Throughput, Percent Improvement, and Message Count graphs for HICON.

HICON has the highest contention of our six workloads; blocking and abort costs have a significant impact on system throughput even for small client cases.



CURRENT System, Default Workload Settings

Figure 5-16. HICON: Throughput, % Improvement, Message Count

Note that each update to a hot object O on page P causes all clients (other than the updater) to either mark O as unavailable or to drop P. The frequency of such marking and dropping increases with the number of clients in the system, causing the number of fetches per commit to rise for both schemes as more hot fetches occur per commit. AOCC's higher abort rate causes it to fetch more pages overall. The result of these hot fetches is that the pages sent per commit metric for HICON rises rapidly. AOCC's pages sent rises faster than ACBL's pages sent, but both schemes use a significant number of fetches. (AOCC's pages sent per commit at 24 clients is 28.9, compared to ACBL's 23 pages sent per commit.)

Although the number of page sends is high, HICON is not a disk-intensive workload, when compared to a workload such as UNIFORM. The hot pages are fetched frequently enough by the clients that they remain cached at the server, thus fetches for hot pages do not cause disk reads. Cold accesses that miss in a client cache do cause disk reads, but cold accesses make up only one fifth of the average transaction accesses.

Due to the relatively low disk costs, the differences between AOCC and ACBL have a high impact on their relative performance. Prior to the peak throughput points, Percent improvement rises from 4.4% to 50.2% from 1 to 8 clients. AOCC's throughput peaks later than AOCC's (at 12 clients rather than 8), causing percent improvement to rises rapidly from 8–12 clients, to 73.7%. Beyond 12 clients, AOCC's throughput "degrades" at a slightly slower rate than ACBL's throughput; percent improvement increases slowly from 12–24 clients, rising from 73.7% to 85.7%.

As with UNIFORM, AOCC's data contention costs are lower than ACBL's data contention costs. Both schemes have a higher abort rate under HICON. While AOCC has a much higher abort *rate* than ACBL, AOCC's cost *per abort* is much lower than ACBL's per-abort cost.

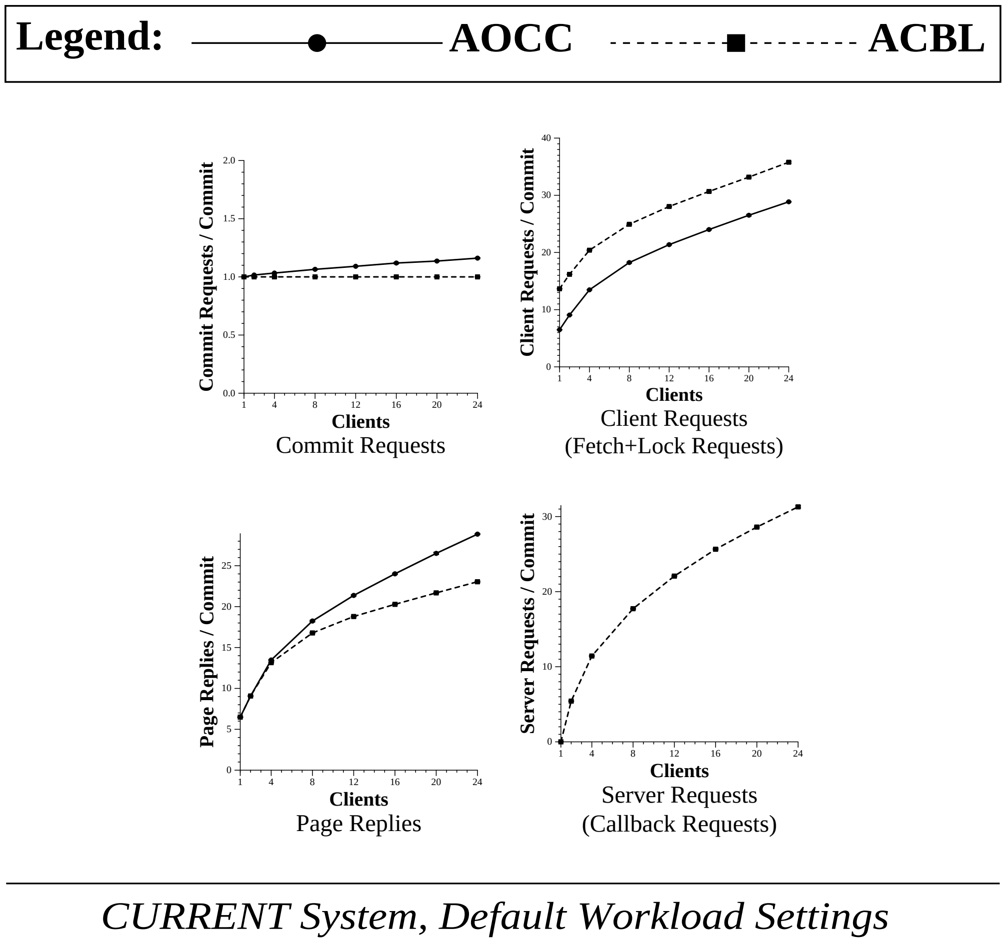


Figure 5-17. HICON: Message Breakdown

ACBL aborts result in more write lock requests per commit, since all write locks are dropped when a transaction aborts. Aborts also result in more combined fetch/read-lock requests, since some of the objects accessed during an aborted transaction are marked or dropped from the client cache prior to being re-accessed during a restart. ACBL's abort rate is much higher under HICON than under the four previous workloads, producing a new result: wasted lock waiting time becomes a significant component of the total lock waiting time.

As clients are added to the system and contention increases, two things occur: (1) the average time required for *each* lock acquisition increases, due to increased blocking and also to a higher level of server CPU contention; (2) the average number of lock requests per commit increases due to aborts, as discussed above. These two effects have a compounding effect, and cause a rapid increase in both the wasted lock waiting time and the total lock waiting time per commit.

The impact of data contention is much greater under HICON than under the previous workloads. One way to consider the impact of data contention is to measure

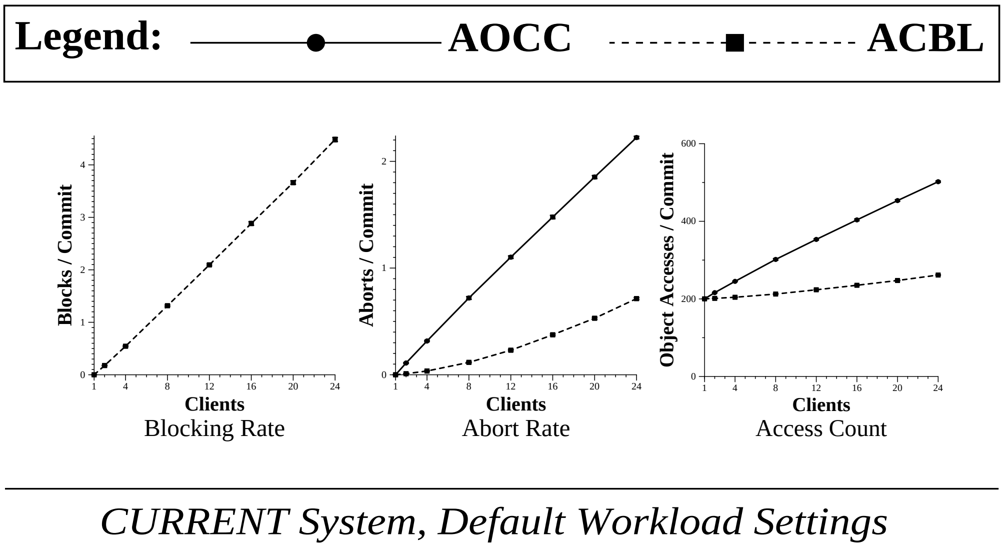


Figure 5-18. HICON: Blocks, Aborts, Accesses (per Commit)

the percent of the average transaction latency under AOCC that is due to wasted work (where the rest of the average latency is due to transaction executions that successfully commit). For SMALL+HOTCOLD, average AOCC wasted work rises to 24% of the average transaction latency by 24 clients; for HICON, wasted work rises to 61% of transaction latency.

Peak Throughput Results

Figure 5-20 gives the Server Utilization and Disk Utilization graphs. As discussed above, disk costs are low relative to server CPU costs, due to the fact that most fetch requests are for hot pages that are present in the server cache. Server CPU is thus the critical resource for AOCC.

Note that ACBL does not saturate either of the two server resources. Average blocking time at the server increases with added clients. Beyond 12 clients, the addition of new clients causes more additional blocking than additional useful work; overall resource utilization decreases due to this added blocking latency.

ACBL approaches this excessive blocking point faster than AOCC approaches server CPU saturation, causing ACBL's throughput to peak earlier than AOCC's throughput. Due to this difference, there is a rapid growth in percent improvement between 8 and 12 clients: as shown in the Percent Improvement graph in Figure 5-16, percent improvement rises from 50.2% to 73.7% in this client region. Beyond this region, ACBL's throughput degrades (due to excessive blocking) at a slightly faster rate than AOCC's throughput degrades (due to abort-induced saturation). Percent improvement rises slowly in the 12 to 24 client region (from 73.7% to 85.7%).

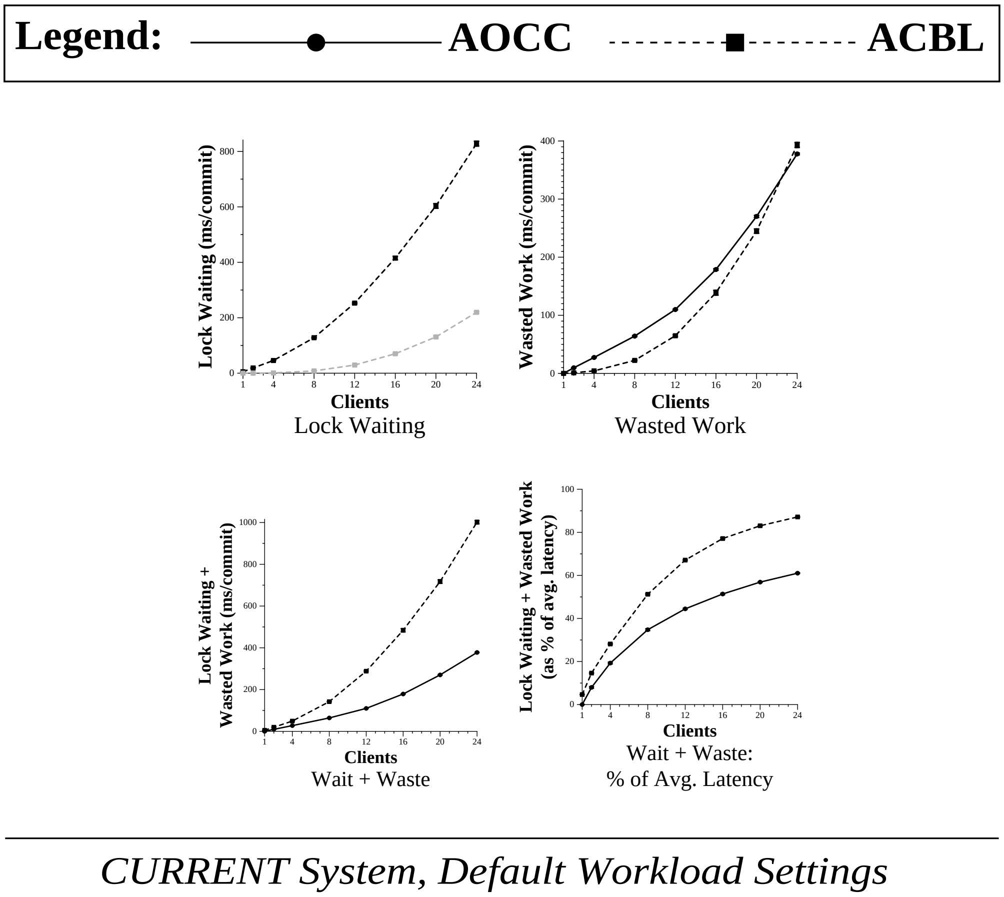


Figure 5-19. HICON: Lock Waiting and Wasted Work

5.7 TINY+PRIVATE Workload

The TINY+PRIVATE region was designed to examine the impact of very hot pages (pages with very high contention) that are accessed *infrequently*. It is identical to the PRIVATE workload, except a uniformly shared TINY region has been added, and 2% of the total accesses have been changed from other-region accesses to TINY accesses. (80% of the accesses are private accesses, 18% are other-region accesses, and 2% are TINY accesses.)

The default is for the TINY region to be exactly one page in size. Although only 2% of all accesses go to this region, its very small size makes it likely that transactions will perform conflicting object-level accesses. As a result, even though 98% of all accesses do not cause data conflicts, TINY+PRIVATE is a high contention workload.

This is a workload where ACBL's adaptive-granularity locking is very important. ACBL uses page-level locks for the private updates that each client performs, and object-level locks for all accesses to the the TINY region. Since most

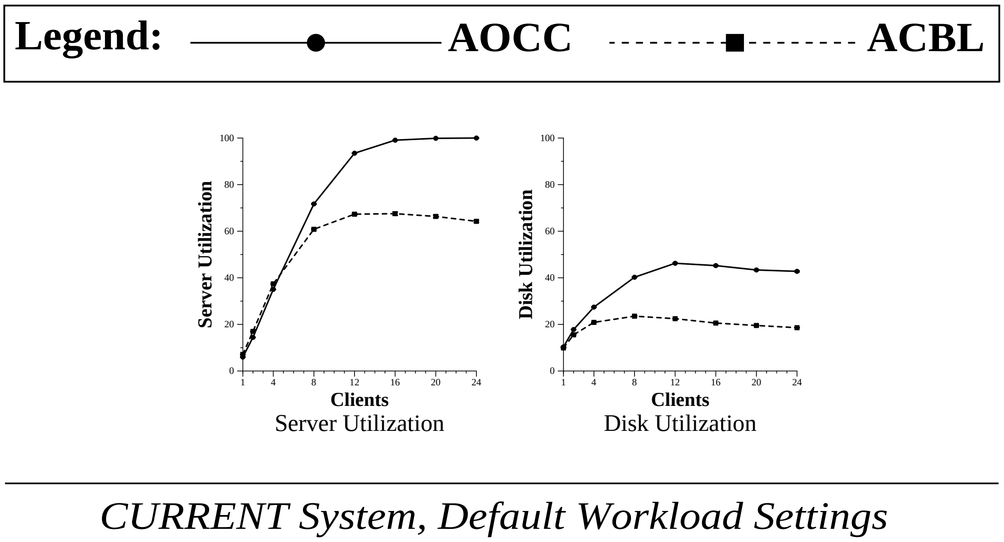


Figure 5-20. HICON: Server and Disk Utilization

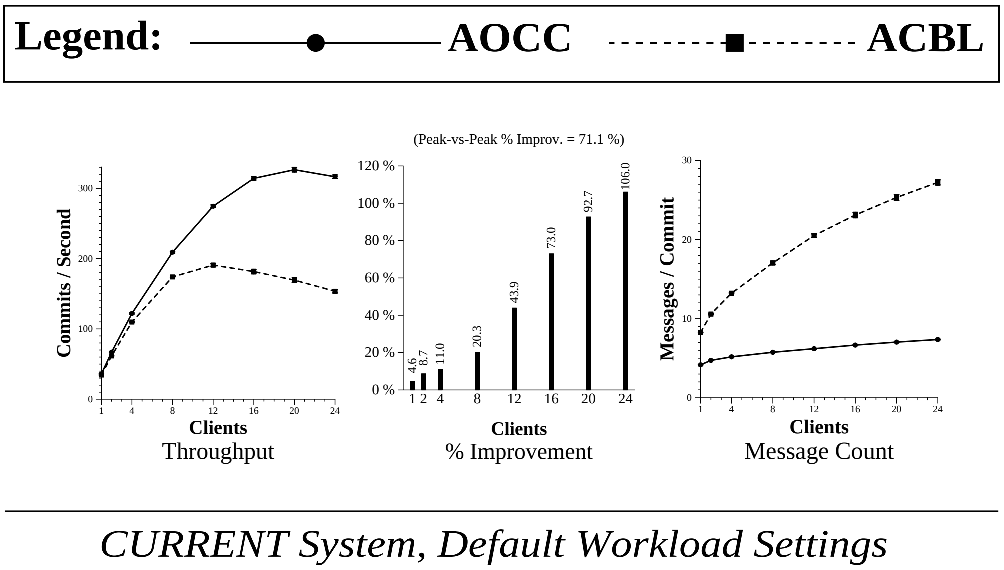


Figure 5-21. TINY+PRIVATE: Throughput, % Improvement, Message Count

TINY+PRIVATE transactions modify the TINY page, page-level contention between transactions is significantly higher than the actual object-level contention present in the workload. (At the page level, every transaction that modifies the TINY page is in conflict with every other such transaction.) Thus, compared to ACBL's blocking and abort rates as presented here, the blocking and abort rates for a static page-level locking scheme would be much higher; ACBL would significantly outperform a page-level scheme. ACBL would also outperform a static object-level locking scheme, as such a scheme would use more write lock requests for updates to private pages.

Figure 5-21 gives the Throughput, Percent Improvement, and Message Count graphs for TINY+PRIVATE. The results are similar to the HICON results. For both of these high contention workloads, AOCC's throughput peaks later and rises higher than ACBL's throughput due to the fact that AOCC's abort-based costs are lower than the combination of ACBL's lock waiting and abort-based costs.

Main Results

TINY+PRIVATE is designed to observe the impact of a single very hot (high contention) page on overall performance. The results for this workload are similar to the HICON results. Thus we learn that a very small region that is accessed infrequently (one page, accessed 2% of the time) can have roughly the same impact as a larger region that is accessed much more frequently (250 pages, accessed 80% of the time).

The results of HICON and TINY+PRIVATE are similar in the following ways:

- ACBL's abort rate becomes significant at higher clients. Due to the compounding effect of a rising abort rate and rising lock waiting times, ACBL's wasted work rises rapidly and eventually surpasses AOCC's wasted work.
- ACBL's combined Wait + Waste metric is always higher than AOCC's wasted work metric.

There are also differences, of course. The main difference is due to the fact that TINY+PRIVATE has a very low fetches per commit, due to a shorter transaction length and infrequent TINY accesses. Under AOCC, each fetch reply represents an opportunity for an AOCC client to "catch" the need to abort. With few fetches per commit, there are few opportunities to perform early aborts, and more aborts are therefore detected at the server rather than at the clients. Notice in the Commit Requests graph in Figure 5-22 that the commit requests per commit rises to nearly 1.5 requests per commit at 24 clients. (In contrast, under HICON the commit requests per commit approaches 1.2 at 24 clients.)

As a result, AOCC's special server mechanism that sends object updates in abort replies has a major impact on the caching behavior under TINY+PRIVATE. This mechanism is meant to reduce the number of fetches used during restart, but it is not used much for the other five workloads, since for these other workloads most aborts occur at the clients. For this workload, when an abort occurs at the server, updates for certain out-of-date TINY objects are returned in the abort reply (for any

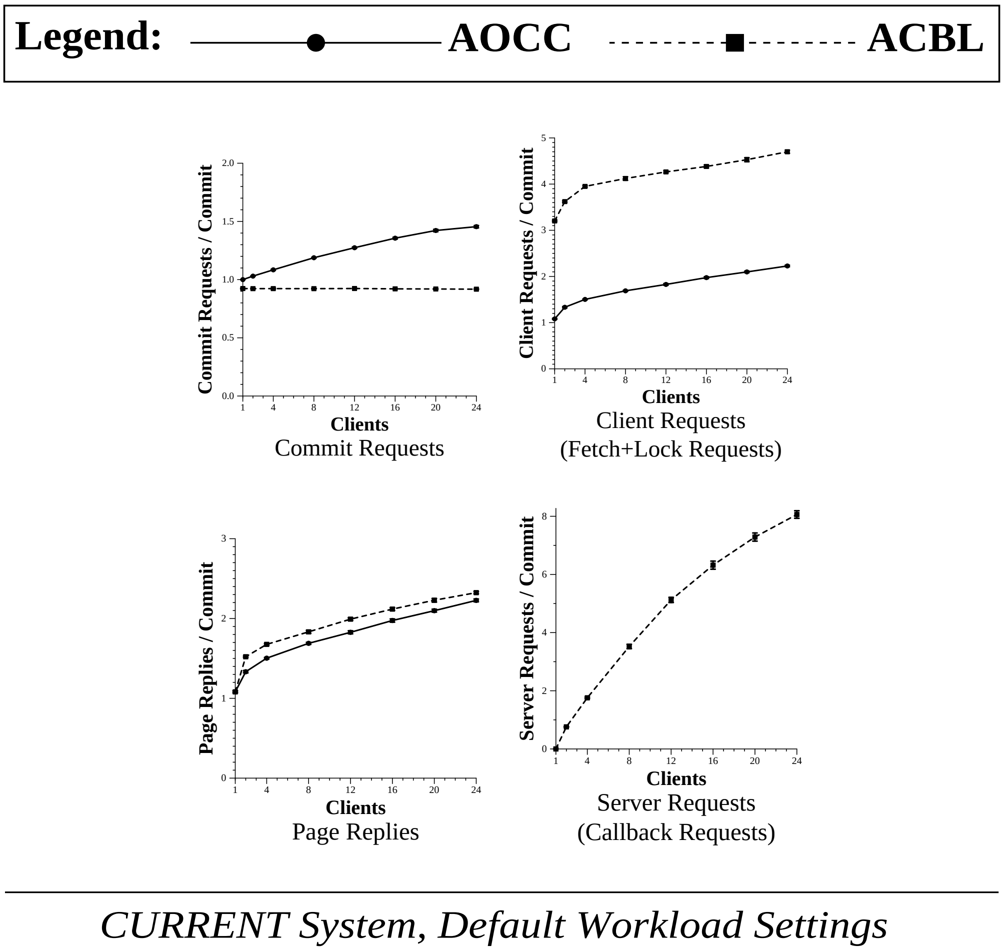


Figure 5-22. TINY+PRIVATE: Message Breakdown

TINY object that was read by the failed transaction and is currently out-of-date in the committing client's cache). Due to the updates sent in an abort reply, it is more likely that the resulting restart will not need to fetch the TINY page due to a missing TINY object, thus saving some fetches of this page.

The net result is that the combination of a commit request and abort reply can have the same impact as a fetch request and reply; failed commit requests tend to replace fetch requests. Note that abort replies do not count as page replies, since only updated objects are sent. AOCC's page replies per commit thus turns out to be *lower* than ACBL's page replies per commit, even though AOCC performs more accesses per commit.

Note in the Percent Improvement graph in Figure 5-21 that percent improvement does not rise rapidly until after 8 clients. This is due to the fact that AOCC's wasted work costs grow in roughly linear fashion, while ACBL's wasted work costs start low but rise rapidly once ACBL's abort rate starts to become significant. Thus we can see in the Wait + Waste graph in Figure 5-24 that the gap between AOCC's and

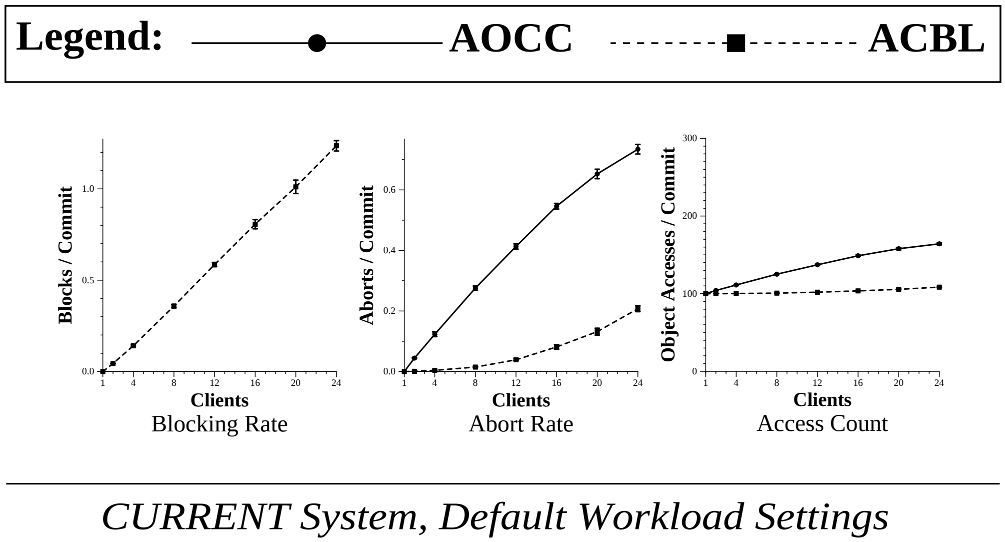


Figure 5-23. TINY+PRIVATE: Blocks, Aborts, Accesses (per Commit)

ACBL's combined metric only begins to widen rapidly beyond the 8 client point.

Peak Throughput Results

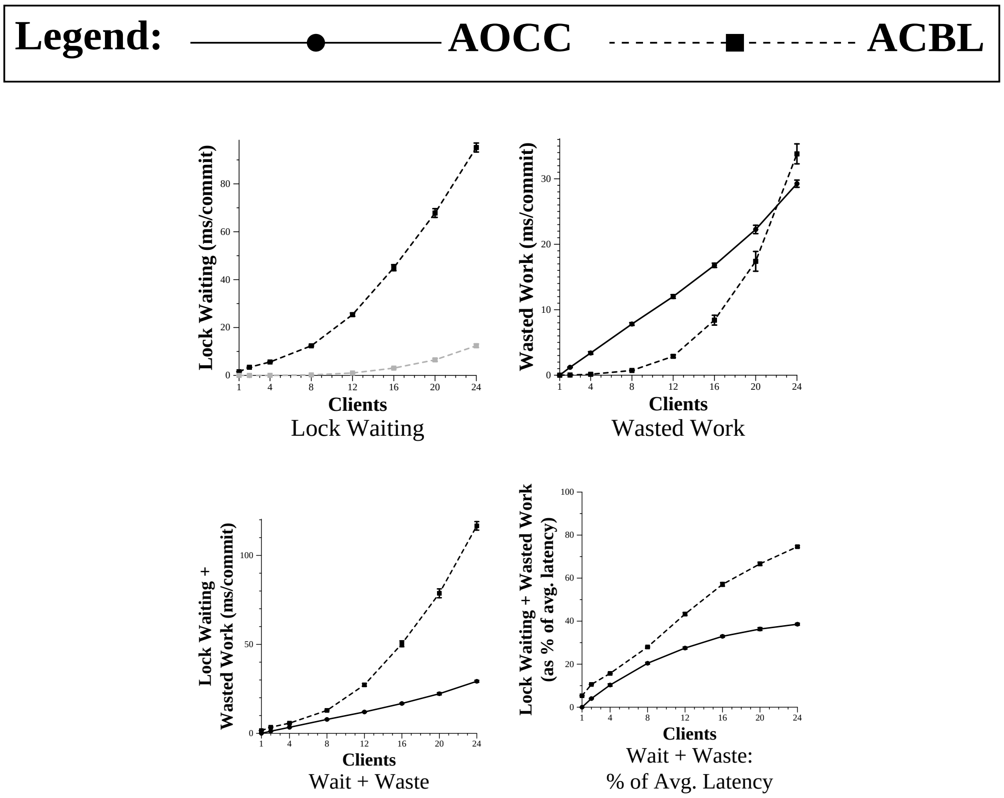
Figure 5-25 gives Server and Disk Utilization graphs for TINY+PRIVATE. As with HICON, AOCC uses more available server CPU than server disk, and the server CPU is the critical resource. ACBL also uses more server CPU than disk, but reaches a client point where excessive blocking limits its resource utilization; it does not saturate either server resource.

Because AOCC approaches server CPU saturation at a slow rate compared to the rate that ACBL approaches its excessive blocking point, AOCC's throughput peaks later than ACBL's throughput: ACBL peaks at 12 clients, while AOCC peaks at 20 clients. Due to this difference, there is a rapid increase in percent improvement above eight clients, as shown in the Percent Improvement graph in Figure 5-21.

5.8 Summary of Key Insights

In this chapter we presented results for our CURRENT system settings and default workload parameter settings. The system parameters were chosen to represent a reasonable model of a present-day system, and the workloads were chosen to allow us to explore a range of different access patterns and contention levels. AOCC consistently outperforms ACBL for this main set of experiments. In this section we summarizes the key insights that explain AOCC's better performance.

In Chapter 2 we model average transaction latency using the following formula:



CURRENT System, Default Workload Settings

Figure 5-24. TINY+PRIVATE: Lock Waiting and Wasted Work

latency = success-latency + failure-latency
success-latency = $E_s + F_s + L_s + B_s + C_s$
failure-latency = $E_f + F_f + L_f + B_f + C_f$

where

E = Execution latency at client for local read and write accesses
F = Fetch latency for fetch requests/replies, including disk read latency
L = Lock request latency for lock and callback requests/replies
B = Blocking latency at server due to object-level conflicts
C = Commit latency for commit requests/replies and processing at server

We now use this model to discuss the results reported above. ACBL invests heavily in lock and callback messages and in blocking (for higher contention cases): its L_s and B_s costs are high. The motivation for this approach is that this locking investment will pay dividends in the form of reduced failure costs, since locking keeps the abort rate low.

In contrast, AOCC is designed to minimize message costs, possibly at the expense

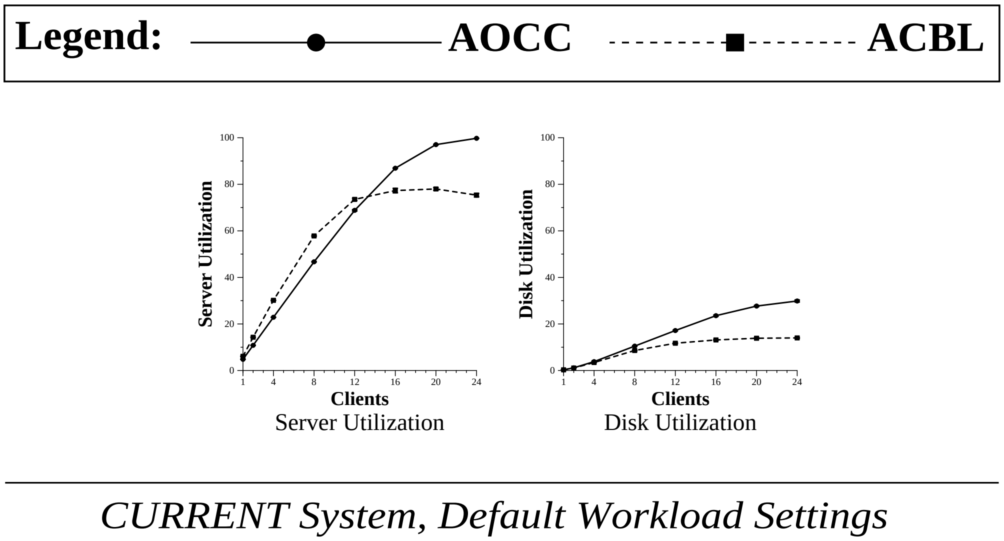


Figure 5-25. TINY+PRIVATE: Server and Disk Utilization

of higher failure costs due to a higher abort rate. Some aborts are avoided by using piggy-backed invalidations to keep client caches "nearly" up-to-date. This approach is effective with respect to eliminating many unnecessary aborts due to "sequential sharing" (where a transaction that updates object X commits before a transaction that reads X begins). However, concurrent or near-concurrent data conflicts do occur; all optimistic schemes will have a high abort rate when there is high contention in the workload. For this reason, AOCC is designed to minimize failure costs, with an emphasis on keeping total fetch cost (F_s + F_f) and total commit cost (C_s + C_f) from rising rapidly with an increased abort rate.

AOCC has three mechanisms designed for keeping failure costs low: the use of early abort detection at the clients avoids unnecessary commit requests (and also offloads validation work from the server); the use of the undo log to restore modified objects on abort avoids some unnecessary fetches during restart; the use of object updates in abort replies also avoids some fetches during restarts. These three mechanisms are very effective. Our results show that most aborts are early aborts that are detected at clients. As a result, total commit cost (C_s + C_f) is almost unchanged with an increasing abort rate. For workloads with very few fetches per commit (namely TINY+PRIVATE) the number of commit requests does increase. However, as more aborts occur at the server, more object updates are sent in abort replies, and fetch costs decrease.

With respect to fetch costs, the combination of the three mechanisms described above results in a total fetch and disk read costs per commit (F_s + F_f) that remains very close to ACBL's fetch and disk read costs under low contention and reasonably close to ACBL's costs under high contention (even though AOCC has a significantly higher abort rate).

ACBL's cost for lock and callback messages (L_s + L_f) results in total message costs that are high compared to AOCC's message costs, where the message gap increases with number of clients. Moreover, as contention increases, ACBL has rising blocking costs. Successful blocking (B_s) increases steadily across the entire client range, while failed blocking (B_f) rises rapidly at the high end of the client range, where aborts due to deadlocks become a significant factor. Aborts also result in wasted lock message costs (L_f). AOCC's failure cost per commit are very low, as AOCC has no B_f or L_f costs. For the two highest contention workloads, HICON and TINY+PRIVATE, ACBL's failure *costs* can be higher than AOCC's failure costs, even though ACBL's abort *rate* remains much lower.

In summary, ACBL's investment in locking does not pay sufficient dividends, regardless of contention level. This chapter demonstrates this result for our default settings. The next chapter shows the robustness of this result across a wide range of settings.

Chapter 6

Sensitivity Analysis

In the previous chapter (Chapter 5) we showed that AOCC outperforms ACBL when we use our default system and workload settings. This chapter shows that this result is robust: we present a set of sensitivity analysis experiments which show that the relative superiority of AOCC holds across a wide range of parameter settings.

Section 6.1 presents experiments that vary parameters of the system model, while Section 6.2 presents experiments that vary parameters of the workload model. Across these two sections, the performance gap (the percent improvement of AOCC over ACBL) can narrow or widen as parameters are varied. Normally, AOCC remains the better scheme across an entire parameter range; the two schemes sometimes become roughly equal at "extreme" parameter settings. One notable exception is presented in Section 6.2.2: ACBL can significantly outperform AOCC for a workload where most or all transactions are read-only and there is very little disk traffic per commit.

Neither Section 6.1 nor Section 6.2 includes an experiment where ACBL significantly outperforms AOCC on a read-write workload. However, the experiments that show a narrowing performance gap "point" towards a region of the overall parameter space where ACBL should be the better scheme. Section 6.3 characterizes this region, which we refer to as the "ACBL Wins" region. It also gives simulation results supporting this characterization, and argues that this region represents an unlikely scenario.

In addition to showing the robustness of our results, the sensitivity analysis experiments provide new insights into AOCC vs. ACBL tradeoffs. These insights and the key results from the experiments are summarized in Section 6.4.

6.1 System Model Experiments

This section describes system model experiments. Section 6.1.1 presents sensitivity analysis experiments for the core system parameters (disk and network bandwidth, processor speed, *etc.*). These experiments show the effect of varying a single parameter at a time. Section 6.1.2 examines the impact of changing all of the core system parameters together, moving from the default settings to a set of "future system" settings.

6.1.1 Core System Parameters

Eight sensitivity analysis experiments are used to examine the impact of changing the core system parameters, *i.e.*, the parameters for the network, disk, client, and server models. A pair of figures describe the setup and results for these experiments: the table in Figure 6-1 describes the parameter range used for each experiment and summarizes the percent improvement results observed across this parameter range, while Figure 6-2 presents eight throughput graphs, one for each experiment.

These experiments use the SMALL+HOTCOLD workload. We chose this workload because it is the best representative of a "real" OODB workload: we expect real applications have low to medium contention and exhibit both uniform and skewed sharing patterns. A 20% net object write probability and average write clustering is used. Settings other than the parameter under study are all set to default values. This section reports on results for the 12-client case. We also performed 6- and 24-client experiments; the same relative results hold for these other client cases.

Each row in the Figure 6-1 table describes one of the eight experiments. The first column indicates the parameter that is varied. The second column indicates the parameter range, where range X $\rightarrow$ is expressed such that the range moves from "worse" setting X to "better" setting Y. Thus, we move from smaller to larger main memory data structures, from slower to faster processors, from high to low disk access times, and from high to low CPU charges. The third column summarizes percent improvement results for this parameter range: $P_X \rightarrow P_Y$ gives the percent improvement at X and Y, respectively. Thus, the third column shows whether the performance gap narrows or widens as the parameter moves to a "better" setting.

The overall result is that relative performance does change as the core system parameters are varied, but AOCC remains the better scheme. In five cases the performance gap widens as a parameter shifts to an improved setting, while in three cases it narrows. The performance gap widens with larger client or server caches, faster client CPUs, lower disk access times, and higher network bandwidth. The gap narrows with a faster server CPU and lower (fixed or variable) network CPU charges.

AOCC is more sensitive to changes that affect the server disks: improvements to disk speed or reductions in disk use widen the performance gap. ACBL is more sensitive to changes that affect the server CPU: a faster CPU or reductions in CPU use narrow the performance gap. These sensitivity results are due in part to actual use of each resource per commit: additional fetches due to restarts cause AOCC to have a slightly higher disk use per commit, while lock and callback requests cause ACBL to have a much higher server CPU cost per commit. In addition, relative use of the two resources affects sensitivity. For example, since AOCC has low server CPU costs due to a low message count, its use of the disks is a relatively larger fraction of its total costs. Therefore, even if the two schemes had *identical* disk cost per commit, AOCC would still be more sensitive to changes that affect the disks and ACBL would still be more sensitive to changes that affect the server CPU.

Server CPU Speed (See top-right graph, Figure 6-2.) ACBL is more sensitive to server CPU; increasing the server CPU speed narrows the performance gap.

Parameter(s)	Settings Used	Resulting % Improv. SMALL+HOTCOLD (12 Clients)
Client Cache Size	5% → 50% of DB	14.5% → 23.1%
Server Cache Size	10% → 100% of DB	9.3% → 31.5%
Client CPU Speed	15 → 200 MIPS	19.8% → 32.0%
Server CPU Speed	30 → 400 MIPS	39.4% → 13.3%
Slow Disk Access Time for 4 KB Page	20 → 2 msec	16.7% → 34.2%
Network Bandwidth	4 → 800 Mbps	1.6% → 22.2%
Var. Network CPU Cost	7168 → 128 instr./KB (fixed net. CPU = 250 instr.)	8.0% → 5.6%
Fixed Network CPU Cost	6000 → 250 instr. (var. net. CPU = 128 instr./KB)	18.2% → 5.6%

Figure 6-1. Core System Parameters: Summary of Experiments

Disk Access Times (See middle-right graph, Figure 6-2.) For the disk access time experiment, we vary both the slow and fast disk access times using a fixed ratio of 0.38. This ratio approximates the fast:slow ratio used for the CURRENT and FUTURE disk settings. AOCC is more sensitive to server disk; faster disks increase the performance gap.

Client and Server Caches (See bottom row, Figure 6-2.) A larger client cache means fewer fetches (and thus fewer disk reads). A larger server cache reduces the number of fetches that require disk reads. In both cases, allocating more cache space reduces disk costs. Since AOCC is more sensitive to this change, the performance gap widens.

Client CPU Speed (See middle-left graph, Figure 6-2.) Increasing the client CPU while holding the server CPU constant causes a higher load on both the server CPU and server disks. A similar increase in load occurs when the client CPU charges (or "think times") for object accesses are lowered; think times are examined in Section 6.2.5. As disussed in Chapter 2, for workloads where contention is not so high that ACBL performs "too much blocking," the impact of AOCC's extra fetches is smaller than the impact of ACBL's extra messages. As a result, increasing the load on both server resources has a higher relative impact on ACBL, and the performance gap widens.

Network Bandwidth (See center graph, Figure 6-2.) For this experiment, a bandwidth less than 20 Mbps results in high network contention. Below this bandwidth point, delays due to network contention are a large component of average transaction

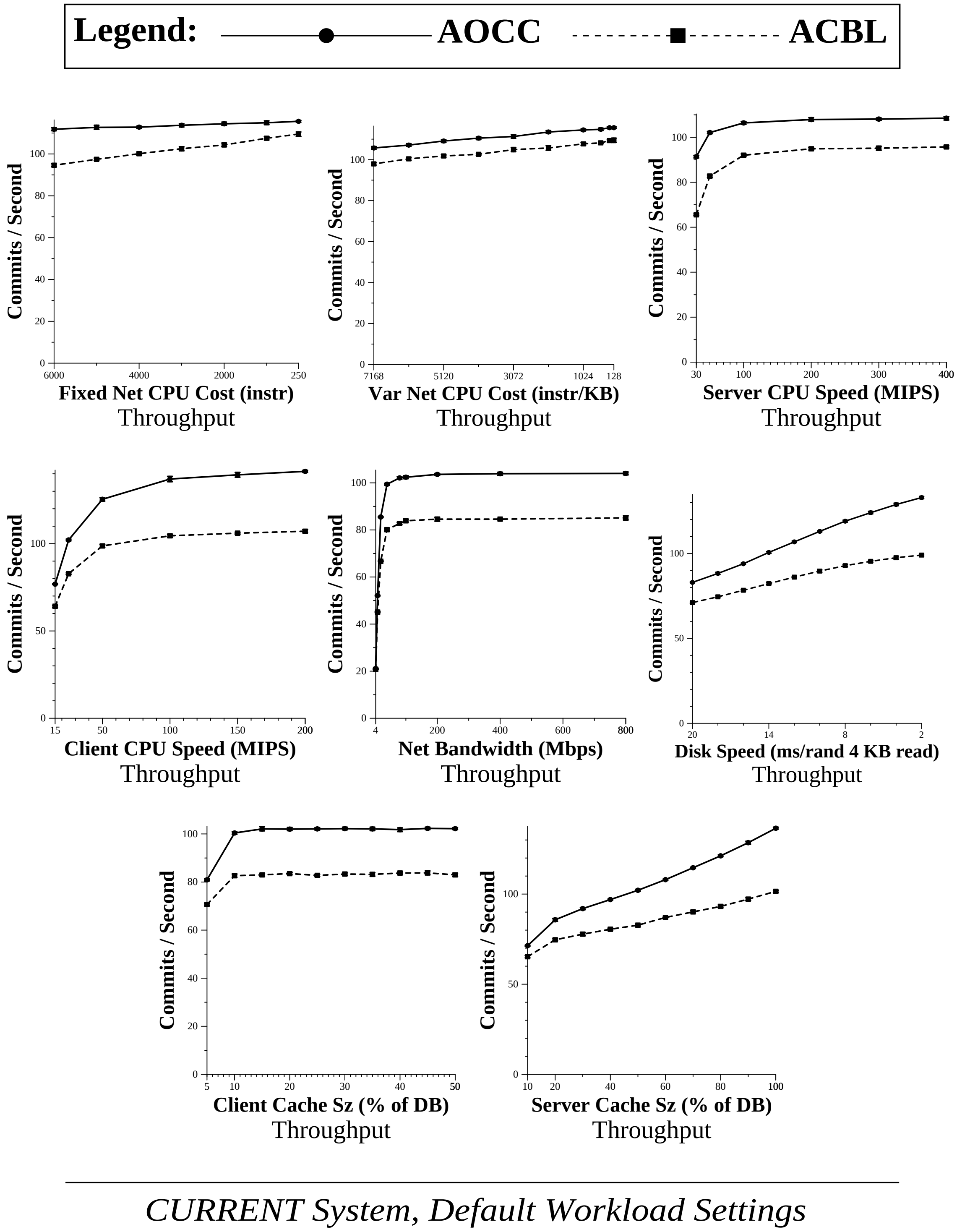


Figure 6-2. Core System Parameter Experiments: Throughput Results

latency. Since both schemes incur these delays, the performance gap is small in this region. AOCC has a slightly higher *bytes per commit*; it sends fewer messages, but uses piggy-backed information and read set information not used by ACBL. This byte-count difference has an impact at very low bandwidth. Thus, in the low bandwidth region an increasing bandwidth causes a widening performance gap. Above 20 Mbps, however, further increases in bandwidth have little impact on relative performance.

It is important to note that AOCC's higher byte-count is much less significant than ACBL's higher message count. For example, for the 12 client SMALL+HOTCOLD case, AOCC sends 23,607 bytes using 11.0 messages while ACBL sends 22,417 bytes using 41.2 messages (per commit). Thus, AOCC uses 5% more bytes, but ACBL uses 275% more messages.

Network CPU Costs (See top-left and top-middle graphs, Figure 6-2.) For the two experiments involving the fixed and variable network CPU charges, in each case one parameter is varied and the other is set to a low constant value so that the overall CPU charge will be controlled by the parameter under analysis.

Reducing either the fixed or variable network CPU charge results in less server CPU overhead per message. ACBL is more sensitive to changes in server CPU use. Moreover, since CPU charges at both the client and server are lower, latency for a round-trip client-server exchange is reduced. Compared to AOCC, ACBL uses more round-trip exchanges per commit; ACBL is more sensitive to round-trip latency. Due to both of these factors, the performance gap narrows.

The top-left and top-middle graphs in Figure 6-2 show that the impact of lowering the fixed CPU charge is greater than the impact of lowering the variable CPU charge. ACBL's higher message costs are due to lock and callback messages. Since these messages are small, the fixed CPU charge is the more significant of the two network charges.

6.1.2 Future System Results

Of the eight experiments described above, five show that moving a single system parameter (or a pair of related parameters) towards a "better" setting causes AOCC's percent improvement over ACBL to increase, while three experiments show the opposite result. These "better" settings describe a likely future system: we expect system resources will become faster (larger, *etc.*) over time[1]. Thus, we decided to explore the impact of switching all of the system parameters to a set of values that represent one possible "future" system.

Figure 6-3 summarizes the parameter settings for the CURRENT and FUTURE systems. Parameters not listed are set to their default values (as given in Section 4.1). Note that the third column in Figure 6-3 shows the range of values used in the single-parameter sensitivity analysis experiments presented in Section 6.1.1.

[1]Chapter 7 considers the impact of moving to a wide-area network (WAN), where latency will be higher rather than lower.

Parameter	CURRENT	FUTURE	Section 6.1.1 Experiment
Server CPU speed	50 MIPS	200 MIPS	30–400 MIPS
Client CPU speed	25 MIPS	100 MIPS	15-200 MIPS
Network bandwidth	80 Mbps	160 Mbps	4–800 Mbps
Fixed network cost	6000 instr.	3000 instr.	6000–250 instr.
Variable network cost	7168 instr./KB	2048 instr./KB	7168–128 instr./KB
Slow disk bandwidth	3322 μsecs/KB	2580 μsecs/KB	5000–500 μsecs/KB
Fast disk bandwidth	1288 μsecs/KB	990 μsecs/KB	1900–190 μsecs/KB
Number of server disks	4	8	—

Figure 6-3. CURRENT and FUTURE Parameter Settings

Moving from the CURRENT to FUTURE system parameters, CPU speeds quadruple, network bandwidth doubles, network CPU charges drop by more than half, the number of disks per server doubles, and the speed of each disk improves (average access time is roughly 22% lower). Appendix A describes the approach we used to choose these settings.

We experimented with these FUTURE parameter settings with all six workloads, using a number of different cluster and object write probabilities. One important change is that the ratio of client to server resources has shifted: for the FUTURE system, the server is relatively more powerful, allowing it to support more concurrent clients: for the 1–24 client range, none of the six workloads causes AOCC or ACBL to fully saturate any resource. (For the CURRENT system, AOCC always saturates one of the two server resources in the 1–24 client range.)

As we move from CURRENT to FUTURE, one of two changes occurs. For low contention cases, the performance gap between AOCC and ACBL becomes smaller, while for higher contention cases, the performance gap becomes larger.

For low contention workloads (and for the low-client cases of all the workloads) message use is the key difference between AOCC and ACBL. Unlike the CURRENT system experiments, under the FUTURE settings message costs do not "blow up" in the 1–24 client range due to server CPU saturation. The same increasing message-count gap occurs, but the resulting performance gap (as measured by the percent improvement metric) is smaller.

For high contention workloads, AOCC's abort costs grow with data contention level (increasing abort rate) and resource contention level (increasing cost of using the server CPU or server disks). The FUTURE system can support more concurrent clients than the CURRENT system (prior to resource saturation). As a result, while AOCC's abort rate grows at the same rate under both systems, its abort costs are relatively lower, and AOCC's throughput peaks at a higher number of clients.

The increased server resources of the FUTURE system produce a higher ACBL throughput at each client point. However, unlike AOCC, the number of clients required for ACBL to reach a peak throughput does not shift to a much higher client point. ACBL's high-contention peak occurs due to "too much blocking," and the server resources do not saturate (for *either* the CURRENT or FUTURE systems). Compared to AOCC, the location of ACBL's peak throughput is relatively more dependent on contention level (which is based only on number of clients) and relatively less dependent on available server resources. As a result, the increase in server resources has only a small impact on the number of clients ACBL can handle before it reaches its peak throughput.

In summary, with high contention ACBL does not capitalize on the FUTURE system's ability to support a larger number of concurrent clients, while AOCC does. The result is a rapid widening of the performance gap beyond the point where ACBL nears its peak throughput.

The above insights are based on examining the six workloads using a number of different write probabilities. To demonstrate these insights, we present one low-contention example and one high-contention example.

Figure 6-4 gives an example of the general low-contention result: throughput and percent improvement graphs are given for the HOTCOLD workload, for both the CURRENT and FUTURE settings. In this case, the FUTURE system percent improvement is lower than the corresponding CURRENT system percent improvement across the entire 1–24 client range. (CURRENT percent improvement falls between 6.7% and 12.8%, while FUTURE percent improvement falls between 2.1% and 5.6%.)

Figure 6-5 gives an example of the general high-contention result, showing the same graph types for the HICON workload. Moving from CURRENT to FUTURE, note the change in the shape of AOCC's throughput curve: the peak throughput shifts from 12 clients to 20 clients. ACBL's throughput curve has roughly the same shape in both cases, where peak throughput moves from 8 to perhaps 10 clients (the 8 and 12 client points for the FUTURE curve are at the same level, suggesting the peak is between these points).

Under the CURRENT system, percent improvement does "stabilize" above 16 clients. Under the FUTURE system, due to the shift in the location of AOCC's peak, percent improvement continues to rise through 24 clients. The lower latency and lower resource utilization produced by the new FUTURE system settings have an impact even for this high contention example: the FUTURE percent improvement is somewhat smaller than CURRENT percent improvement from 1–12 clients. At 16 clients and above, however, we see the impact of the shift in AOCC peak throughput: FUTURE percent improvement becomes larger than CURRENT percent improvement.

Across all of our simulation experiments, the FUTURE experiments produced some of the highest performance differences. *E.g.*, for this HICON experiment, percent improvement rises to 125% at 24 clients; AOCC more than doubles ACBL's throughput.

6.2 Workload Model Experiments

This section describes workload model experiments. Section 6.2.1 examines varying both write probability and write clustering; Section 6.2.2 examines varying the mix of read-only vs. read-write transactions in the workload; Section 6.2.3 examines varying the restart behavior; Section 6.2.4 examines varying the average transaction length and also the varying the range of transaction lengths (the distance between shortest and longest lengths); Section 6.2.5 examines varying the local client computation time per access (the "think" time).

6.2.1 Write Probability, Write Clustering

Experiments were performed across all six workloads to examine the impact of varying the write probability and the quality of write clustering. In addition, for each combination of write probability and write clustering, we used a range of client cases (1–24 clients). All of these experiments show the same relative results: ACBL is more sensitive to update frequency, contention level, and the quality of write clustering. This section uses results from HOTCOLD workload experiments to demonstrate these points. Since ACBL is more sensitive to contention, workloads with higher contention produce larger changes as write probability and write clustering are changed. HOTCOLD was chosen as the example workload to demonstrate that significant changes in performance occur even for a low contention workload.

Figure 6-6 shows the result of varying the write probability and also of varying the quality of write clustering. The six result graphs shown are for the HOTCOLD workload and CURRENT system settings. For each row in the graph, we show three different combinations of cluster write probability and object write probability that produce a 10% net write probability (top row) or 20% net write probability (bottom row). Moving from left to right across a row, the net write probability does not change: the three experiments in each row have the same average number of object writes per commit. However, the average number of *page updates* does change. For the left column, roughly one fifth of all accessed pages are updated; for the middle column, roughly half of the pages are updated; for the right column, most of the accessed pages are updated. Moving from top graph to bottom graph in each column, the opposite conditions hold: the average number of page updates remains fixed, while the average number of object writes doubles.

Each of the six graphs shows results for 1–24 clients, and in each graph we observe the same overall result shown in Chapter 5, namely that the performance gap increases with number of clients, until the point where both schemes are nearing their peak values.

Moving top-to-bottom, at each client point, the increase in write probability causes ACBL's message costs to increase (more lock requests are used, and moreover more of these requests result in object-level locks). More blocking also occurs. AOCC's abort rate rises, but its wasted work costs do not rise as rapidly as ACBL's locking costs. As a result, doubling the write probability increases the performance gap (at every client point).

Note that it is possible to increase contention level by increasing the number of clients or by increasing the write probability. Our results show that both cases normally cause the performance gap to widen. The performance gap does not necessarily widen across the entire range of clients or write probabilities used, however. For increasing clients, resource saturation (or excessive blocking) places limits on peak throughput, as we show in Chapter 5. For increasing write probability, the same effect can occur.

Moving left-to-right, we find (at each client point) that decreasing the quality of write clustering results in a larger performance gap. This is a new result (not shown in Chapter 5): AOCC is less sensitive to how well the static placement of objects onto disk pages matches the temporal grouping of write accesses within transactions. (ACBL uses more write lock requests per transaction as we decrease the quality of write clustering.)

Since throughput peaks can occur at different client points, it is also interesting to compare peak vs. peak percent improvement. The following table gives this metric for the six experiments in Figure 6-6. Note that AOCC's peak throughput advantage over ACBL increases with either an increase in net write probability or a decrease in quality of write clustering.

Net Write Probability	**Quality of Write Clustering**		
	Good	**Average**	**Poor**
10%	4.7%	10.4%	16.4%
20%	4.9%	12.4%	23.4%

Peak vs. Peak Percent Improvement

6.2.2 Read-Only/Read-Write Mix

As discussed in Section 2.5.3, for a single-server setting, both AOCC and ACBL can commit read-only transactions without contacting the server. However, in a multi-server setting AOCC does need to use a commit request (or set of commit requests) to validate the read set of a read-only transaction. Although our simulation study uses a single server, we are interested in supporting the multi-server case. Therefore, our implementation of AOCC for this study does use a commit request for read-only transactions. This section examines the impact of this additional round-trip exchange on the relative performance of AOCC and ACBL. (For a transaction where just one object is written, ACBL uses more messages than AOCC; it is only the full read-only case where ACBL has a lower message count.)

Since read-only and read-write transactions do interact, we decided to explore the impact of varying the "mix" of read-only vs. read-write transactions in the workload. This is accomplished using the *percent forced read-only* parameter in our workload model. This parameter is normally set to zero percent; we normally do not "force" any transactions to be read-only. If set to X%, the workload generator ignores the write probability settings for X% of the transactions generated (and simply produces read-only transactions).

The experiments presented in this section vary the percent forced read-only from 0% to 100%. We chose SMALL+HOTCOLD for this set of experiments because, as mentioned earlier, it is the best representative of a "real" OODB workload. At 0% forced read-only we obtain the our standard result (AOCC outperforms ACBL); at 100% read-only we know that ACBL has a lower message count and should outperform AOCC. There are two interesting questions, then: where does the crossover in performance occur, and what is ACBL's peak advantage over AOCC (for the 100% read-only case)?

Figure 6-7 shows the result of varying the mix of read-only and read-write transactions for the SMALL+HOTCOLD workload with default settings. Transactions that are not forced to be read-only use a net object write probability of 20% (with average write clustering). The 1, 12 and 24 client cases are shown. The X axis is the percent forced read-only, while the Y axis is commits per second. (Note that the scale of the Y axis is different for the 3 graphs : the peak throughputs achieved at 1, 12 and 24 clients are 16, 146, and 171 commits per second, respectively.)

For the 12 and 24 clients cases, ACBL equals AOCC's throughput at a workload mix of 98% read-only transactions. (For the 1 client case, the crossover occurs with a 90% read-only mix, due to the fact that ACBL uses no callbacks.) The throughputs match at the point where AOCC and ACBL messaging costs are roughly equal. As we can see from this experiment, the workload must consist almost entirely of read-only transactions for this case to occur.

The small percent improvement shown for ACBL is due to the fact that the average latency of a SMALL+HOTCOLD transaction is much higher than the latency of a round-trip commit request, due to the client CPU time and fetch costs that result from performing 200 object accesses (on average). If transactions were shorter and did not perform any disk reads as part of fetching, we would see a much larger advantage for ACBL at 100% read-only. To show this case, we ran a special SMALL+HOTCOLD experiment where the think times are reduced by a factor of 10 and the server cache is set to 100% of the database size. The client cache size remains at 25% of database size: fetches can still occur, but no disk reads occur. Due to these changes, transaction latencies are much lower, making the extra commit request used by AOCC relatively expensive.

Figure 6-8 shows the throughput and percent improvement graphs for these experiments. There is a crossover in throughput somewhere between 80% and 100% forced read-only. With a fully read-only workload, ACBL's outperforms AOCC by 25.2%.

In summary: ACBL can outperform AOCC due to read-only transactions, but there are two requirements for read-only transactions to be a significant factor: most transactions in the workload must be read-only; average transaction latencies for the execution of object accesses must be low relative to a round-trip commit latency. For the latter case to occur, there must be very little disk activity per commit: almost all accesses must hit in the client or server caches.

Recall that the comparison performed in this section is only relevant for a multi-server system. However, our experiments use only one commit request, while several servers can be contacted in the multi-server case. Thus, the crossover point and percent improvement results given here are only approximate results; further study of the multi-server case is needed. In addition, we believe it is possible to design an optimistic scheme for the multi-server case that normally does not need to use a commit request for read-only transactions. Studying this issue is an area of future work.

6.2.3 Restart Behavior

Most concurrency-control simulation studies (including the client-server studies most similar to our work [10, 23, 24, 57, 58]) use a "perfect restart" assumption: on an abort, the access pattern of the failed transaction is used without modification for the subsequent restart execution.

Even if exactly the same transaction code is re-executed for a restart, the restart execution does not necessarily access the same set of objects in the same order. *E.g.*, consider a search tree whose nodes are all represented as individual objects within the database. Suppose a transaction performs a traversal from the root node to a leaf node, selecting branches according to a search criteria. If this transaction fails and restarts, re-execution (using the same root and search criteria) can result in a different traversal of the tree: the traversal will normally follow the same path, but the new traversal can "diverge" from the original traversal at any node in the original path whose state has changed since the previous traversal visited the node.

In the above example, encountering a modified object can cause a *suffix* of the original access sequence to be replaced with a new set of object accesses. Our simulator models this suffix-replacement scenario. A restarted transaction *begins* with the same access sequence. However, when object X is accessed during a restart, if X's value has changed since it was accessed during the previous (failed) execution, the remaining accesses in the sequence are replaced with new accesses, with probability P. The original accesses continue to be used with probability (1-P). If the non-replacement case is selected, another suffix-replacement decision occurs if another changed object value is encountered. We refer to probability P as the *restart change probability*.

A restart change probability of 0% is the "perfect restart" assumption: there is zero chance that a modification will cause the access sequence to change during restart. A restart probability of 100% means that the access sequence will always change during restart, for all restarts where a committed update by another client has been performed for one of the objects in the original access sequence.

When undo caches are used at the clients, most modified pages can be restored and retained on abort. Without undo caches, modified objects must be dropped at abort time, and these objects will cause page fetches if they are re-accessed. Therefore, a perfect restart assumption *aids* restart performance if undo caches are used, but *hurts* restart performance if undo caches are not used.

We *do* use undo caches in our experiments. Since AOCC has a higher abort rate than ACBL, a restart change probability of 0% (perfect restart) favors AOCC, while a change probability of 100% favors ACBL. The default setting (used for most of our experiments) is the "middle ground" setting of 50%.

If a workload has good inter-transaction locality, we expect it to have low fetch costs for both first-run executions and restarts: the client cache should be effective in both cases. If a workload has poor inter-transaction locality, its first-run fetch costs will be high, as the client cache will not be effective across transaction commits. Unlike the good locality case, restart fetch costs for the poor-locality case can vary dramatically with restart change probability. If restarts mostly repeat the accesses performed by the failed execution (if the change probability is low), the client cache is "preloaded" with useful state, and restart fetch costs will be low. However, if restarts mostly access *new* objects (if the change probability is high), restarts will have the same bad cache behavior as first-run executions, and restart fetch costs will be high.

Figure 6-9 shows the impact of varying the restart change probability for two different workloads, SMALL+HOTCOLD and UNIFORM. These workloads are used to demonstrate high locality (SMALL+HOTCOLD) and low locality (UNIFORM) results. In both cases the CURRENT system settings were used, with 12 clients and a net object write probability of 20% (with average write clustering). Since ACBL has a low abort rate, its throughput shows little change for both workloads. For SMALL+HOTCOLD (left graph), AOCC shows only a small change in throughput as the restart change probability is varied from 0% to 100%, and there is only a small change in percent improvement (from 25.1% to 21.2%). For UNIFORM (right graph), AOCC shows a larger change in throughput, and there is a larger change in percent improvement (from 20.7% to 5.6%).

As the change probability increases, AOCC's fetches per commit increases, due to more restart-incurred fetches. For SMALL+HOTCOLD, 12 clients (the case shown in Figure 6-9), AOCC's fetches per commit rises from 4.2 to 4.5 as we move from 0% to 100% change probability, a change of 0.3 fetches. For UNIFORM, AOCC's fetches per commit rises from 17.9 to 21.0 as we move from 0% to 100% change probability, a change of 3.1 fetches. Note that UNIFORM not only has a larger increase in fetches per commit, its *per-fetch* costs are high relative to SMALL+HOTCOLD's per-fetch costs, due to more contention over the server disks. *E.g.*, for the 12 client AOCC case, SMALL+HOTCOLD disk utilization is less than 65%, while UNIFORM's disk utilization is greater than 83% (across all change probabilities).

The largest impact of a high restart change probability occurs when the server disks are (nearly) saturated, since fetch costs "blow up" when this occurs. Thus the impact of change probability increases with the number of clients: a higher number of clients results in additional disk load per client and also in an increase in fetches per commit generated by each client. (As we saw in Chapter 5, fetches per commit climbs with number of clients, for both AOCC and ACBL: a higher number of clients means a more rapid removal of state from each client's cache, due to committed updates by other clients, causing client caches to become less effective. For AOCC, a higher number of clients also means a higher abort rate and thus additional fetches due to restarts.)

Combining our observations about client numbers and inter-transaction locality, our general results are as follows. For workloads with good inter-transaction locality, restart behavior normally has little impact, while for high client cases a high restart change probability does narrow the performance gap, but the impact is small relative to the changes that occur with poor-locality workloads. For workloads with poor inter-transaction locality, increasing the restart change probability results in a decrease in the performance gap. The impact of an increased change probability is small for small client cases and grows larger as we move to higher client cases.

Overall, for the CURRENT system settings, we did not see a case where a high restart change probability causes ACBL to outperform AOCC. Section 6.3 shows that a high restart change probability can be combined with specific changes to our default system settings to produce a scenario where ACBL is the better performer.

6.2.4 Transaction Length

The minimum and maximum transaction length parameters are used to set the range of transaction lengths produced by the workload generator: lengths are chosen uniformly from the specified range. For our default setup, TINY+PRIVATE uses the range 90–110, PRIVATE uses the range 140–180, and the remaining four workloads use the range 180–220; this produces average lengths of 100, 160, and 200 object accesses, respectively. We performed two sensitivity analysis experiments to study the impact of transaction sizes: one that varies the average transaction *length*, and one that varies the size of the *range* used for selecting transaction lengths (while holding the average length constant). We chose SMALL+HOTCOLD for these experiments because we consider it to be the best representative of a "real" OODB workload.

The transaction length experiment uses the SMALL+HOTCOLD workload with average lengths between 100 and 400 accesses. In each case, the minimum and maximum lengths are 10% below and above the average. (*E.g.*, for average length 100, the range used is 90–110.) Contention varies with the square of transaction length. For example, moving from length 100 to length 400 causes a sixteen-fold increase in contention level. Large changes in contention level would dominate any other effects of changes in transaction length, *i.e.*, those changes we want to examine in this experiment. To maintain a constant contention level, we take advantage of the fact that contention varies inversely with database size: for each transaction length we use a different database size, such that contention levels are roughly the same in all cases. The CURRENT system parameters and a 10% net object write probability (average write clustering) are used for this experiment.

Figure 6-10 gives the 12-client throughput results for this experiment. As one would expect, throughput (transactions per second) drops for both schemes with longer transactions. An increasing transaction length does cause the performance gap to shrink, but not rapidly: moving from 100 to 400 average accesses, percent improvement changes from 27.7% to 18.8%.

Since the first experiment uses transaction lengths that are all within 10% of average, it does not test the case where a workload uses transactions with a wide range of different lengths. To examine this case, the second transaction size experiment holds average transaction length constant, at 200 accesses, and uses different widths for the transaction range (different pairs of minimum and maximum lengths) to vary the mix of transaction sizes. The smallest width used is 100 (range 150–250) while the largest width is 360 (range 20–380). The CURRENT system parameters and a 20%

net object write probability (average write clustering) were used for this experiment.

Figure 6-11 gives the 12-client throughput results for this experiment. Note that the average transaction length has not changed, thus we do not expect average system throughput to drop rapidly, as in the first experiment. We do find, however, that increasing the range of transaction lengths causes system throughput to drop slightly, as shown in the Figure 6-11 throughput graph. A wider variance in lengths has slightly more impact on ACBL than on AOCC: moving from width 100 to width 360, the percent improvement changes from 22% to 28%.

For a locking scheme, short transactions can block on long transactions; including longer lengths in the transaction mix potentially hurts all transactions, and it is not surprising to find a decrease in ACBL throughput. For an optimistic scheme, as the range in transaction lengths widens, more short transactions can execute during the timespan of a long transaction, and it therefore becomes more likely that transactions with longer lengths will abort and restart. Therefore, it is also not surprising to find a decrease in AOCC throughput.

One might assume that longer transactions will abort several times before they manage to commit, due to their length. However, a restart execution has lower fetch costs (normally) compared to a first-run execution, and therefore executes faster. In other words, "longer" transactions turn into "shorter" transactions.

On Starvation

Finally, we note that with a wide variance in transaction lengths, *starvation* is possible for an optimistic scheme. A starving transaction repeatedly attempts to commit and fails, due to frequent repetitive short update transactions that always manage to commit a new update that conflicts with the (longer) starving transaction. This case does not occur for the above experiments because the workload that is used does not ensure that there is always a short transaction running (which interferes with some longer transaction). Repetitive aborts can occur, but this is a transient phenomenon.

While we expect starvation to be rare, we note that it is possible to add a mechanism to an optimistic scheme to detect and "rescue" a starving transaction. This idea is discussed in the future work section (Section 7.2).

6.2.5 Compute-Intensive Applications

The *read think* and *write think* parameters are client CPU charges that model local client computation per read or write access respectively. These parameters are meant to model the work of the application itself, and not of the underlying OODB. (Other charges, such as the 300 instruction cache lookup charge, are meant to model the system-level costs of an object access.) As a default, we use 5000 instructions per read access and 10,000 instructions per write access (50 and 100 instructions/byte, respectively).

AOCC incurs more aborts than ACBL, and performs more object accesses per commit. Therefore, an increase in the read and write think times would appear to "hurt" AOCC more than ACBL. However, consider that a single block event will cause one ACBL client to wait for another client to complete its current transaction. Thus, on average, an ACBL transaction performs its own object accesses and also waits for some other-client object accesses to be performed. As contention rises, ACBL transactions wait on more accesses performed by other transactions, whereas AOCC transactions re-execute more of their own accesses (but never wait on the accesses of other transactions). As a result, both schemes are "hurt" by increasing the think times.

This section presents experiments that examine the impact of changing the read and write think charges. We chose SMALL+HOTCOLD for these experiments because, as mentioned above, it is the best representative of a "real" OODB workload.

Figure 6-12 gives SMALL+HOTCOLD throughput results for read think charges of 50 and 500 instructions per byte (5000 and 50,000 instructions per object access, respectively). For each experiment, the write think time is set to twice the read think time. The CURRENT system parameters and a 20% net object write probability (with average write clustering) were used for these experiments.

As expected, both schemes are hurt by the ten-fold increase in think times; throughput drops significantly for both schemes. (Note that the height of the Y-axis differs for the two graphs in Figure 6-12.)

As the time spent performing object accesses (or waiting on object accesses performed by other clients) becomes a larger fraction of total transaction latency, the main advantage of using AOCC (messaging cost savings) becomes a smaller fraction of total latency. In addition, server CPU utilization drops as think times increase; delays due to contention over this server resource become smaller. The result is that the performance gap shrinks with increasing think times, but the relative performance of the two schemes does not change. Percent improvement for the 50 instruction/byte case ranges from 9% to 43%, while percent improvement for the 500 instruction/byte case ranges from 1% to 23%.

For the same SMALL+HOTCOLD experiment, if think times are decreased, the performance gap increases. *E.g.*, if read and write think times are changed to 5 and 10 instructions per byte, respectively, percent improvement ranges from 22% to 59%.

In general, a change in think times results in an inverse change in the size of the performance gap, while the relative positioning of the two schemes does not change. There is an exception to this general rule; we discuss this case in Section 6.3.

6.3 ACBL Wins Region

Section 6.2.2 shows that ACBL can outperform AOCC when the large majority of transactions are read-only and there is very little disk traffic per commit. This section examines the question of what conditions are required for ACBL to outperform AOCC when most transactions are read-write transactions.

From the workload model experiments, we know that a high restart change probability can have a large negative impact on AOCC, but only when the workload has low locality of reference. Informally, we can say that restarts must "behave badly" with respect to the current client cache contents. Restarts must perform new accesses that are not repeats of the accesses of the previous (failed) execution. In addition, these new accesses must not be accesses to objects that were recently used by earlier transactions. (The latter requirement rules out the use of workloads with high locality of reference.)

For restart costs to matter, AOCC's abort rate must be high, therefore contention must be high. However, we know that ACBL can suffer from "too much blocking" at very high contention. Thus, contention should not be as high as the contention produced by, for example, the HICON workload.

The UNIFORM workload has low locality of reference and reasonably high contention (but not as high as HICON's contention). It meets the requirements above, and is a good "killer workload" for AOCC. However, the UNIFORM restart change probability experiment in Section 6.2.3 did not show ACBL could outperform AOCC. (It did show a large drop in AOCC's performance with increased change probability: AOCC's throughput drops down to ACBL's throughput, but there is no crossover.) This suggests it is also necessary for the system setup to favor ACBL.

From the system model experiments, we know that the following changes from our default system setup should "help" ACBL more than AOCC: lower CPU charges for message send/receive, a faster server CPU, lower disk bandwidth, and a smaller server cache. The latter two changes have the same impact: the disks become more utilized with either a lower bandwidth or a smaller server cache.

Using these insights, we define a new set of system parameter settings that favor ACBL; we refer to these settings as the "ACBL Wins" settings. Starting with the CURRENT settings, the following changes are made:

1. The server CPU speed was doubled (from 50 to 100 MIPS).

2. The fixed and variable CPU charges for sending and receiving messages were lowered to 3000 fixed instructions and 2048 instructions per KB (*i.e.*, we shifted to the FUTURE settings for these two parameters).

3. The server cache size was reduced from 50% to 10% (of the working set size).

Note that we chose to reduce server cache size but not to reduce disk bandwidth; as discussed above, either change makes the disks more heavily utilized.

To further "help" ACBL, we use a high quality of write clustering when experimenting with the ACBL Wins system: a net object write probability of 10% is used, where cluster write probability is 20% and object write probability is 50%. (ACBL only needs to use write locks for one fifth of all pages that are accessed.)

Figure 6-13 gives throughput and percent improvement graphs for an experiment that uses this ACBL Wins system with the UNIFORM workload and a 100% restart change probability. Note that a negative percent improvement result indicates a case where ACBL outperforms AOCC, and is drawn as a bar dropping below the X axis on the percent improvement graph.

The percent improvement graph in Figure 6-13 shows us that AOCC slightly outperforms ACBL for small client cases (by less than 1%); for the 8–24 client range, ACBL outperforms AOCC by up to 6.5%. (The peak vs. peak percent improvement is in ACBL's favor by 6.0%.)

It is important to note that the 8 client point is the point where ACBL begins to saturate the server disks; ACBL does not outperform AOCC prior to this point. By 8 clients, AOCC's disk utilization has reached 80%. We believe that this saturation or *disk thrashing* condition is required for ACBL to outperform AOCC.

The need for high disk contention is explained as follows. ACBL has high blocking costs under high contention. Even though an ACBL transaction performs fewer disk operations per commit, it also blocks on other transactions, and thus waits on the disk operations of other transactions. Disk waiting costs can therefore be equal (or in AOCC's favor) even if each AOCC commit uses more disk operations than each ACBL commit. Once AOCC begins to saturate the disks, however, its disk waiting times rise sharply (while ACBL's disk waiting times do not).

In summary, the "ACBL Wins" region is bounded by the following three conditions:

1. Contention must be high, but not so high that ACBL performs "too much blocking."

2. The accesses performed during restarts must cause enough fetches for AOCC's fetch count to be significantly worse than ACBL's fetch count (per commit). For this to occur, the overlap between restart accesses and the accesses of the previous failed transaction must be low and the workload as a whole must have poor inter-transaction locality.

3. It must be the case that the locking scheme does not drive the server CPU towards saturation faster than the optimistic scheme drives the server disks towards saturation. In addition, the optimistic scheme must drive the server disks near saturation; disk resource contention at the server must be high.

We performed some additional experiments to test these conditions. Starting with the "ACBL Wins" system, UNIFORM workload, and 100% change probability, as used in the above experiment, we made one modification per experiment:

- If "perfect restarts" are used (0% restart change probability), ACBL does not outperform AOCC.

- If we use SMALL+HOTCOLD rather than UNIFORM, ACBL does not outperform AOCC.

- The ACBL Wins system is essentially a partial switch from CURRENT to FUTURE system settings, where only the changes that favor ACBL were made. If we also use the FUTURE settings for disk, network bandwidth, and client CPU, ACBL does not outperform AOCC. (This additional experimental setup differs from the default FUTURE setup in only two ways: change probability is set to 100%, and the server cache is set to 10%.)

- If the read and write think times are set to 500 and 1,000 instructions per byte, the server resources do not saturate in the 1–24 client region, and ACBL does not outperform AOCC.

  (This shows there is an exception to the think time results given in Section 6.2.5. For a case where ACBL outperforms AOCC, a sufficiently large increase in think times moves the system "out of" the ACBL Wins region, and AOCC then outperforms ACBL.)

These experiments confirm our characterization of the three conditions that define the "ACBL Wins" region. This region has bounded scope, and is unlikely to occur in practice. Conventional wisdom is that OODB applications do not have high contention, thus the first condition is not expected. Even for high contention workloads, we expect that restarts normally *do* overlap significantly with previous failed accesses. We also do not expect most OODB systems are configured in a "disk poor" manner such that the disks approach saturation at a client point that is much lower than the desired number of users of the system.

6.4 Summary

This chapter explores how the relative performance of ACBL vs. AOCC changes as we vary different system and workload parameter settings. For a wide range of settings, the performance gap (AOCC's percent improvement over ACBL) can narrow or widen, but AOCC remains the better scheme. Thus, while Chapter 5 gives results and analysis for the CURRENT system and default workload settings, the same relative results and the same insights hold across a large region of the overall workload and parameter space.

In addition to demonstrating the robustness of our results, the experiments described in this chapter provide new insights. The important new results are as follows:

- AOCC is more robust with respect to contention level, update frequency, and the quality of write clustering. The performance gap widens with an increase in either the number of object writes per commit or in the number of page updates per commit; the gap also increases when contention is increased by adding clients to the system.

- Sensitivity analysis experiments for the core system parameters (server, client, network, and disk) show that changes that reduce disk load relative to server CPU load cause the performance gap to widen, while changes that reduce server CPU load relative to disk load cause the performance gap to narrow. Thus, the gap widens with faster disks, faster client CPUs, larger client caches, and a larger server cache. The gap narrows with a faster server CPU and lower CPU charges for message send/receive.

- Changing all system parameters from CURRENT to FUTURE settings has a different impact for low and high contention cases. For low contention, the decrease in message costs is the most significant change; relative performance is based on message costs, thus the performance gap narrows. For high contention, the increase in available system resources is the most significant change. AOCC is able to make good use of the additional resources, while ACBL is not (due to its high blocking costs). The result is a wider performance gap. In addition, both schemes exhibit a shift in the location of their peak throughput (to a higher client point), where AOCC's shift is more dramatic.

- If transaction restarts mainly repeat previous (failed) accesses, this favors AOCC; if restarts mainly perform new accesses, this hurts AOCC. Restart change probability has little impact on performance gap if the workload has good inter-transaction locality, while the performance gap narrows if the workload has poor inter-transaction locality. The impact of a high change probability increases with increased server disk utilization.

- Increasing the average transaction length causes a small narrowing of the performance gap. Increasing the range of transaction sizes used by the workload causes a small increase in the performance gap.

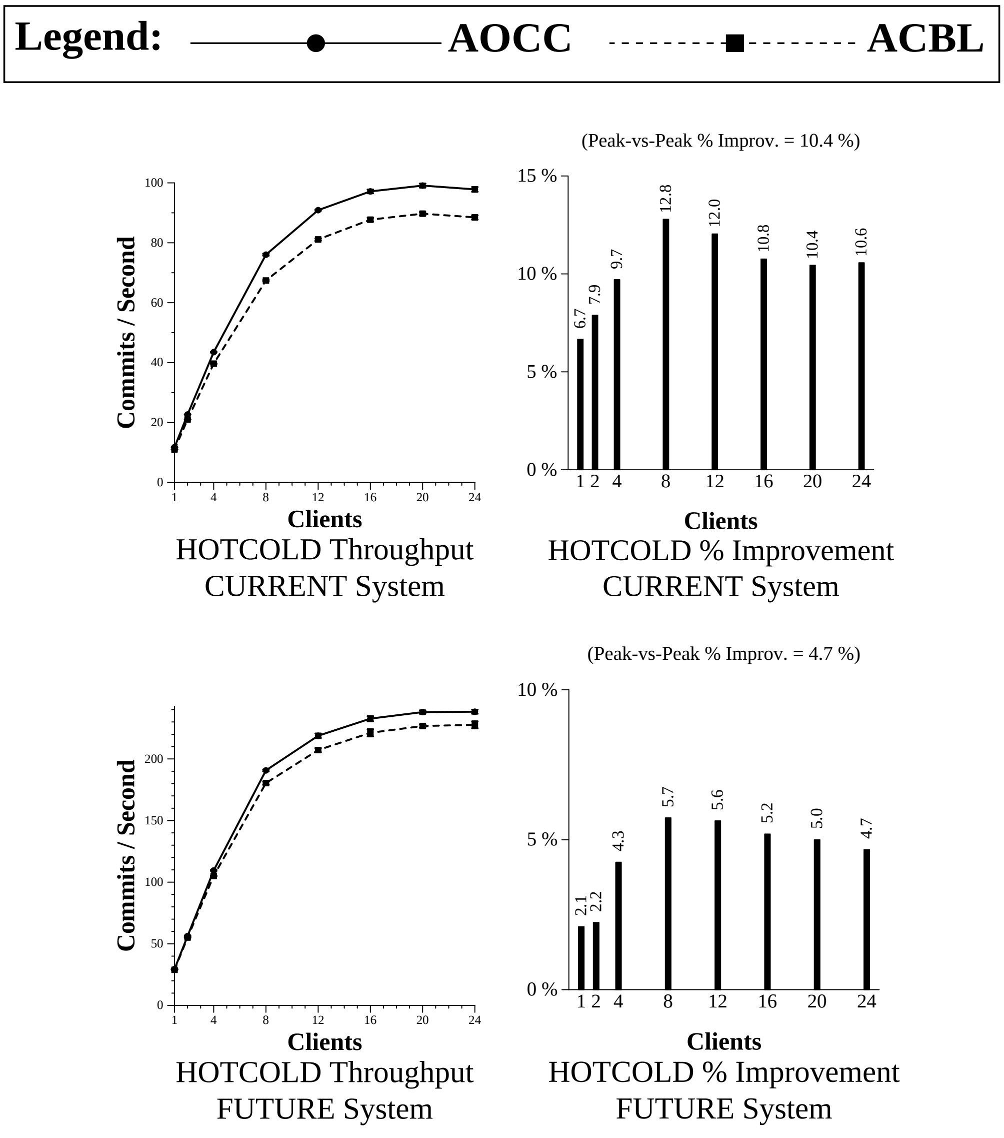
- Increasing the client computation time (the think time) per access narrows the performance gap, while decreasing the time widens the gap.

Our experiments found two cases where ACBL can outperform AOCC. First, if the workload mix mostly consists of read-only transactions *and* transaction latencies are small relative to round-trip latencies, ACBL can significantly outperform AOCC. This case requires very few disk operations per commit; almost all accesses must hit in the client or server cache. In addition, this case only applies to a multi-server system. While we simulated an AOCC read-only commit request to approximate this multi-server case, a more realistic study of multi-server performance is needed; this is an area of future work.

Second, for read-write transactions, we identified an "ACBL Wins" region of the overall system and workload parameter space. This region is bounded in scope. There are three necessary conditions. There must be high contention. There must be a restart access pattern that does not match the "preloaded" client cache contents (accesses of the failed transaction are mostly not repeated, and the workload in general has low locality of reference). AOCC must saturate the server disks faster than ACBL saturates the server CPU, and moreover AOCC must actually start to saturate the disks before ACBL can significantly outperform AOCC.

It is interesting to note that callback locking schemes are designed to perform well for workloads with low contention and good inter-transaction locality. Such cases clearly fall outside the "ACBL Wins" region. Thus, AOCC outperforms ACBL on the workloads that motivate the use of callback locking.

Our general conclusion is that optimism is robust for the workloads and systems covered by the scope of this thesis: when the average transaction size is smaller than the client cache size, our optimistic scheme exhibits excellent performance across different sharing patterns, across good and bad write clustering, across low and high contention, and across low and high locality of reference.



CURRENT System, Default Workload Settings

Figure 6-4. CURRENT vs. FUTURE: Low-Contention Example (HOTCOLD)

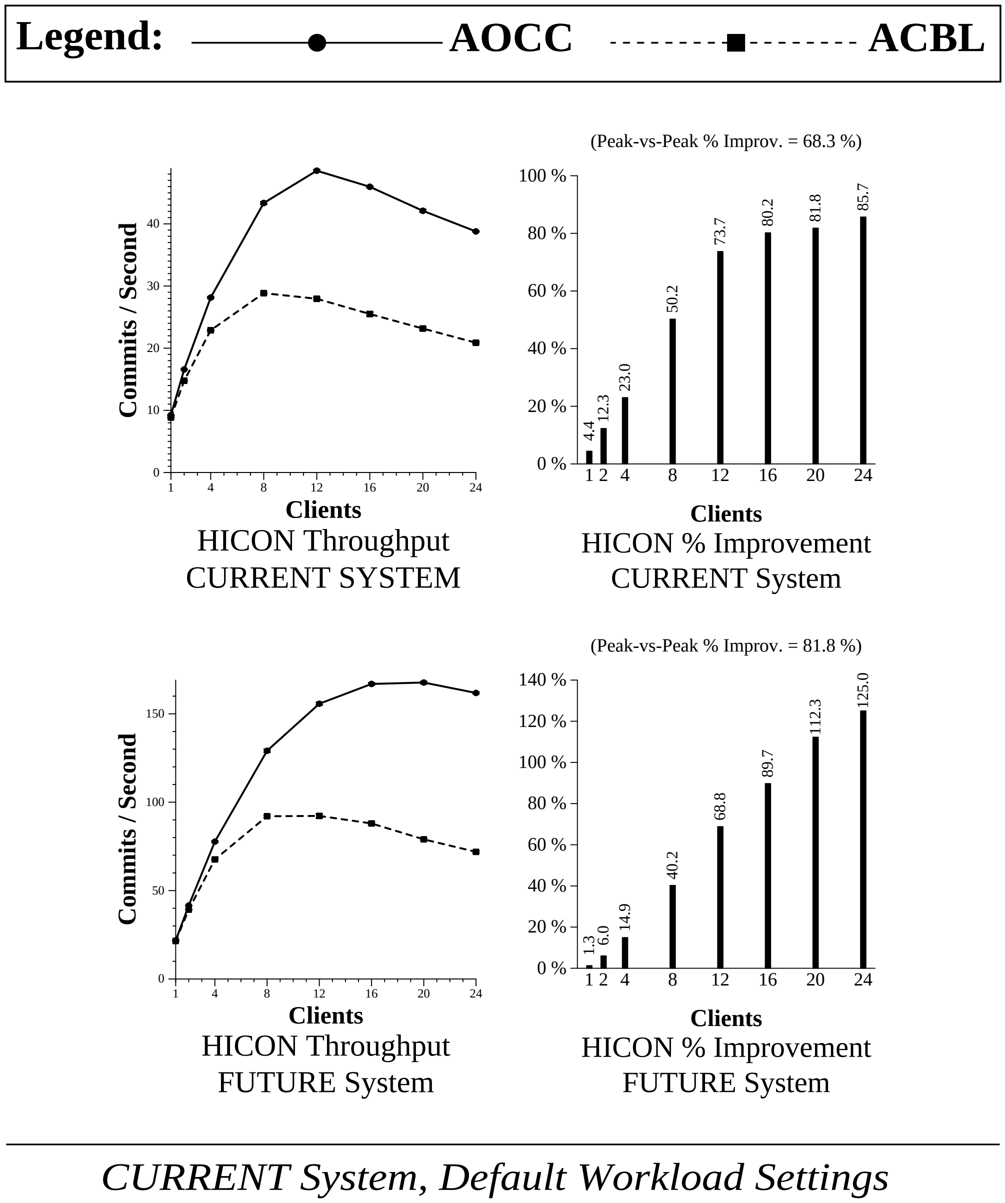


Figure 6-5. CURRENT vs. FUTURE: High-Contention Example (HICON)

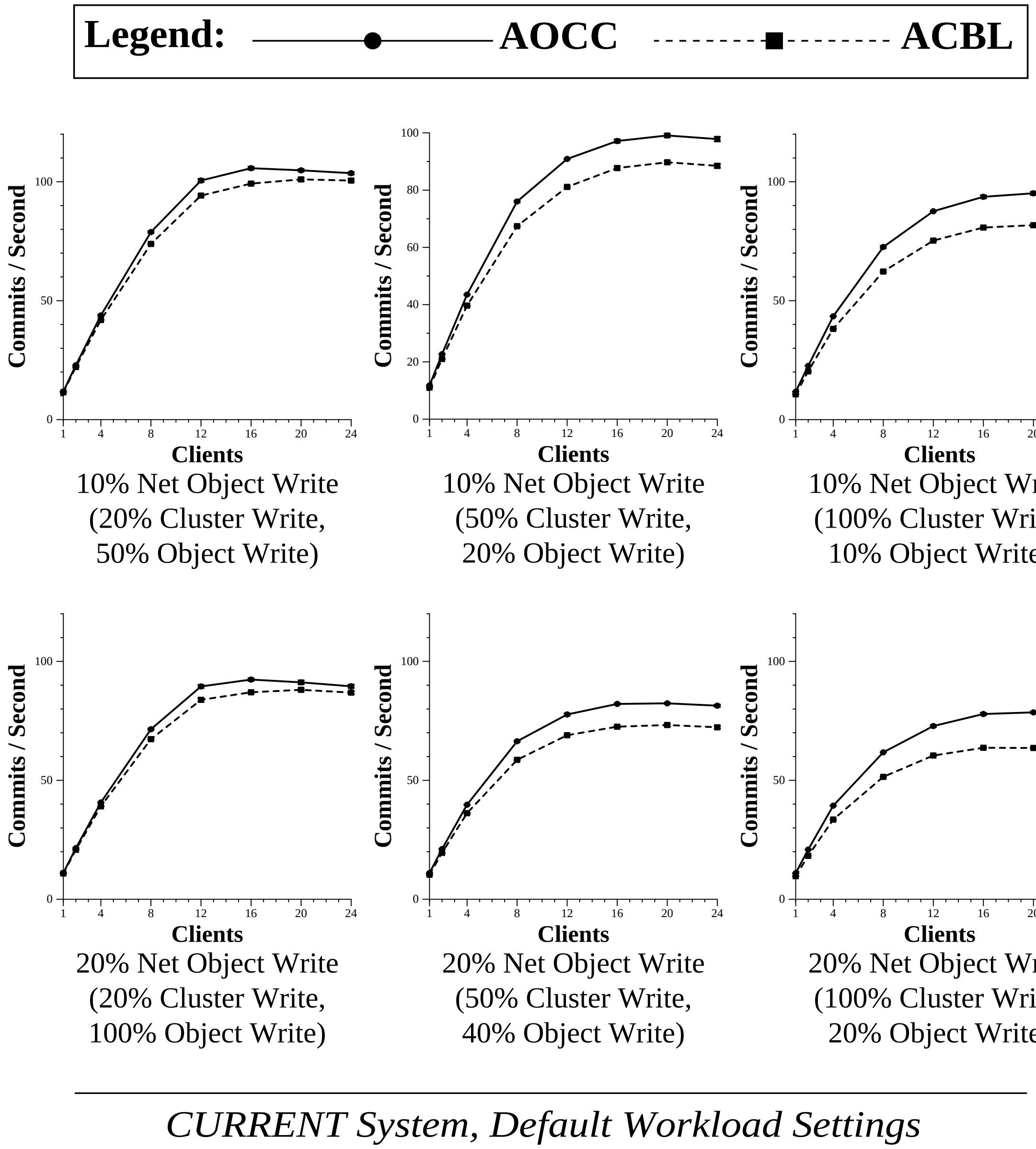
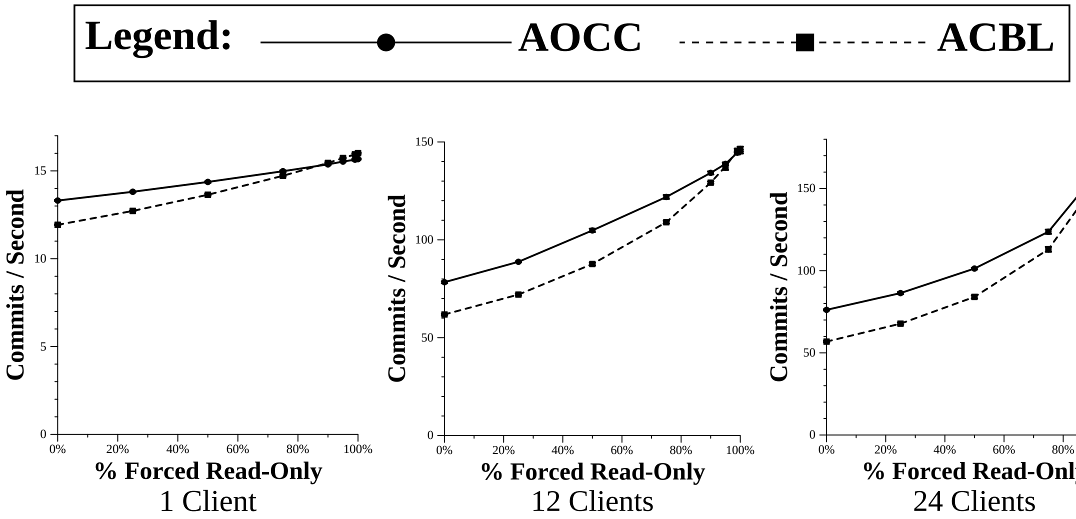
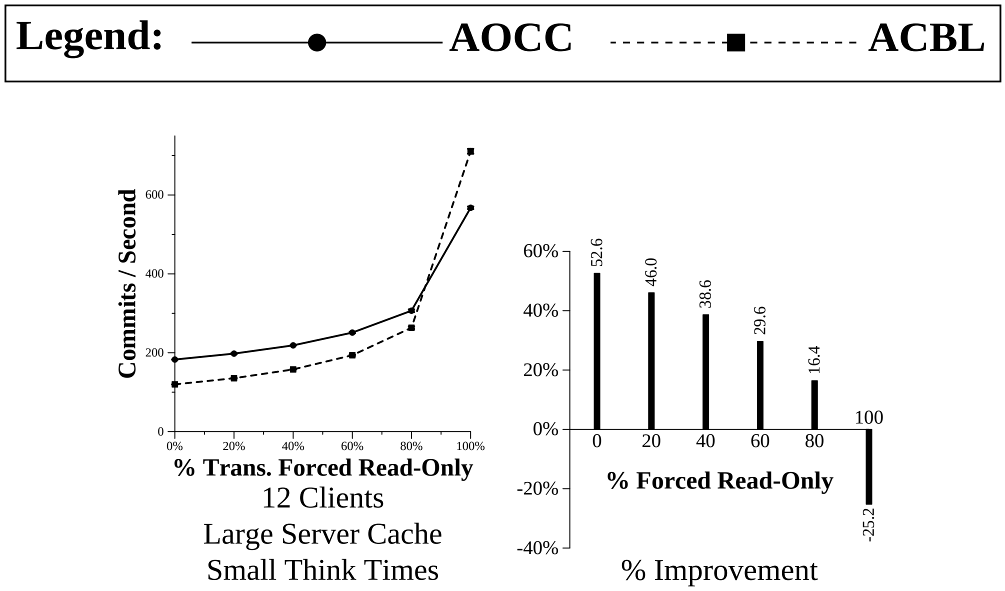


Figure 6-6. Varying Write Probability, Write Clustering (HOTCOLD)



CURRENT System, Default Workload Settings

Figure 6-7. SMALL+HOTCOLD: Read-Only Mix Experiment



CURRENT System, Default Workload Settings

Figure 6-8. SMALL+HOTCOLD: Main Memory Server

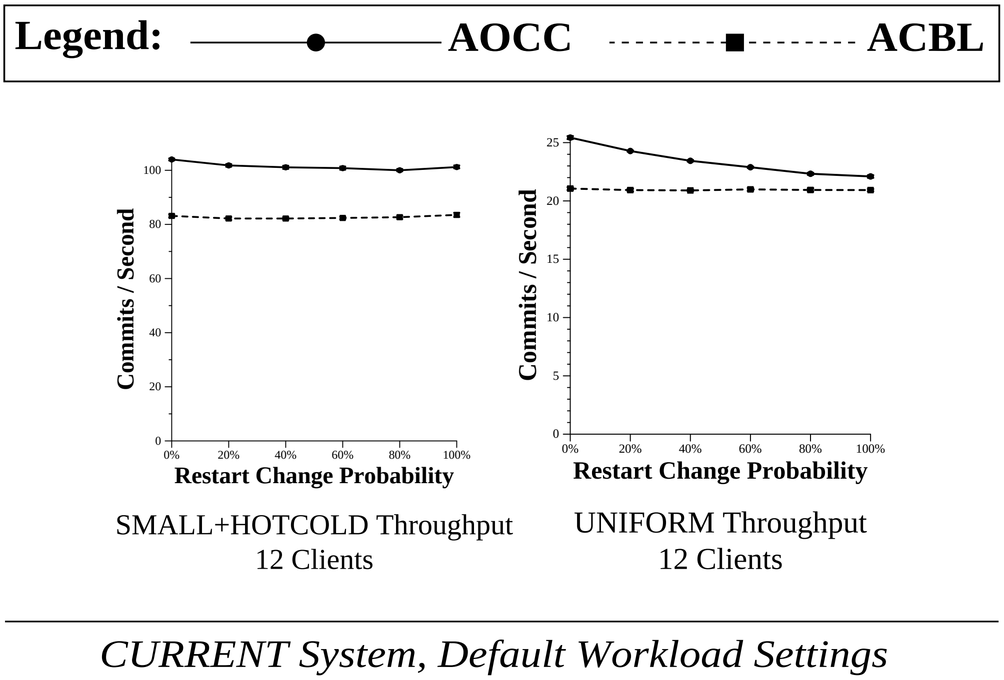


Figure 6-9. Varying the Restart Change Probability

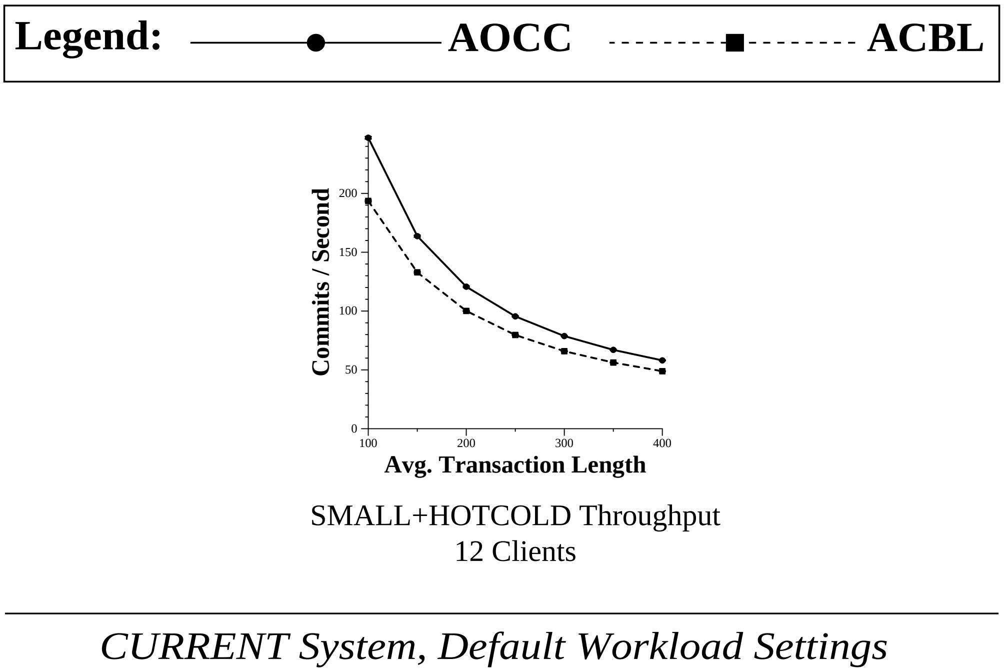
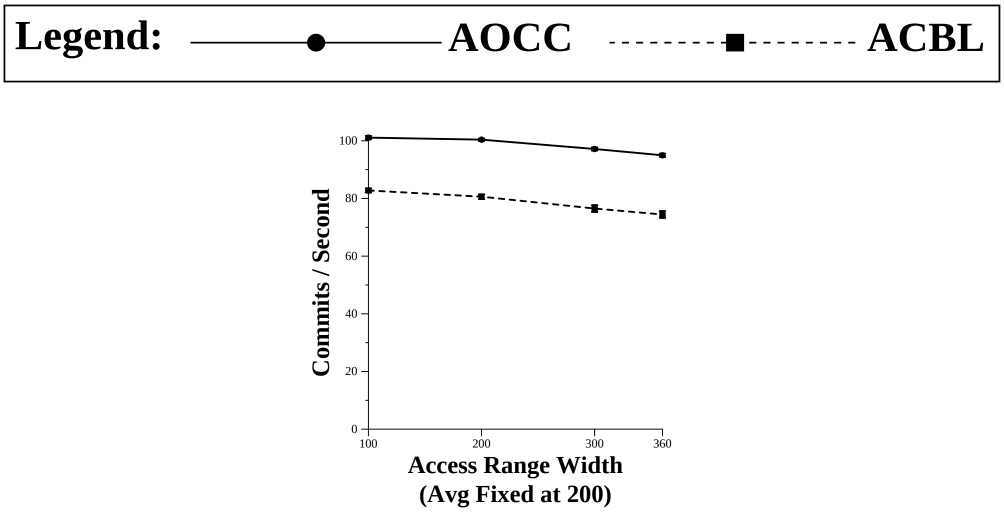


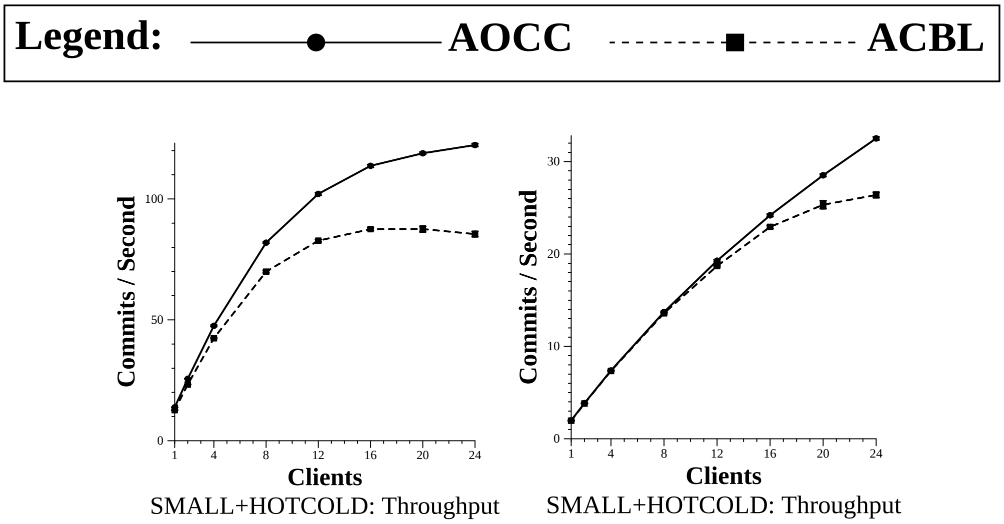
Figure 6-10. Varying the Average Transaction Length



SMALL+HOTCOLD Throughput
12 Clients

CURRENT System, Default Workload Settings

Figure 6-11. Varying the Range of Transaction Sizes



SMALL+HOTCOLD: Throughput
Read Think Time = 50

SMALL+HOTCOLD: Throughput
Read Think Time = 500

CURRENT System, Default Workload Settings

Figure 6-12. SMALL+HOTCOLD: Think Time Experiment

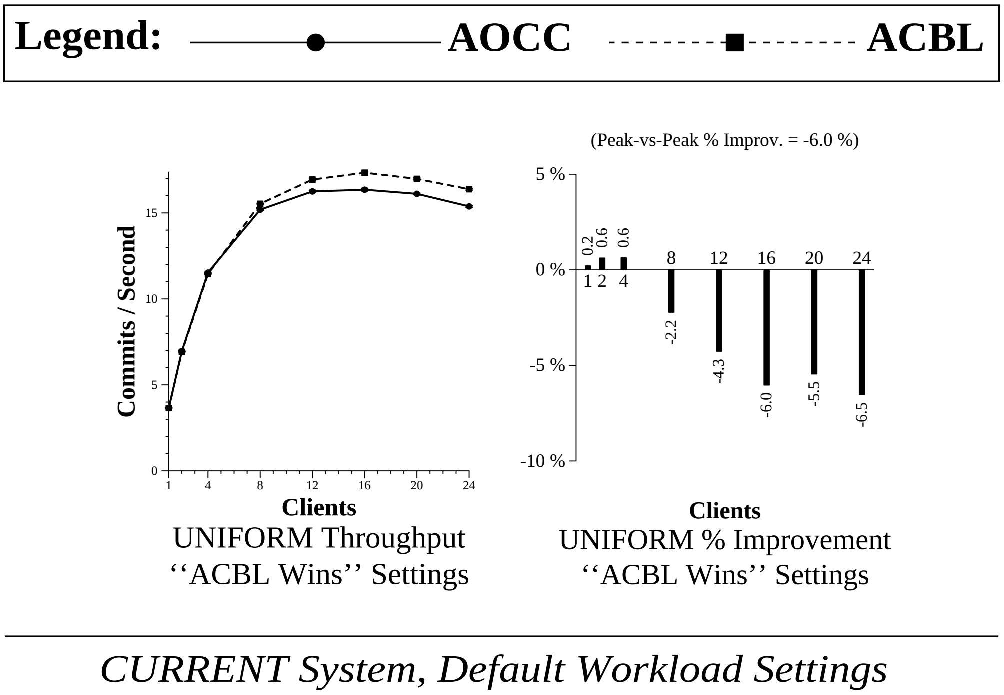


Figure 6-13. UNIFORM: ACBL Wins Region

Chapter 7

Conclusions

7.1 Results and Insights

In a client-server system, the goal of moving both data and computation to clients is to produce a system that has high throughput, low response times, and good scaling properties. Moreover, for workloads that have inter-transaction locality of reference, retaining cached state at clients across transaction boundaries should provide additional benefits. The choice of concurrency control scheme has a large impact on whether these performance and scalability goals are met.

In this thesis we study two schemes that both attempt to minimize the required synchronization steps between client and server. ACBL, a callback locking design, can avoid read lock requests for cached data if these locks are cached across transaction boundaries. This approach works reasonably well when updates to shared objects are infrequent and when accesses to read-write shared pages are infrequent. However, we show that the performance of this scheme is sensitive to many different variables: the number of clients using a shared data set, the frequency of update, the degree of object-level contention (real data conflicts), the degree of page-level contention (false sharing), and the likelihood that objects modified by a single transaction are clustered onto a small number of pages (write clustering). All of these variables can have a large impact on ACBL's client-server synchronization costs.

AOCC, our new optimistic scheme, has low synchronization overhead for the low-contention case; it outperforms ACBL by a small but significant margin on the workloads that ACBL is designed to perform well on. This result supports conventional wisdom, which says that optimism can outperform locking when the abort rate is low. We *also* show that AOCC is more robust with respect to all of the variables listed above: as the number of clients increases, as the frequency of updates increases, as real data conflicts or false sharing increases, and as the quality of write clustering decreases, the performance gap between the two schemes widens.

Relative to ACBL, AOCC's abort rate rises much faster with contention level. However, AOCC is the better scheme even when it has a high abort rate. In fact, AOCC outperforms ACBL by a wider margin under high contention than under low contention. This result is the opposite of conventional wisdom, which equates a high abort rate with low performance. Our results are different because AOCC uses mechanisms designed to improve restart performance, while previously studied optimistic (or semi-optimistic) designs do not.

The primary AOCC design goal was to keep client-server interaction to a minimum. However, since optimistic schemes *do* incur high abort rates, reducing the number of interactions due to aborts and restarts was considered an integral part of this goal. Thus, AOCC's design includes mechanisms for (1) avoiding aborts due to out-of-date accesses; (2) detecting the need to abort early to avoid an unnecessary commit request; (3) restoring and updating the state of a client cache on abort to avoid fetch requests during restarts. This combination of mechanisms works very well: AOCC's round-trip synchronization costs do not rise rapidly with a rising abort rate. Note that early abort detection combined with low fetch costs during restart "protects" the server from high restart costs: much of the cost of a restart is paid at the client that executed the failed transaction. Therefore, restarts don't have a large impact on the server's ability to scale to many clients.

One of the fundamental assumptions that defines the scope of our study is that client caches are large enough to retain the objects accessed by a transaction. AOCC's low restart costs do not necessarily apply outside of this scope. However, we believe that our results have wide applicability: the client cache assumption holds for many present-day systems and applications, and will encompass more cases in the future, due to improved cache designs and increasing cache sizes. Main memory at clients will continue to increase. Furthermore, hybrid cache designs [36, 47] retain both useful pages and useful objects, resulting in more effective use of cache space. In addition, cooperative caching designs allow fetches to be serviced by other clients [15, 22, 25], while main-memory databases (*e.g.*, DaliJagadish94) are becoming more common; it is increasingly likely that additional fetches due to aborts will be serviced from client or server memory rather than server disk.

In addition to its good performance properties, AOCC is the better scheme with respect to simplicity, scalability, and robustness:

Simplicity. AOCC is much simpler to implement than an adaptive-granularity callback locking scheme such as ACBL. The difference in the complexity of the two designs is easily seen in Chapter 2. A locking scheme *must* use multiple locking granularities to reduce locking overhead while still achieving high concurrency. In contrast, AOCC obtains high levels of concurrency without the need to dynamically switch its conflict-detection granularity. Unlike a locking scheme, an optimistic scheme can perform all conflict detection at object-level without incurring high message costs.

Scalability. In general, a locking scheme has high message overhead compared to AOCC. For a callback locking scheme such as ACBL, message costs grow with the number of clients in the system that are sharing data. Thus, with respect to message costs, AOCC scales better with number of clients.

In addition, we believe AOCC scales better than ACBL with respect to the distance between client and server. Note that we simulated a local-area network (LAN), but not a wide-area network (WAN). AOCC's low use of synchronous round-trip exchanges makes it less sensitive than a locking scheme to high network latencies, and we believe AOCC's advantages over a locking scheme will increase as one moves from a LAN to a WAN setting.

Robustness. AOCC transactions executing at different clients have minimal interaction; synchronization between transactions occurs at the server, and only at the commit point of a transaction. Under a callback locking scheme both the use of blocking and the use of callbacks makes it likely that the performance of a transaction running at client X will depend on the performance properties of another client Y. Thus, we believe AOCC is more robust with respect to a high variance in the availability or performance properties of clients.

Confirming AOCC's advantages for a WAN system and for a system with a high variance in client properties remains an area of future work. The remainder of this chapter discusses some additional future work. To keep this discussion short, we have restricted our discussion to two issues: improving the design of our optimistic scheme, and improving the simulation testbed.

7.2 Future Work

7.2.1 Starving Transactions

A transaction is *starving* if it repeatedly fails to validate and commit, due to frequent repetitive updates performed by other transactions that do manage to commit. This scenario is rare and often transient. Starvation can end without the intervention of a special mechanism if the interfering clients move on to other work.

Starvation is a well-known *potential* problem for optimistic schemes, and a number of solutions have been proposed. The original Kung/Robinson paper on optimistic methods discusses detecting a starving transaction by placing a limit on the number of successive aborts. The system can then rescue this transaction by holding a semaphore during its next restart execution that effectively locks the entire database, ensuring that the restart will succeed [38]. Thomasian and Rahm proposed a page-level variant of this idea: on abort, page-level locks can be acquired for the pages accessed by failed transaction T, ensuring that T's restart will succeed as long T does not access any *new* pages during restart [48]. The Thomasian/Rahm scheme uses locking on the very first abort; it attempts to avoid all multiple-restart scenarios.

A possible area of future work is to use simulation to investigate the effectiveness of different starvation solutions. A solution requires two components, detection and "rescue." For detection, one could study different heuristics used to trigger a rescue mission. While the solutions described above use an abort count (of either 1 or N), other criteria could be applied, such as the age of a transaction.

For rescue, two questions arise. First, for a starving transaction T, how does the system identify the state that should be protected from update during T's restart execution. T's current read set is a good starting point; perhaps the system can identify other objects that are likely to be accessed during restart. Second, what level of granularity should be used to lock this state? If restarts are "perfect" then object-level locking can be used. If restarts perform some new accesses but these accesses still use the pages accessed by the previous execution, page-level locking will successfully protect the restart from abort. If restarts are less predictable, it may be necessary to lock units larger than pages.

7.2.2 Read-Only Transactions

Allowing a read-only transaction to choose a serialization point somewhere "in the past" with respect to the current committed state of the database is a standard approach to improving read-only commit performance. For example, some multi-version concurrency control schemes (such as the scheme by Agrawal *et. al.* [2]) choose a start point T_{start} at the start of read-only transaction T and then ensure that all of T's accesses are consistent with this time. Thus, T always commits by choosing this point as its serialization point. This guarantee of a commit is a very nice property (especially for an optimistic scheme).

However, multi-version schemes have potentially high space overhead. For the above example, while T is executing, if an update to object O is installed with a commit time later than T_{start}, an "old" version of O that is consistent with T_{start} must be maintained so that if T should happen to read object O, it will read a version of O that is consistent with its chosen start time. If there are many read-only transactions in the system, the management and garbage collection of old versions adds significant overhead to the system.

In a client-server system, the starting state of a client cache where a read-only transaction is about to run can retain "old" versions of recently updated objects simply by *not* updating the cache with new state. This is not a full solution, however, because the cache may not start out with all of the state that will be used by the read-only transaction. Moreover, once the transaction commits, a significant amount of work may be required to bring the cache back up to date with respect to the committed state of the database. Finding a solution to these problems (that does not have the same high overhead as a standard multi-version scheme) is an interesting area of future work.

For single-version systems, it is also possible to allow a read-only transaction to choose a serialization point that is "in the past." As discussed in Section 2.5.3, our optimistic design does not need to use read-only commit requests for a single-server system. At the server, updates are applied in commit order, and transaction-consistent invalidation messages are generated due to these updates. Thus, a client that has processed all invalidations generated through time T_{reply} (the time that the last fetch reply was sent to the client) has a consistent cache with respect the committed state of the system at time T_{reply}. If a read-only transaction is not aborted due to invalidations, it can be "serialized" at this point in time.

Thus, allowing an immediate commit with serialization time T_{reply} improves commit performance. An area of future work is to develop an algorithm that allows such immediate commits for multi-server systems. For such systems, a client can know that the server S1 objects in its cache are consistent with respect to time $T_{S1-reply}$, and the server S2 objects are consistent with respect to time $T_{S2-reply}$. However, this information does not imply global consistency for the client cache.

7.2.3 New Studies

Our simulation environment can be extended to support some of the future work that we have described. One required feature is the ability to model heterogeneous clients. For example, to create a starvation case, one client could run a workload with "longer" transactions while the other clients could run a workload with "shorter" transactions. Similarly, one client could have a slower CPU, or a lower-bandwidth client-server network connection as compared to the other clients in the system.

A real system runs a mix of workloads, where groups of clients are each running a different application. Thus, it would be interesting to examine system behavior for different workload mixes.

Finally, where our current simulator allows us to study steady-state behavior, it would also be interesting to study dynamic behavior. For example, suppose an application uses working set A for time period T1 and then switches to working set B for time period T2. During each time interval there is high inter-transaction locality of reference for the working set in use. At the beginning of time period T2, working set B pages are not cached at the clients; there will be a transition period where B pages are "swapped in." It would be interesting to examine the behavior of different schemes across this transition period.

Appendix A

On System Parameter Settings

The system model and the CURRENT and FUTURE settings are presented in Section 4.1. This appendix discusses the process we used to select the two sets of parameter settings, giving some additional insight into our experimental setup. This additional information is interesting, but is not needed for understanding our simulation results.

A.1 Fixed Settings

Parameter	Setting
Object size	100 bytes
Page size	4 KB
Objects per page	40
Working set size	1250–1300 pages
Server cache size	50% of working set size
Modified object buffer size	50% of working set size
Client cache size	25% of working set size

Used for both CURRENT *and* FUTURE *systems.*

Figure A-1. Size-Based Parameters

We begin by discussing the parameter settings that do *not* change as we move from the CURRENT to the FUTURE system. In particular, we chose to keep the size-based parameters the same, as shown in Figure A-1, which summarizes the settings for all of the size-based parameters.

A small object size was chosen because we expect many small objects to be present in real OODBs. For example, in the OO7 benchmark [8, 7], a widely accepted object-oriented database benchmark, most objects are smaller than 100 bytes. Also, small objects are the most interesting case to study with respect to concurrency control. With a 4 KB page size and 40 objects per page, a scheme like ACBL can lock 40 objects with a single write lock request; this allows us to explore the benefits of using an adaptive locking scheme. We chose a 4 KB page size as a reasonable size for fetch requests, given current network costs.

The parameters that change as we move to the FUTURE system all involve speed rather than size: we wanted to isolate the impact of moving to a faster system, and size changes would have complicated the picture. For example, had we switched to a larger page size for the FUTURE system, there would be more objects per page. This would mean ACBL would need fewer page-level locks, but it would also mean that ACBL would switch to object-level locking more frequently, since there would be more false sharing at the page level. Thus, by fixing the size parameters, it is easier to reason about the changes that occur as we move from CURRENT to FUTURE system.

Parameter	Cost
Cache lookup	300 instr.
Register / Unregister (Lock / Unlock)	300 instr.
Validation time per object	0–300 instr.
Cost of deadlock detection	0 instr.

Used for both CURRENT *and* FUTURE *systems.*

Figure A-2. Misc. Client and Server Costs

Some CPU charges when expressed as a number of instructions do not change as we move from the CURRENT to the FUTURE system. Figure A-2 summarizes the settings for these parameters. (The FUTURE processors are faster, so in this sense these parameters do change; they take less time.) The settings for these parameters are justified in Chapter 4.

Client and Server Settings

Parameter	CURRENT	FUTURE
Server CPU speed	50 MIPS	200 MIPS
Client CPU speed	25 MIPS	100 MIPS

Figure A-3. Client and Server CPU Speeds

Figure A-3 gives the client and server CPU speeds. For the CURRENT system, the client and server processors execute 25 and 50 million instructions per second, respectively. The FUTURE system has processors that are four times as fast. It can be argued that faster speeds make more sense given that faster processors are already available commercially. Chapter 6 has an experiment that varies the client and server CPU speeds and shows the effect of using faster processors.

A.2 Network Settings

Parameter	CURRENT	FUTURE
Network bandwidth	80 Mbps	160 Mbps
Fixed network cost	6000 instr.	3000 instr.
Variable network cost	7168 instr./KB	2048 instr./KB

Figure A-4. Network Parameters

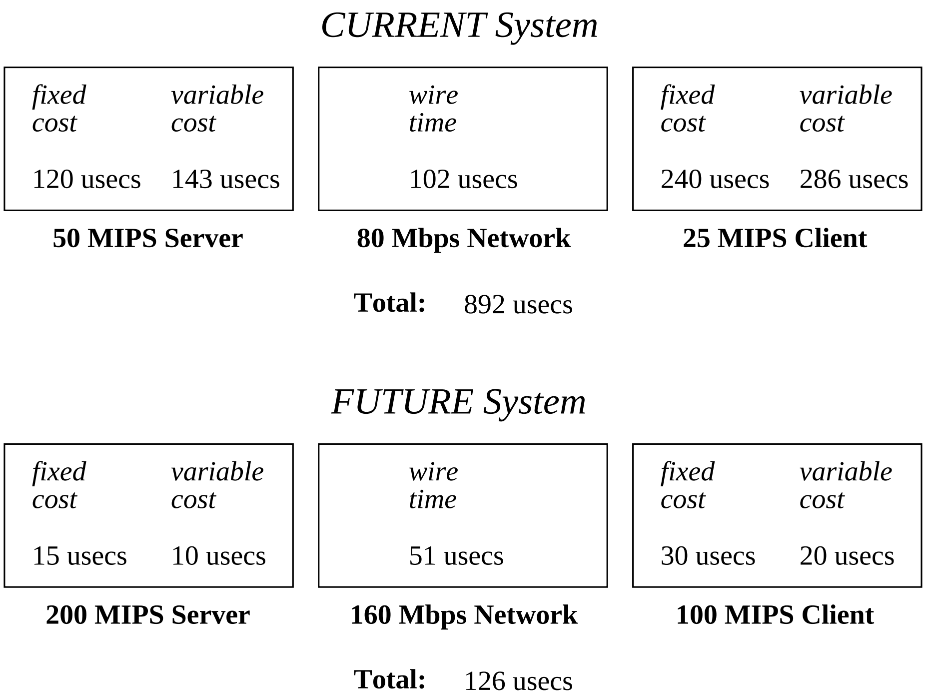


Figure A-5. Latency for 1 KB Message

Figure A-4 summarizes the network parameter settings for the CURRENT and FUTURE systems. Given these settings, and the fact that our FUTURE CPU's are faster than our CURRENT CPU's, latencies are significantly lower in our FUTURE system. This is demonstrated in Figure A-5, which gives a breakdown of the 1-way latency for a 1 KB message, for both the CURRENT and FUTURE systems. Note that in each case the server CPU is twice as fast as the client CPU; while the server and client are charged the same number of fixed and variable instructions, the server executes these instructions in half the time.

Briefly, the reason we use a lower fixed cost for the FUTURE system is that we expect future operating systems will require fewer kernel interactions per message. At the same time, the variable cost will be lower because user-level processes will be given more direct access to the network interface: this will result in less unnecessary copying (it also enables the use of messaging protocols that are tailored to individual applications or system environments). We present some results for an experimental version of such a *user-level messaging* system below.

The remainder of this section discusses how we chose the specific CURRENT and FUTURE settings shown in Figure A-4.

A.2.1 Network Bandwidth

A 10 Mbps Ethernet is probably the most common network in use today. However, when used with our other CURRENT settings, a 10 Mbps bandwidth can cause the network to saturate at 10 clients or less. We want to run experiments with up to 24 clients, and we want the characteristics of the different concurrency control schemes under study, rather than the maximum bandwidth of the network, to dictate the performance results. (Moreover, as explained above, our network model is inaccurate under saturation.) For these reasons, we use a bandwidth of 80 Mbps for our CURRENT system. This bandwidth is available as a "commodity item" today, using an FDDI network (a fiber token-ring). We use a FUTURE bandwidth of 160 Mbps. Such speeds are already available with ATM technology.

A.2.2 CPU costs

Determining reasonable fixed and variable instruction costs for the CURRENT and FUTURE systems turned out to be an interesting challenge. Ultimately, we decided to calculate estimates for these costs by examining some reported latency measurements from actual systems. To produce estimates for the CURRENT system, we examined latency measurements from present-day "monolithic-kernel" operating systems, such as Mach 2.5 and Ultrix 4.2; for the FUTURE system, we examined latency measurements from experimental user-level messaging systems.

For monolithic-kernel operating systems, all message protocol processing runs within the kernel. Every send or receive requires protection domain crossings (context switches). In addition, unnecessary copying costs are incurred if the message must be copied between user space and kernel space; this is required if there is no shared-memory support to give the kernel direct access to user-level data. Network protocols have traditionally been built using a layered approach, where each layer queues a packet on the input queue of the next layer; this approach has the advantage of being modular, but has additional overhead compared to an approach that uses a single end-to-end implementation.

Newer approaches such as the ones described in [5, 20, 56] move most of the network code to user-level. For protection and fairness reasons, kernel interaction is still required to set up user-level communication. Shared memory regions are used to provide the required level of protection. However, once such regions are established, user-level processes can communicate without using kernel calls. As [56] points out, an ideal "zero copy" protocol would be possible if the network interface could "map in" arbitrary memory from the user-level address space (send and receive queues could contain arbitrary descriptors of message contents). Most current I/O architectures do not allow I/O devices to do arbitrary mapping. Thus, a limited amount of shared message buffer memory is allocated to each user-level process, and message data must be copied to/from this space by the user-level process. Even such a 1-copy approach saves copying compared to a monolithic kernel that does not share message buffers with user-level processes.

Von Eicken [56] gives a good summary of the advantages of the user-level approach. For backward compatibility, user-level libraries can implement the protocols commonly available in existing kernels (*e.g.*, UDP and TCP). However, where a kernel implementation of such a protocol is generally tuned to perform well across a range of applications, if the same protocol is provided as a user-level library, it can be tuned on an per-application basis. Moreover, one can implement a new protocol tailored to the specific communication requirements of a particular application.

Machine	5000/200	5000/200	5000/200	5000/200
Network	Ethernet	Ethernet	Ethernet	Ethernet
Bandwidth (Mbps)	10	10	10	10
Operating System	Mach 2.5A	Mach 2.5A	Ultrix 4.2	Ultrix 4.2
Message Protocol	TCP	UDP	TCP	UDP
Small Msg. (bytes)	54	42	54	42
1-Way Time (μsecs)	700	725	760	760
Wire Time (μsecs)	43	34	43	34
CPU Time (μsecs)	657	691	717	726
Longer Msg. (bytes)	1514	1514	1514	1514
1-Way Time (μsecs)	3020	2940	3065	3025
Wire Time (μsecs)	1211	1211	1211	1211
CPU Time (μsecs)	1809	1729	1854	1814
Fixed Cost (μsecs)	307	331	337	347
Var. Cost (μsecs/KB)	404	361	398	378
SPECInt92 Rating	19.5	19.5	19.5	19.5
Fixed Cost (instr.)	5990	6448	6581	6776
Var. Cost (instr./KB)	7877	7041	7770	7379

Latencies taken from Table 2 in Maeda/Bershad [44]; note that our message sizes are an estimate of the total bytes sent.

Figure A-6. Kernel-Based Messaging Costs

We estimate the fixed and variable CPU costs associated with message processing using a simple back-of-the-envelope calculation. Starting with two reported round-trip latencies (for a small and a large message size), we halve these numbers to get 1-way latencies and then subtract out the 1-way wire times; this should leave us with just the CPU-based costs. Assuming a simple linear model, we get fixed and variable costs in μsecs and μsecs/KB; we divide these results in half since our model assigns half the cost to the sending processor and half to the receiving processor. Finally, since our network model has instruction-based parameters, we multiply by the SPECInt92 rating of the machine used: our final fixed and variable CPU costs are in terms of instructions and instructions/KB (to be charged at both the sender and receiver). Note that the same machine was used as both sender and receiver in these studies, thus it does not matter whether we convert to instructions before or after we divide the computed costs in half.

Figure A-6 shows how this process works, by estimating fixed and variable CPU costs based on reported latencies from a Maeda/Bershad study [44] that used two DECStation 5000/200's and a 10 Mbps Ethernet. (Other studies report similar latencies for monolithic kernels, *e.g.*, [53] reports similar Ethernet/Ultrix latencies for the 5000/200.) Each column shows one back-of-the-envelope calculation; the final estimated costs are at the bottom. We give the estimates for two monolithic kernels (Mach 2.5A and Ultrix 4.2) and two network protocols (TCP and UDP). It is clear that there is some consistency across operating systems and also across protocols. The fixed cost range is 5990–6776 instructions, while the variable cost range is 7041–7877 instructions/KB. Given these ranges, we chose a fixed cost of 6000 instructions and a variable cost of 7168 instructions (7 instructions/byte) for our CURRENT system.

As discussed in Section 4.1.5, our network model assumes there is no overlap between the "wire time" of a message and the component of message latency that is due to CPU costs at the sender and receiver. Also, the entire message traverses the network as a single unit. The accuracy of this model depends in part on the preferred (or standard) size of a network packet, and on the size of the message. If a real system uses packetization and reassembly, the "wire time" of one packet can overlap with the CPU time involved in the processing of another packet.

In an Ethernet setting, messages larger than 1514 bytes use multiple packets, and some overlap will occur. Based on some simple experiments that we ran using the same machines as the ones used by Maeda/Bershad, the "apparent" variable cost for a large message is roughly 4 instructions/byte. For smaller messages, we observed the same variable cost of 7 instruction/byte that we calculated above. In other words, while the actual variable cost is 7 instructions per byte, there is roughly a 3 instruction/byte overlap between CPU time and wire time, producing an apparent variable cost of 4 instructions/byte.

Since we did not want to change our network model, we were faced with a choice: we could stick with the computed variable cost, or we could use a lower one. The first choice produces the correct instruction charges at each end, but results in overestimates of message latencies for large messages. (For a 4 KB message, the resulting latency is roughly 15% too high.) The second choice produces more accurate large-message latencies, but the sending and receiving processors are under-charged. We decided to stick with the original 7 instructions/byte variable charge, for two reasons. First, note that large messages are mostly used for fetch replies (which send 4 KB pages). AOCC and ACBL use the same number of fetches per commit, unless AOCC has a high abort rate, in which case it can use more fetches than ACBL. Thus, the higher latency for large messages is incurred by both schemes in a roughly equal amount. The main messaging difference between the two schemes is that ACBL uses lock requests and callback requests that AOCC does not; these requests (and the resulting replies) are small, thus our model computes correct latencies for them. Second, our network model assumes that there is no other message traffic in the system; in reality, an distributed OODB would share a network with other applications, and thus a 15% stretching of large message latencies hardly seems out of line.

Note that our choice produces both the correct charges and the correct wire utilization charge; the only thing it gets wrong is the overlap of these two charges. In contrast, using a lower variable cost of 4 instructions/byte changes this cost by a factor of 3/7, or 43%. We felt that using the more accurate present-day charge for message CPU costs was important, since a primary distinction between AOCC and ACBL is the number of messages sent. Under-charging for messages would not accurately reflect the effect of high message traffic on the server CPU, and we expected this effect would prove to be important in our experiments.

Machine	Sparc-20	Sparc-20
Network	Fore ATM	Fore ATM
Bandwidth (Mbps)	140	140
Message Protocol	TCP	UDP
Small Msg. (bytes)	48	48
1-Way Time (μsecs)	79	69
Wire Time (μsecs)	3	3
CPU Time (μsecs)	76	66
Longer Msg. (bytes)	1000	1000
1-Way Time (μsecs)	180	170
Wire Time (μsecs)	57	57
CPU Time (μsecs)	123	113
Fixed Cost (μsecs)	37	32
Var. Cost (μsecs/KB)	25	25
SPECInt92 Rating	98.2	98.2
Fixed Cost (instr.)	3615	3124
Var. Cost (instr./KB)	2484	2484

Latencies are from the Figure 9 graph in von Eicken et. al. *[56]; the numbers we use are "best guesses" for some data points on this graph.*

Figure A-7. Reported Latencies: User-Level Messaging

We now consider FUTURE system CPU costs. Figure A-7 shows our back-of-the-envelope calculations for TCP and UDP latencies over a prototype user-level messaging system [56] called UNet. Note that the measured latencies have dropped dramatically from those in Figure A-6. The variable cost for both cases is only 25 μsecs per KB. The fixed cost is 32–37 μsecs. Since a faster machine was used, when we convert to instructions using the Sparc-20's SPECInt92 rating, the improvement in terms of instructions is still good, but less dramatic.

For the present-day CPU costs that we computed using the Maeda/Bershad study, an Ethernet was used and at most one packet was sent per message. In contrast, for these future CPU costs, an ATM packet size that holds 48 data bytes was used; there was clearly overlap between computation and "wire time." Thus, our computed costs are probably under-estimates of the actual CPU costs associated with U-Net messaging. Nevertheless, we decided to use these numbers; perhaps this moves us "further" into the future than the current U-Net implementation. Since U-Net was built with existing hardware, even better performance should be possible in the future. Starting with the numbers in Figure A-7, we "round down" to obtain 3000 instructions fixed cost and 2 instructions/byte variable cost (2048 instructions/KB) for our FUTURE system.

In contrast to the CURRENT system, where our goal was to accurately model present-day costs based on measurements of existing operating systems, we can only guess at future costs; the von Eicken results simply act as a guide. Since our hope was that we could use the FUTURE system to study the impact of much lower messaging costs, we push the von Eicken values in this direction. (Note that our FUTURE server has a MIPS rating that is roughly twice the SPECInt92 rating from the von Eicken study; thus while we use instruction estimates that are based on the U-Net results, we end up with lower latencies by simulating a faster processor.)

A.3 Disk Settings

Parameter	CURRENT	FUTURE
Disk setup cost	5000 instr.	5000 instr.
Slow disk bandwidth	3322 μsecs/KB	2580 μsecs/KB
Fast disk bandwidth	1288 μsecs/KB	990 μsecs/KB
Disks per server	4	8

Figure A-8. Disk Parameters

Figure A-8 gives the CURRENT and FUTURE settings for the disk parameters.

All disk accesses are 4 KB accesses, the size of a database page. The CPU cost charged to the server should reflect the total cost of setting up a 4 KB disk access. (Since the simulator does not charge for the interrupt and event scheduling that occurs when a disk access completes, the setup cost should also reflect completion-based costs as well.) We use the same CPU setup cost for our CURRENT and FUTURE systems: 5000 instructions. (This is the same charge used in [10].) Since the FUTURE CPU's are 4 times faster, FUTURE disk setup times are 4 times faster.

A.3.1 Choosing the Disk Bandwidths

We chose the slow and fast bandwidths for the CURRENT system using the characteristics of a Seagate Barracuda, an existing drive in common use. Its average seek time is 8.75 ms, its rotation time is 8.33 ms, and its average transfer rate for 4 KB is 0.37 ms. An average 4 KB read should therefore take 13.3 ms (using the average seek time, half the rotation time, and the average transfer time). We use this random 4 KB access time for our slow bandwidth. Using the units of our disk model, the slow bandwidth is thus 3322 μsecs/KB.[1]

Scheduler	% of Total Bandwidth	% of Average Seek
Random	7%	100%
Seek Based	17%	37%
+ Rotation Based	25%	23%

Numbers Derived from Seltzer et. al. *[50]*

Figure A-9. Results of Intelligent Disk Scheduling

For the fast bandwidth, we need to determine the bandwidth one would expect from the Barracuda if an intelligent scheduling algorithm were used to install the object updates stored in the modified object buffer (MOB). In a real system, a MOB could hold a very large set of object updates. When the MOB grows large enough to trigger an update installation process, there could be thousands of pages requiring update; there would be many pages for an intelligent scheduling algorithm to choose from.

Fortunately, the problem of scheduling disk accesses out of a large pool of possible choices has already been studied. In Seltzer, Chen, and Ousterhout [50], a detailed disk simulator is used to compare different scheduling algorithms for scheduling up to 1000 4 KB disk writes. Three schedulers are compared: a random scheduler, a scheduler that optimizes seek times but ignores rotational latency (it scans across the disk in one direction), and a scheduler that takes both seek and rotational latencies into account (essentially a shortest-time-first strategy, modified to also force the disk arm to move across the disk). The last approach achieves the best throughput, but requires the scheduler to know more about the disk's performance characteristics.

Figure A-9 reports two metrics for each scheduler: bandwidth achieved, expressed as a percent of the disk's total bandwidth (the speed that the bits are flying under the disk head); and the average seek plus rotation time, expressed as a percent of the random scheduler's seek plus rotation time. The improvements achieved by the two intelligent schedulers are impressive, and we believe that similar schedulers could be used by our server to install updates from the MOB.

Note that unlike the write-only schedulers just described, our scheduler would need to schedule both a read and a write for some pages; pages with updates in the MOB are not necessarily in the server cache. Thus, the scheduling algorithms from the Seltzer study would need to be modified. A simple strategy is possible here: use a two-step scheduling process where K reads are followed immediately by $K+N$ writes (the K pages that were just read would be written, plus an additional N nearby pages that were already cached could be updated). If the schedule for the K reads is a good one, the schedule for the $K+N$ writes should be as good, and possibly better,

[1] A separate detailed disk simulator [42] was used to simulate random 4 KB reads for the Barracuda; the average access time was very close to our calculated value.

since there may be more operations (for $N > 0$). There might be an additional seek between the read and write phase. *E.g.*, a seek-based algorithm could scan in one direction to do K reads, seek back to the first read location, and scan in the same direction to do $K + N$ writes. The cost of the extra seek would be amortized over the $2K + N$ disk operations, and should be negligible. In general, then, we believe algorithms that are as effective as the ones described in Seltzer *et. al.* are possible for a database server.

Since the two intelligent algorithms optimize seek time or seek plus rotation time, the second column in Figure A-9 gives us a scaling factor that we can apply to other disk drives. (In contrast, the reported bandwidth percentages in the first column are specific to the Fujitsu Eagle drive that was used in the study, and cannot be applied directly to other drives.) We assume an algorithm that does as well as the seek-based algorithm, achieving 37% of the seek plus rotation time of a random scheduler. Using numbers from the Barracuda, our fast bandwidth comes out to 1288 μsecs/KB, based on a 5.15 ms 4 KB access time. (The smarter algorithm would give us a bandwidth of 836 μsecs/KB. However, the seek-based algorithm is easier to implement, and is not dependent on the characteristics of a particular disk drive, so it is clearly the better choice from a practical standpoint.)

The performance of FUTURE disk drives is hard to predict. Ruemmler and Wilkes [49] note that while rotational speed was stuck at 3600 RPM for many years, it has been steadily increasing (high performance disks are currently using 7200 RPM) and should continue to increase at a compound rate of 12% per year. They also note that linear density is increasing by about 12% per year. Interestingly, they do not discuss future seek times directly. They do mention that the number of tracks per inch is increasing at roughly 41% per year (with the improved linear density factored in, overall density should improve at 60% per year). Thus, the standard disk size used by most workstations will eventually be smaller than the size used today. This should reduce seek times; there is less distance to travel, and stiffer disk arms can be used for smaller disks (allowing for high acceleration). However, "settling" of the disk head onto the desired track should get harder with more tracks packed into less space. If we assume that seek times are roughly tied to disk size, and that we will switch from 3.5 inch disks to 2.5 inch disks, average seek times might drop by around 25%. If this takes two years, the drop in seek times will be 12% per year compounded — perhaps 12% is a magic number for disks!

Let us assume, then, that a 12% per year improvement applies to linear density, rotational speed, and average seek time. Starting with the Barracuda numbers, in two years we can expect a disk that spins at 9032 RPM (6.64 ms rotational latency), has an average seek time of 6.78 ms, and has an average 4 KB transfer time of 0.225 ms. As a result, our small bandwidth would be 2580 μsecs/KB and our fast bandwidth would be 990 μsecs/KB (again assuming that an intelligent scheduler can reduce seek plus rotation time to 37% of that achieved by a random scheduler.) We decided to use these bandwidths for our FUTURE system.

A.3.2 Choosing the Disks Per Server

While future disk performance is hard to predict, there is one thing we do know for certain: disk performance is not improving as fast as CPU or network performance. Thus, for overall disk bandwidth to "keep pace" with the other system components, more disk drives (or more disk heads per disk drive) will be required. We expect future systems will use many more disks per server, thus our FUTURE system has twice as many disks per server as our CURRENT system. We chose 4 and 8 disks per server, respectively. Below, we describe how we arrived at the 4 disk setting for the CURRENT system. (For the FUTURE system, we were more arbitrary; we simply doubled the number of disks.)

For most system parameters, such as disk and network bandwidth, we looked to existing technology to choose a CURRENT parameter setting. The number of disks per server, on the other hand, varies widely from system to system. In fact, system administrators tend to add disk capacity as necessary, when the current server disks become a resource bottleneck. As a result, we could choose almost any setting (within practical bounds) for the number of server disks and claim that this was a reasonable choice. However, the number of disks used (and the resulting overall disk bandwidth) is important with respect to a comparative study of concurrency control schemes; it would be a mistake to simply choose an arbitrary value for this setting.

Instead, it is important to look at the characteristics of the workloads used, and to choose an aggregate bandwidth that satisfies two conditions. First, the disk bandwidth should not be so low that all of the workloads saturate the disks at only a few clients. This case is clearly uninteresting for a concurrency control study: the performance of all of the schemes will depend almost entirely on the disk bandwidth; any performance differences between schemes that might have been observed using a higher disk bandwidth will be "masked," and all concurrency control schemes will appear to have roughly the same performance. Second, the disk bandwidth should not be so high that it is impossible for any of our workloads, regardless of setup, to saturate the disks. At least one of our workloads, the UNIFORM workload, exhibits very poor locality of reference, causing many fetch requests. Since we expect real applications will exhibit better locality, and thus will require less overall disk bandwidth, it is unlikely that systems will be configured with enough disks to handle UNIFORM without at least approaching disk saturation. In other words, our hypothetical system administrator is not going to purchase twice as many disks as are actually needed to provide users with the performance that they expect from the system.

Keeping these conditions in mind, we settled on the following approach for choosing the number of disks per server. First, looking at average disk operations per commit, we chose the HOTCOLD workload as our "mean workload" with respect to use of the server disk. We then ran some experiments that used all other CURRENT settings as specified but varied the number of disks per server, and we chose the lowest disks-per-server setting that did not cause the disk utilization to rise above 90% until the number of clients reached 20 or more. This approach ensured two things. First, for HOTCOLD in particular, we know that we will be able to examine both the case where the disks are not saturated (*e.g.*, in the 1–20 client range) and also the case where the disks are saturated (above 20 clients). Second, since HOTCOLD is our "median workload," we expect that some of our workloads will not saturate the disks at all, while some will saturate the disks more quickly than HOTCOLD does. We find this to be the "intuitively correct" setup for our experiments: workloads that use the server disks frequently should run into disk saturation problems, while workloads that rarely use the server disks should not.

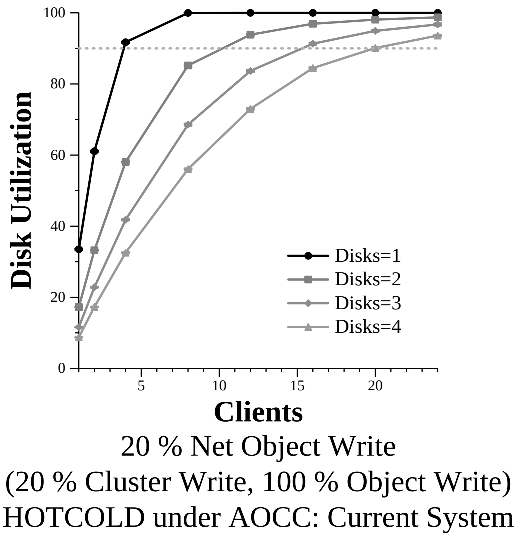
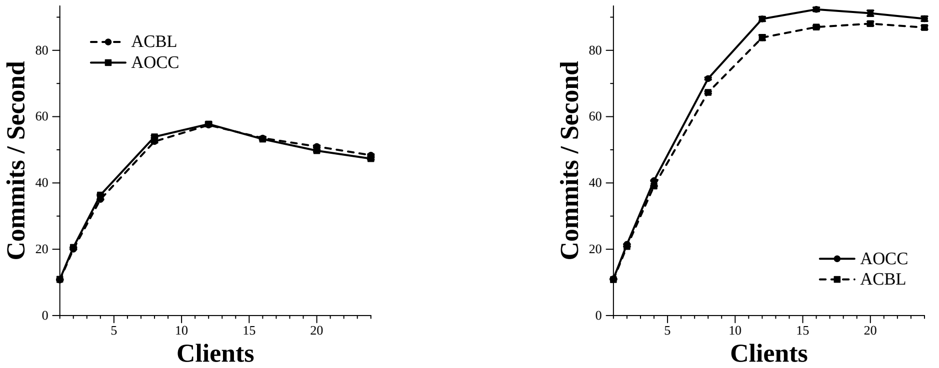


Figure A-10. Disk Utilization vs. Number of Server Disks

Figure A-10 shows the average disk utilization when we run HOTCOLD under the CURRENT system and vary the number of disks from 1 to 4. A 20% net object write probability was used. We show AOCC disk utilization since in it is higher than the ACBL utilization. Note that the dotted line shows the 90% disk utilization point: a 1-disk system reaches this point by 4 clients; a 2-disk system by 10 clients; a 3-disk system by 16 clients; and a 4-disk system by 20 clients.

For the 4-disk case, we expect that differences between AOCC and ACBL can be observed, if there are any (at least below the 20 client point, where AOCC reaches 90% disk utilization). For the 1-, 2-, and 3-disk cases, on the other hand, we expect the high disk utilization to "mask" any differences between the two schemes. These expectations prove to be correct. In Figure A-11, we show the throughput results for AOCC and ACBL for both a 2-disk and 4-disk system; the 2-disk system results are on the left, while the 4-disk results are on the right. For the 2-disk system, the two schemes perform almost identically across the entire client range. For the 4-disk system, however, we see that AOCC can outperform ACBL. AOCC is the better scheme for this workload; given sufficient disk bandwidth, it can run more transactions per second. However, without sufficient bandwidth, it is constrained to run the same number of transactions per second as ACBL.

There are two other points we should make using the graphs in Figure A-11. First, note that once the server disks are saturated, adding clients to the system tends



20 % Net Object Write
(20 % Cluster Write, 100 % Object Write)
HOTCOLD: Current System, 2 Disks

20 % Net Object Write
(20 % Cluster Write, 100 % Object Write)
HOTCOLD: Current System, 4 Disks

Figure A-11. "Masking Effect" of Disk Saturation

to reduce system throughput. The system cannot perform any *more* commits per second once its disks are saturated, and thus adding additional clients simply places additional load on the server, and causes additional blocking or aborts, thus causing a net drop in overall performance. (In fact one might like to prevent additional clients from "entering" the system once the peak performance level has been reached; see the discussion of adaptive admission control in Chapter 5.) Second, note that both AOCC and ACBL have much better performance under the 4-disk system. Both schemes incur long disk delays due to high disk utilization under the 2-disk system. Thus, there is nothing AOCC-specific about our choice of disks-per-server; even if we were comparing ACBL to another locking scheme, we would still select a 4-disk system for such a study.

References

[1] 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, May 1995.

[2] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed multi-version optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1), 1987.

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

[4] P. Berstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

[5] M. Blumrich, C. Dubnicki, E. W. Felton, and K. Li. Memory-mapped network interfaces. *IEEE Micro*, pages 21–28, February 1995.

[6] P. Butterworth, A. Otis, and J. Stein. The Gemstone database management system. *CACM*, 34(10), October 1991.

[7] M. Carey, D. DeWitt, C. Kant, and J. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *OOPSLA'94: 9th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, October 1994.

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

[9] M. Carey et al. Shoring up persistent applications. In *ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, May 1994.

[10] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-graned sharing in a page server OODBMS. In *ACM SIGMOD International Conference on Management of Data*, pages 359–370, Minneapolis, MN, May 1994.

[11] M. Carey, S. Krishnamurthi, and M. Livny. Load control for locking: The 'half-and-half' approach. In *9th ACM Symposium on Principles of Database Systems*, pages 72–84, Nashville, TN, April 1990.

[12] M. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM TODS*, 16(4):703–746, December 1991.

[13] M. Castro, March 1996. Private Communication.

[14] 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, June 1989.

[15] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *1rst Usenix Symposium on Operating System Design and Implementation*, 1994.

[16] O. Deux et al. The story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[17] O. Deux et al. The O2 system. *CACM*, 34(10), October 1991.

[18] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation architectures for object-oriented database systems. In *16th International Conference on Very Large Data Bases (VLDB)*, Brisbane, Australia, 1990.

[19] P. Drew and R. King. The performance and utility of the Cactis implementation algorithms. In *16th International Conference on Very Large Data Bases (VLDB)*, pages 135–147, Brisbane, Australia, 1990.

[20] P. Druschel, L. L. Peterson, and B. S. Davie. Experience with a high-speed network adaptor: A software perspective. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 2–13, London, England, August 1994.

[21] 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, November 1976.

[22] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *15th ACM Symposium on Operating System Principles*, 1995.

[23] M. Franklin. Caching and memory management in client-server database systems. Technical Report (Ph.D.) 1168, Computer Sciences Dept., University of Wisconsin-Madison, July 1993.

[24] M. Franklin and M. Carey. Client-server caching revisited. In *Int'l Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[25] M. Franklin, M. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *18th International Conference on Very Large Data Bases (VLDB)*, 1992.

[26] M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt. Crash recover in client-server EXODUS. In *ACM SIGMOD International Conference on Management of Data*, San Diego, CA, June 1992.

[27] D. Gerson, May 1989. Private Communication.

[28] S. Ghemawat. *The Modified Object Buffer: a Storage Manamement Technique for Object-Oriented Databases.* PhD thesis, Massachusetts Institute of Technology, 1995.

[29] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symposium on Operating System Principles*, Litchfield Park, Arizona, December 3–6 1989.

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

[31] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structures. In *2nd Usenix Symposium on Operating System Design and Implementation*, October 1996.

[32] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected operation in the thor object-oriented database system. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[33] T. Haerder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, June 1984.

[34] M. P. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report DEC/CRL 92/07, Digital Equipment Corp. Cambridge Research Lab., Cambridge, MA, December 1992.

[35] A. Joshi. Adaptive locking strategies in a multi-node data sharing system. In *17th International Conference on Very Large Data Bases (VLDB)*, Barcelona, September 1991.

[36] A. Kemper and D. Kossmann. Dual-buffer strategies in object bases. In *20th International Conference on Very Large Data Bases (VLDB)*, Santiago, Chile, 1994.

[37] 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, March 1990.

[38] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, June 1981.

[39] M. Y. Lai and W. K. Wilkinson. Distributed transaction management in Jasmin. In *10th International Conference on Very Large Data Bases (VLDB)*, August 1984.

[40] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *CACM*, 34(10), October 1991.

[41] G. Lausen. Formal aspects of optimistic concurrency control in a multi-version database system. *Information Systems*, 8(4):291–301, 1983.

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

[43] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Quebec, Canada, June 1996.

[44] C. Maeda and B. N. Bershad. Protocol server decomposition for high-performance networking. In *14th ACM Symposium on Operating System Principles*, pages 244–255, Asheville, NC, December 1993.

[45] Objetivity. Inc. objectivity/db documentation vol. 1, 1991.

[46] ONTOS. Inc. ONTOS db 2.2 reference manual, 1992.

[47] J. O'Toole and L. Shrira. Hybrid caching for scalable object systems (think globally, act locally. In *6th Int'l Workshop on Persistent Object Systems*, Tarascon, France, September 1994.

[48] E. Rahm and A. Thomasian. A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking. In *10th Int'l Conference on Distributed Computing Systems*, 1990.

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

[50] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Winter Usenix Technical Conference*, pages 313–324, Washington, DC, winter 1990.

[51] J. W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM TOCS*, 2(2):155–180, May 1984.

[52] V. O. Technology. VERSANT system reference manual, release 1.6, 1991.

[53] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, San Francisco, CA, September 1993.

[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, CO, May 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, San Diego, CA, June 1992.

[56] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed comuting. In *15th ACM Symposium on Operating System Principles*, Copper Mountain, CO, December 1995.

[57] Y. Wang. *Performance Studies of Cache Consistency and Concurrency Control Algorithms in a Distributed Client/Server Architecture.* PhD thesis, University of California, Berkeley, 1992.

[58] Y. Wang and L. A. Rowe. Cache consistency and concurrency control in a client/server DBMS architecture. In *ACM SIGMOD International Conference on Management of Data*, pages 367–376, Denver, CO, May 1991.

[59] W. Weihl. Distributed version management for read-only actions. *IEEE Transactions on Software Engineering*, SE-13(1), January 1987.

[60] D. Weinreb, N. Feinberg, D. Gerson, and C. Lamb. An object-oriented database system to support an integrated programming environment. *Database Engineering*, 7(1):85–95, 1988.

[61] K. Wilkinson and M. Neimat. Maintaining consistency of client-cached data. In *16th International Conference on Very Large Data Bases (VLDB)*, pages 122–133, Brisbane, Australia, 1990.