Memory-Mapped Transactions

by

Jim Sukha

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
Bradley C. Kuszmaul
Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Memory-Mapped Transactions

by

Jim Sukha

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Memory-mapped transactions combine the advantages of both memory mapping and transactions to provide a programming interface for concurrently accessing data on disk without explicit I/O or locking operations. This interface enables a programmer to design a complex serial program that accesses only main memory, and with little to no modification, convert the program into correct code with multiple processes that can simultaneously access disk.

I implemented LIBXAC, a prototype for an efficient and portable system supporting memory-mapped transactions. LIBXAC is a C library that supports atomic transactions on memory-mapped files. LIBXAC guarantees that transactions are serializable, and it uses a multiversion concurrency control algorithm to ensure that all transactions, even aborted transactions, always see a consistent view of a memory-mapped file. LIBXAC was tested on Linux, and it is portable because it is written as a user-space library, and because it does not rely on special operating system support for transactions.

With LIBXAC, I was easily able to convert existing serial, memory-mapped implementations of a B^+-tree and a cache-oblivious B-tree into parallel versions that support concurrent searches and insertions. To test the performance of memory-mapped transactions, I ran several experiments inserting elements with random keys into the LIBXAC B^+-tree and LIBXAC cache-oblivious B-tree. When a single process performed each insertion as a durable transaction, the LIBXAC search trees ran between 4% slower and 67% faster than the B-tree for Berkeley DB, a high-quality transaction system. Memory-mapped transactions have the potential to greatly simplify the programming of concurrent data structures for databases.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Bradley C. Kuszmaul
Title: Research Scientist

Acknowledgments

These acknowledgments are presented in a random order:

I would like to thank my advisor, Charles E. Leiserson for his helpful comments on this thesis, and on his helpful advice in general.

I would also like to thank Bradley C. Kuszmaul, who has been deeply involved in this project from day one. Bradley's initial idea started me on this project that eventually became Libxac, and I have had many helpful discussions with him on this topic ever since.

I'd like to thank Bradley for giving me an implementation of a B^+-tree, and Zardosht Kasheff for the code for the cache-oblivious B-tree. The experimental results on search trees without Libxac are primarily the work of Bradley, Michael A. Bender, and Martin Farach-Colton.

All of the people in the SuperTech group deserve a special round of acknowledgments. Those people in SuperTech, also in random order, include Gideon, Kunal, Jeremy, Angelina, John, Tim, Yuxiong, Vicky, and Tushara. Everyone has been wonderfully patient in listening to my ramblings about transactions, names for Libxac, research, and life in general. They have all kept me (relatively) sane throughout the past year. In particular, I'd like to Kunal, Angelina, and Jeremy for their feedback on parts of my thesis draft, and more generally for choosing to serve the same sentence. I'd also like to thank Ian and Elizabeth, who are not in the SuperTech group, but have also been subjected to conversations about transactions and other research topics. Thanks to Leigh Deacon, who kept SuperTech running administratively.

Thanks to Akamai and MIT for the Presidential fellowship that funded my stay here this past year. Also, thanks to the people I met through SMA for their helpful comments on the presentation I gave in Singapore.

Thank you to my family and to my friends, here at MIT and elsewhere. Without their support, none of this would have been possible.

I apologize to all those other people I am sure I have missed that deserve acknowledgments. I will try to include you all when the next thesis rolls around.

This work was partially supported by NSF Grant Numbers 0305606d and 0324974, and by the Singapore MIT Alliance. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Contents

1 Introduction **17**

1.1 Explicit I/O vs. Memory Mapping 19

1.2 Locks vs. Transactions 23

1.3 Concurrent Disk-Based Programs 30

1.4 Thesis Overview 32

2 The Libxac Interface **37**

2.1 Programming with LIBXAC 38

2.2 Semantics of Aborted Transactions 45

2.3 Optimizations for LIBXAC 50

2.4 Related Work 51

2.5 Advantages of the LIBXAC Interface 55

3 The Libxac Implementation **57**

3.1 Overview 58

3.2 Transactions on a Single Process 62

3.3 Concurrent Transactions 67

3.4 Conclusion 74

4 A Performance Study **77**

4.1 The Hardware 78

4.2 Performance of Nondurable Transactions 80

4.2.1 Page-Touch Experiment 80

4.2.2 Page-Touch with an Advisory Function 81
4.2.3 Decomposing the Per-Page Overhead 85
4.3 Durable Transactions 91
4.4 Testing Concurrency in Libxac 92

5 Search Trees Using Libxac 97
5.1 Introduction 98
5.2 Serial B^+-trees and CO B-trees 102
5.3 Search Trees Using Libxac 105
5.4 Durable Transactions on Search Trees 106
5.5 Summary of Experimental Results 108

6 Conclusion 113
6.1 Ideas for Future Work 114
6.2 Libxac and Transactional Memory 115

A Restrictions to the Libxac Interface 125
A.1 Restrictions to Libxac 125

B Transaction Recovery in Libxac 129

C Detailed Experimental Results 135
C.1 Timer Resolution 135
C.2 Page-Touch Experiments 137
C.3 Experiments on Various System Calls 137
C.4 Durable Transactions 141
C.5 Concurrency Tests 144
C.6 Search Trees using Libxac 144

List of Figures

1-1 Two versions of a simple C program that reads the first 4-byte integer from each of 5 randomly selected pages of a file, computes their sum, and stores this value as the first 4-byte integer on a randomly selected 6th page. In Program A, the entire file is brought into memory using `read`, the selected pages are modified, and the entire file is written out to disk using `write`. On the first 5 pages, Program B does an explicit disk seek to read the first integer. Then B does a seek and a write to modify the 6th page. 20

1-2 A third version of the C program from Figure 1-1, written using `mmap`. 22

1-3 Concurrent processes sharing data through a memory mapped file. . . 24

1-4 An interleaving of instructions from the execution of the programs in Figure 1-3 that causes a data race. 24

1-5 Two different locking protocols for the program in Figure 1-2. Program D acquires a global lock, while Program E acquires a different lock for every page. For simplicity, only the body of the code is shown here. . 26

1-6 Program F. This version of the program from Figure 1-2 is written with memory-mapped transactions. 28

1-7 An illustration of the possible space of programs that can concurrently access data on disk. 31

2-1 LIBXAC program that increments the first integer of a memory-mapped file. 40

2-2 A side-by-side comparison of the program in Figure 1-2 and the parallel version written with Libxac. 42

2-3 A recursive function that uses nested transactions. 43

2-4 A transaction that that accesses local variables inside a transaction. . 48

3-1 Changes to the memory map for a simple transaction. 61

3-2 An example of a consistency tree. 69

4-1 A simple transaction that (a) reads from n consecutive pages, and (b) writes to n consecutive pages. 81

4-2 Average time per page to execute the transactions shown in Figure 4-1 on Machine 1. For each value of n, each transaction was repeated 1000 times. 82

4-3 The transactions in Figure 4-1 written with the advisory function. The transaction in (a) reads from n consecutive pages, the transaction in (b) writes to n consecutive pages. 83

4-4 Average time per page to execute the transactions shown in Figure 4-3 on Machine 1. For each value of n, each transaction was repeated 1000 times. 84

4-5 Concurrency Test A: Each transaction increments the first integer on a page 10,000 times. 94

4-6 Concurrency Tests B and C: Test B increments every integer on the page. Test C repeats the transaction in Test B 1,000 times. I omit the outermost for-loop, but as in Figure 4-5, each transaction is repeated 10,000 times. 94

5-1 An illustration of the Disk Access Machine (DAM) model. 99

5-2 The van Emde Boas layout (left) in general and (right) of a tree of height 5. 101

5-3 Machine 3: Time for kth most expensive insert operation. 109

B-1 An example of a Libxac log file when transactions execute (a) on one process, and (b) on two processes. 130

C-1 Machine 1:Distribution of Delay Times Between Successive `gettimeofday` Calls. 137

C-2 Average time per page to execute the transactions shown in Figure 4-3 on Machine 2. For each value of n, each transaction was repeated 1000 times. 138

C-3 Average time per page to execute the transactions shown in Figure 4-3 on Machine 3. For each value of n, each transaction was repeated 1000 times. 139

C-4 Average time per page to execute the transactions shown in Figure 4-3 on Machine 4. For each value of n, each transaction was repeated 1000 times. 140

List of Tables

2.1 The LIBXAC functions for nondurable transactions. 39

4.1 Processor speeds and time per clock cycle for the test machines. . . . 80

4.2 Average # of clock cycles per page access for transactions touching 1024 pages, with and without the advisory function. Numbers are in thousands of cycles. Percent speedup is calculated as $100\left(\frac{\text{Normal - With Adv}}{\text{Normal}}\right)$. 85

4.3 Number of clock cycles required to enter `SIGSEGV` handler, call `mmap`, and exit handler (average of 10,000 repetitions). 86

4.4 Clock cycles required to write to a page for the first time after memory mapping that page. Each experiment was repeated 5 times. 87

4.5 Clock cycles required for a `memcpy` between two 4K character arrays in memory. 88

4.6 Average # of clock cycles per page access for transactions touching 1024 pages. All numbers are in thousands of clock cycles. 89

4.7 Average Access Time (μs) per Page, for Transactions Touching 1024 Pages. 91

4.8 Time required to call `msync` and `fsync` on a 10,000 page file with one random page modified, 1000 repetitions. All times are in ms. 93

4.9 Concurrency tests for nondurable transactions. Times are μs per transaction. Speedup is calculated as time on 1 processor over time on 2 processors. 95

4.10 Concurrency tests for durable transactions. Times are milliseconds per transaction. 96

5.1 Performance measurements of 1000 random searches on static trees. Both trees use 128-byte keys. In both cases, we chose enough data so that each machine would have to swap. On the small machine, the CO B-tree had 2^{23} (8M) keys for a total of 1GB. On the large machine, the CO B-tree had 2^{29} (512M) keys for a total of 64GB. 103

5.2 Timings for memory-mapped dynamic trees. The keys are 128 bytes long. The range query is a scan of the entire data set after the insert. Berkeley DB was run with the default buffer pool size (256KB), and with a customized loader that uses 64MB of buffer pool. These experiments were performed on the small machine. 104

5.3 The time to insert a sorted sequence 450,000 keys. Inserting sorted sequence is the most expensive operation on the packed memory array used in the dynamic CO B-tree. 105

5.4 Changes in Code Length Converting B^+-tree and CO B-tree to Use LIBXAC. 106

5.5 Time for 250,000 durable insertions into LIBXAC search trees. All times are in ms. Percent speedup is calculated as $\frac{100(t_1-t_2)}{t_2}$, where t_1 and t_2 are the running times on 1 and 2 processors, respectively. 107

5.6 The % speedup of LIBXAC search trees over Berkeley DB. Percent speedup is calculated as $\frac{100(t_L-t_B)}{t_B}$, where t_L and t_B are the running times on the LIBXAC and the Berkeley DB tree, respectively. Speedup is t_1/t_2. 107

C.1 Delay (in clock cycles) between successive calls to timer using `rdtsc` instruction, 10,000 repetitions. 136

C.2 Delay between successive calls to `gettimeofday` (in μs), 10,000 repetitions. 136

C.3 Timing data for entering `SIGSEGV` handler, calling `mmap`, and leaving handler, 10,000 repetitions. All times are processor cycles. 141

C.4 Clock cycles to do 1,000 calls to `memcpy` between two 4K character arrays in memory, 1,000 repetitions. times are in μs. 141
C.5 Average Access Time (μs) per Page, for Transactions Touching 1024 Pages. 142
C.6 Timing data for calling `msync` and `fsync` on a 10,000 page file with a random page modified, 1000 repetitions. All times are in μs. 143
C.7 Time to write 10,000 pages to a file, 1,000 repetitions. All times are in μs. 143
C.8 Time to compute SHA1 and MD5 hash functions on a single page. All times are in thousands of clock cycles. 143
C.9 Concurrency tests for nondurable transactions. Times are μs per transaction. 145
C.10 Concurrency tests for durable transactions. Times are per transaction. 146
C.11 Time to do 250,000 nondurable insertions into LIBXAC search trees. . 147
C.12 Time to do 250,000 durable insertions on a single process into the various search trees, with write-caches on the harddrives enabled. All times are in ms. 148

Chapter 1

Introduction

In this thesis, I argue that memory-mapped transactions provide a simple yet expressive interface for writing programs with multiple processes that concurrently access persistent data. Memory-mapped transactions rely on two important components: memory mapping and transactions. Memory mapping uses virtual-memory mechanisms to present the programmer with the illusion of a single level of storage, simplifying code by allowing a program to access data on disk as though it were stored in main memory, without explicit input/output (I/O) operations. Transactions simplify parallel programs by providing a mechanism for easily specifying that a critical section of code executes atomically, without using locks. Memory-mapped transactions combine the advantages of both memory mapping and transactions to provide an interface that allows programs to concurrently access data on disk without explicit I/O or locking operations. This interface allows a programmer to design a complex serial program that accesses only main memory, and with little to no modification, convert the program into correct code with multiple processes that can simultaneously access disk.

To demonstrate my thesis, I implemented LIBXAC, a prototype for an efficient and portable system supporting memory-mapped transactions. LIBXAC is a C library that supports atomic transactions on memory-mapped files. With LIBXAC, I was easily able to convert existing serial, memory-mapped implementations of a B^+-tree and a cache-oblivious B-tree (CO B-tree) into parallel versions that support concurrent

searches and insertions. To test the performance of memory-mapped transactions in an actual application, I ran several experiments inserting 250,000 elements with randomly chosen keys into the LIBXAC B$^+$-tree and LIBXAC CO B-tree. When a single process performed each insertion as a durable transaction, the LIBXAC search trees ran between 4% slower and 67% faster than the B-tree for Berkeley DB [44], a high-quality transaction system. This result shows that the LIBXAC prototype can support durable memory-mapped transactions on a single process efficiently in an actual application.

In the remainder of this chapter, I exhibit several example programs that illustrate how memory-mapped transactions can be both efficient and easy to use. First, I explain how a program written with memory mapping more easily achieves both of these advantages simultaneously, compared to a program written with explicit I/O. Section 1.1 describes three programs that access 6 random locations of a file on disk. The first program that uses explicit I/O to read and buffer the entire file in an array in memory is easy to code but is inefficient. The second program that does an explicit I/O operation before every access to the file is more efficient, but harder to code. The third program that uses memory mapping, however, is both efficient and easy to code.

Next, I show that in code with multiple processes, transactions can be both efficient and easy to use compared with explicit locking. Programmers traditionally use locks in parallel programs to avoid errors caused by data races. In Section 1.2, I present two programs that use both locks and memory mapping to concurrently access disk. The first program that uses a global lock is easy to code, but is inefficient because the accesses to shared memory are effectively serialized. The second program that uses a fine-grained locking scheme is more efficient because it admits more concurrency, but it is more difficult to program correctly. I then describe a third program using transactions that is as easy to code as the first program but is also as efficient as the second program.

Memory-mapped transactions combine the advantages of both memory mapping and transactions to provide a simple interface for programs with multiple processes that simultaneously access disk. I conclude this chapter in Section 1.4 by illustrating

how memory-mapped transactions fit into the space of programs that concurrently access disk, and by presenting an outline of the rest of this thesis.

1.1 Explicit I/O vs. Memory Mapping

For serial programs that access data on disk, memory-mapped transactions exhibit many of the advantages of normal memory mapping. In this section, I present two versions of a program that both use explicit I/O operations instead of memory mapping to access disk. The first program is easy to code but inefficient, while the second program is more efficient but more difficult to code. Finally, I describe a third program using memory mapping that is both efficient and easy to code, thereby retaining the advantages of both explicit I/O solutions.

Using Explicit I/O to Access Disk

Consider a toy C program which reads the first 4-byte integer from each of 5 randomly selected 4096-byte pages of a file, computes their sum, and stores this sum in a 6th randomly-selected page. Figure 1-1 illustrates two versions of this program, both coded in C using explicit I/O operations. Program A uses the `read` system call to buffer the entire file in memory, does the entire computation, and then uses the `write` system call to save the modifications to disk. Program B does an explicit `read` on each of the 5 integers and then uses `write` to modify the value of the 6th integer.

Program A has the advantage that the main body (Lines 13 through 22) is simple to code. The program can easily manipulate the data because it is buffered in the array `x`. This version of the code also requires only two explicit I/O operations. If the file happens to be stored sequentially on disk, then these operations cause only two disk seeks: one to read in the file and one to write it back out to disk.

On the other hand, Program A is inefficient when the file is large compared to the number of pages being accessed. If the program were accessing 50,000 different pages, then reading in all 100,000 pages of the file might make sense. Reading the whole file into memory is wasteful, however, when a program accesses only 6 different

```
// Program A buffers the
//   entire file.

1  int fileLength = 100000;
2  int main(void) {
3    int i, j, sum = 0;
4    int fd;
5    int* x;
6
7    fd = open("input.db",
8              O_RDWR, 0666);
9    x=(int*)malloc(4096*fileLength);
10   read(fd, (void*)x,
11        4096*fileLength);
12
13   for (j = 0; j < 5; j++) {
14       i = rand() % fileLength;
15
16
17       sum += x[1024*i];
18   }
19
20   i = rand() % fileLength;
21
22   x[1024*i] = sum;
23
24   lseek(fd, 0, SEEK_SET);
25   write(fd, (void*)x,
26         4096*fileLength);
27   close(fd);
28   return 0;
29 }
```

A

```
// Program B reads and writes
//  each int individually.

1  int fileLength = 100000;
2  int main(void) {
3    int i, j, sum = 0;
4    int fd;
5    int value;
6
7    fd = open("input.db",
8              O_RDWR, 0666);
9
10
11
12
13   for (j = 0; j < 5; j++) {
14      i = rand() % fileLength;
15      lseek(fd, 4096*i, SEEK_SET);
16      read(fd, &value, 4);
17      sum += value;
18   }
19
20   i = rand() % fileLength;
21   lseek(fd, 4096*i, SEEK_SET);
22   write(fd, &value, 4);
23
24
25
26
27   close(fd);
28   return 0;
29 }
```

B

Figure 1-1: Two versions of a simple C program that reads the first 4-byte integer from each of 5 randomly selected pages of a file, computes their sum, and stores this value as the first 4-byte integer on a randomly selected 6th page. In Program A, the entire file is brought into memory using `read`, the selected pages are modified, and the entire file is written out to disk using `write`. On the first 5 pages, Program B does an explicit disk seek to read the first integer. Then B does a seek and a write to modify the 6th page.

pages.

Program B is more efficient because it avoids reading in the entire file by doing an explicit disk `read` or `write` to access each page. The tradeoff is that the main body of Program B is cluttered with additional explicit I/O operations. Program A simply reads the first integer on a page in Line 17 by accessing an array in memory. In Lines 15 and 16, Program B must first position the cursor into the file and then read in the value. For a larger program with more complicated data structures and a more complicated data layout, it is cumbersome to repeatedly calculate the correct file offsets every time the program accesses a new piece of data.

Using Memory Mapping to Access Disk

A version of program that uses memory mapping can be both easy to code and efficient compared to a version written with explicit I/O. Figure 1-2 presents Program C, a program that uses memory mapping to combine the ease of programming of Program A with the efficiency of Program B. In Lines 9–11, Program C uses `mmap` to memory-map the entire file instead of reading it into memory. After calling `mmap`, the program can access the file through the pointer `x` as though it was a normal array in main memory. Thus, the body of the two programs, Lines 13–22, are exactly the same.

Although Program C appears similar to Program A, Program C is still efficient because the operating system only buffers those pages of the file that the program accesses through `x`. Thus, the programmer achieves the efficiency of Program B without coding explicit `read` and `write` operations. Memory mapping gives programmers the best of both worlds.

Although Program C in Figure 1-2 is only a toy example, its behavior is designed to match the page-access pattern of more practical applications. For example, consider an application that checks for a path between two vertices in a graph. On a large graph, a long path is likely to jump around to vertices stored on different pages. Similarly, another application that can generate seemingly random memory accesses is following pointers down a linked list.

```
// Program C memory-maps
//   the file.

1  int fileLength = 100000;
2  int main(void) {
3    int i, j, sum = 0;
4    int fd;
5    int* x;
6
7    fd = open("input.db",
8              O_RDWR, 0666);
9    x=(int*)mmap(0, 4096*fileLength,
10                PROT_READ|PROT_WRITE,
11                MAP_SHARED, fd, 0);
12
13   for (j = 0; j < 5; j++) {
14      i = rand() % fileLength;
15
16
17      sum += x[1024*i];
18   }
19
20   i = rand() % fileLength;
21
22   x[1024*i] = sum;
23
24   munmap(x, fileLength);
25
26
27   close(fd);
28   return 0;
29 }
```

C

```
// Program A buffers the
//   entire file.

1  int fileLength = 100000;
2  int main(void) {
3    int i, j, sum = 0;
4    int fd;
5    int* x;
6
7    fd = open("input.db",
8              O_RDWR, 0666);
9    x=(int*)malloc(4096*fileLength);
10   read(fd, (void*)x,
11        4096*fileLength);
12
13   for (j = 0; j < 5; j++) {
14       i = rand() % fileLength;
15
16
17       sum += x[1024*i];
18   }
19
20   i = rand() % fileLength;
21
22   x[1024*i] = sum;
23
24   lseek(fd, 0, SEEK_SET);
25   write(fd, (void*)x,
26         4096*fileLength);
27   close(fd);
28   return 0;
29 }
```

A

Figure 1-2: A third version of the C program from Figure 1-1, written using mmap.

1.2 Locks vs. Transactions

For programs with multiple processes that access shared memory, memory-mapped transactions represent a more convenient alternative than locks. In this section, I illustrate an example of a program written using memory-mapped transactions that is more efficient and easier to code than two corresponding programs both written using locks.

First, I illustrate how a data race in parallel code can cause a program error, and then describe locking, the traditional solution for eliminating data races. I then present two versions of the program from Figure 1-2 coded with locks. The first version is correct and easy to code, but it has poor performance when two processes each run a copy of the program concurrently. The second version executes more efficiently in parallel, but it is incorrect because it has the potential for deadlock. Finally, I describe a third program using memory-mapped transactions that is as simple to code as the first program, but still executes efficiently in parallel as the second program.

Data Races in Parallel Programs

Memory mapping, in addition to facilitating access to data on disk, also allows multiple processes to share data through a memory-mapped file. Consider the two C programs in Figure 1-3. Programs 1 and 2 increment and decrement the first 4-byte integer in the file `input.db`, respectively. If `x[0]` has an initial value of 42, then if both programs run concurrently, we expect that after both programs finish executing, the value of `x[0]` will still be 42.

It is possible, however, that the value of `x[0]` may be corrupted by a data race. A *data race* occurs when two or more concurrent processes try to read or write from the same memory location simultaneously with at least one of those accesses being a write operation [43]. For example, consider an execution where the machine instructions for `x[0]++` and `x[0]--` are interleaved as shown in Figure 1-4. The increment and decrement operations in Figure 1-3 may be decomposed into three machine instruc-

```
// Program 1
int main(void) {
  int fd;
  int* x;
  fd = open("input.db",
            O_RDWR, 0666);
  x=(int*)mmap(x, 4096,
               PROT_READ |
                 PROT_WRITE,
               MAP_SHARED, fd, 0);

  x[0]++;

  munmap(x, 4096);
  close(fd);
  return 0;
}
```

```
// Program 2
int main(void) {
  int fd;
  int* x;
  fd = open("input.db",
            O_RDWR, 0666);
  x=(int*)mmap(x, 4096,
               PROT_READ |
                 PROT_WRITE,
               MAP_SHARED, fd, 0);

  x[0]--;

  munmap(x, 4096);
  close(fd);
  return 0;
}
```

Figure 1-3: Concurrent processes sharing data through a memory mapped file.

Program 1	Program 2	x[0]	R1	R2
R1 ← x[0]		42	**42**	–
	R2 ← x[0]	42	42	**42**
R1 ← R1 + 1		42	**43**	42
	R2 ← R2 - 1	42	43	**41**
x[0] ← R1		**43**	43	41
	x[0] ← R2	**41**	43	41

Figure 1-4: An interleaving of instructions from the execution of the programs in Figure 1-3 that causes a data race.

tions: the first loads the value of x[0] into a register, the next updates the value, and the last stores the register back into memory. Suppose that the initial value of x[0] is 42. If both programs load the value of x[0], after which both programs store a new value, then the final value of x[0] may be 41 or 43, depending on which program completes its store to x[0] first. Such nondeterministic behavior is usually a programming error.

Parallel Programming with Locks

The traditional method for eliminating data races is to acquire locks before executing critical sections of code. Locks guarantee *mutual exclusion*, i.e., that the critical sections of code do not execute concurrently. A straightforward method for eliminating the data race from the code in Figure 1-3 is for each program to acquire a global lock, modify `x[0]`, and then release the global lock. Using the lock ensures that the interleaving of operations in Figure 1-4 cannot occur.

For more complicated programs, however, it is not always clear what the best locking protocol to use is. Suppose that we have two processes, each running a copy of Program C from Figure 1-2 concurrently. With different random seeds, each process most likely accesses different pages. A correct version of Program C still requires locks, however, as it is still possible to have a data race if both processes happen to access the same random page and one of the processes is writing to that page. Figure 1-5 illustrates Programs D and E, two versions of Program C that both use locks.

Program D is correct, but it exhibits poor performance when two copies of D run on concurrent processes. Because Program D acquires a global lock, it is impossible for the processes to update the array `x` concurrently. The critical sections in each process execute serially even if they could have correctly run simultaneously.

Program E acquires a lock before it accesses every page, allowing two processes that each run a copy of Program E to execute concurrently when the set of pages they touch is disjoint. Program E is more efficient than Program D, but unfortunately E suffers from the problem of being incorrect. Because the program randomly selects pages to touch, there is no specified order that each program follows when acquiring locks. Thus, it is possible for the system to *deadlock* if each program waits to acquire a lock that is held by the other. For example, one process running Program E could acquire a lock on page 10 be waiting to acquire the lock on page 43, while the other process has already acquired the lock on page 43 and is waiting on the lock for page 10.

```
//  Program D: Global Lock

1    lockVar globalLock;
2
3      ...
4    lock(globalLock);
5    for (j = 0; j < 5; j++) {
6      i=rand()%fileLength;
7
8      sum += x[1024*i];
9    }
10
11   i=rand()%fileLength;
12
13   x[1024*i] = sum;
14
15   unlock(globalLock);
16
17
```

D

```
//  Program E: Page-Granularity Locking

1    lockVar pageLocks[fileLength];
2    int lockedPages[6];
3        ...
4
5    for (j = 0; j < 5; j++) {
6     lockedPages[j]=4096*(rand()%fileLength);
7     lock(pageLocks[lockedPages[j]]);
8     sum += x[lockedPages[j]];
9    }
10
11   lockedPages[5]=4096*(rand()%fileLength);
12   lock(pageLocks[lockedPages[5]]);
13   x[lockedPages[5]] = sum;
14
15   for (j = 0; j < 6; j++) {
16    unlock(pageLocks[lockedPages[j]]);
17   }
```

E

Figure 1-5: Two different locking protocols for the program in Figure 1-2. Program D acquires a global lock, while Program E acquires a different lock for every page. For simplicity, only the body of the code is shown here.

Deadlock can be avoided in Program E if we first precompute the 6 random pages that will be accessed by the program and then acquire the locks in order of increasing page number. This approach does not work in a more complicated program where the next page to access depends on the data stored in the current page (for example, if we are following pointers down a linked list). Other deadlock-free solutions for this problem exist, but all require additional code that is even more complicated than Program E.

The example in Figure 1-5 demonstrates two alternatives for programming with locks. We can implement a simple but inefficient locking scheme that is clearly correct, or we can implement a complex but more efficient locking scheme that is more difficult to program correctly. Furthermore, we are allowed to pick only one of these alternatives. If we run Program D on one process and run Program E concurrently on another process, then then we have a data race because both programs have not agreed upon the same locking protocol. Locking protocols are often implementation-specific, breaking natural program abstractions and modularity.

As a final sad end to this story, imagine that somewhere in the midst of a large piece of software, we forget to add locks to one copy of Program C. Discovering this error through testing and simulation becomes more difficult as the file size grows. The probability that both programs access the same pages and that the operations interleave in just the right (or perhaps, wrong) way to cause an error is quite small. Data races are not just a theoretical problem: these programming errors can have real-world consequences. In August of 2003, a race condition buried in 4 million lines of C code helped cause the worst power blackout in North American history [30]. A correct locking protocol is useless if the programmer forgets to use it.

Parallel Programming with Memory-Mapped Transactions

Programming with memory-mapped transactions is more convenient than programming with locks. In this section, I present a new version of the program from Figure 1-2 written with memory-mapped transactions that is as simple to code as Program D, but still admits concurrency when the critical sections of code are independent as in Program E.

A transaction, as described in [18, 20], is a fundamental abstraction that is used extensively in database systems. Conceptually, a transaction is a section of code that appears to either execute successfully (i.e., it *commits*), or not execute at all (i.e., it *aborts*). For databases, transaction systems typically guarantee the so-called ACID properties for transactions: atomicity, consistency, isolation, and durability [20]. These properties guarantee that two committed transactions never appear as though their executions were interleaved. Thus, data races can be eliminated by embedding the relevant critical sections of code inside transactions.

Programming with transactions has traditionally been limited to database systems. With Herlihy and Moss's proposal for *transactional memory* in [24], however, many researchers (including [1, 14, 21, 22, 23, 25, 42]) have begun to focus on transactions as a viable programming paradigm in more a general context.[1] The term

[1] I doubt I have come anywhere close to citing all the relevant papers on transactional memory, especially since new papers are appearing on a regular basis. My apologies to anyone I have missed.

```
1     while(1) {
2       xbegin();
3
4       for (j = 0; j < 5; j++) {
5          i = rand() % fileLength;
6          sum += x[1024*i];
7       }
8
9       i = rand() % fileLength;
10      x[1024*i] = sum;
11
12      if (xend() == COMMITTED) break;
13      backoff();
14    }
```

Figure 1-6: Program F. This version of the program from Figure 1-2 is written with memory-mapped transactions.

transactional memory is used by Herlihy and Moss in [24] to describe a hardware mechanism, built using existing cache-coherency protocols, that guarantees that a set of transactional load and store operations executes atomically. More generally, others have used the term to refer to any system, hardware or software, that provides a construct that facilitates programming with transactions.

I describe an interface for programming with transactions in C modeled after the interface described in [1]. The authors of [1] describe a hardware scheme that provides two machine instructions, `xbegin` and `xend`. All instructions of a thread that execute between an `xbegin`/`xend` pair represent a single *transaction* that is guaranteed to execute atomically with respect to all other transactions. I have implemented LIBXAC, a C library supporting transactions in which `xbegin` and `xend` are function calls.

Figure 1-6 illustrates the body of a simple program using a memory-mapped transaction. Program F is another version of our favorite example from Figure 1-2, a version converted to use transactions.[2] Superficially, Program F is almost identical to Program D. The only differences are that we have replaced the acquire/release locking operations with `xbegin` and `xend`, and we have enclosed the entire transaction in

[2]For simplicity, I show only the body of the transaction here. Chapter 1.4 presents the complete interface.

a while loop to retry the transaction in case of an abort. If the transaction aborts, any modifications that the transaction made to `x` are automatically rolled back by the underlying transaction system. Therefore, when the `xend` function completes, the transaction either appears to have atomically modified `x`, or it appears as though no changes to `x` were made at all. Program F still has the advantage of D, that writing a race-free parallel program is relatively easy.

Although Programs D and F appear quite similar, their behavior during execution is quite different. When two copies of Program F run concurrently on different processes, the two processes can modify `x` concurrently, while in Program D, the updates to the array must occur serially. The transactions in Program F are only aborted in the unlikely event that the the two transactions conflict with each other. Thus, Program F can be just as efficient as with page-granularity locking in Program E, but as simple to code as a program with a global lock in Program D.

If two processes each run copies of Program F, there is no problem with deadlock as with Program E. Since one transaction aborts if two transactions conflict with each other, it is impossible to have the two processes each waiting on the other. With transactions, there may be a possibility for *livelock*, i.e., when two or more transactions never succeed because they keep aborting each other. This situation, I argue, is much easier for the programmer to avoid than deadlock. The programmer is better equipped to deal with a transaction abort than a deadlock because the programmer already codes a transaction taking into account the possibility that the transaction may not succeed. In contrast, without some external mechanism for detecting and resolving deadlocks, there is little the programmer can do in the code that is waiting to acquire a lock.

To avoid livelock, the programmer can also implement a backoff function, as in Line 13 of Program F. This function specifies how long a transaction waits before retrying after an abort. If the programmer chooses an appropriate backoff strategy, then it is likely that the transactions will complete in a reasonable amount of time. One key property is that each transaction can have a different backoff function, and the code will still run correctly. Transactions with different backoff strategies do not

have the same incompatibility problem that programs using different locking protocols have.

Finally, the underlying transaction system may specify a policy for aborting transactions that guarantees that a transaction always eventually succeeds, thereby eliminating the problem of livelock altogether. For example, the transaction system could guarantee that whenever two transactions conflict, that the one that began first always succeeds. This mechanism guaranteers that a transaction always succeeds eventually, as the oldest transaction in the system never gets aborted. The underlying transaction system may also implement other policies for resolving transaction conflicts. The effect of using different policies for the application programmer is minimal, however, in the common case where transactions rarely conflict with each other.

1.3 Concurrent Disk-Based Programs

In this section, I characterize the space of programs that concurrently manipulate data on disk and describe where memory-mapped transactions belong in this space.

Throughout this chapter, I have presented multiple versions of the same hypothetical program that sums the first integer from each of 5 random pages of a file and stores the result into the first integer on a 6th random page. Figure 1-7 illustrates where each of these programs belongs in the space of possible programs that concurrently access disk.

The horizontal axis depicts how the program accesses disk. Programs that access disk using explicit I/O operations are on the left, programs that do not access disk at all are in the middle, and programs that use memory mapping to access disk are on the right. Moving right along this axis corresponds to a higher level of abstraction. In Programs A and B, the programmer codes explicit I/O operations, while in Program C the distinction between main memory and disk is almost completely abstracted away by the use of memory mapping.

Similarly, the vertical axis depicts how the programs handle concurrency. Programs that use transactions are at the top, programs that run code serially are in

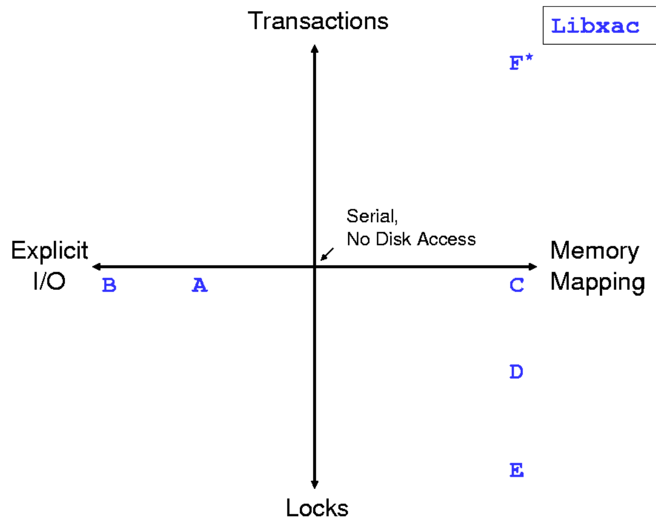


Figure 1-7: An illustration of the possible space of programs that can concurrently access data on disk.

the middle, and programs that use explicit locking are on the bottom. Moving up corresponds to a higher levels of abstraction: with transactions, the programmer can write parallel programs without worrying about the details of concurrency control that a program with locks must deal with. Programs D and E both use memory mapping to access disk, but they use locking to avoid data races.

Programs that use LIBXAC, a library for supporting memory-mapped transactions, fit into the upper right corner of this diagram. This corner represents the most convenient place for the application programmer: programmers can write code with multiple processes that simultaneously access disk without worrying about explicit locking or I/O operations.

Programs in the three other corners of this diagram exist as well. The lower left corner, programs with explicit I/O and locking, is where the programmer has the most control over the details of the code. For a simple application, a programmer may be able to create an efficient, optimized program. The difficulty of doing such optimizations rapidly increases, however, as the application gets more complex. Program E is an example of code that belongs in the lower right corner; the programmer does not worry about moving data between disk and main memory, but does handle concur-

rency control. Finally, a program in the upper left corner works uses transactions, but explicitly manages accesses to disk. In this case, the transaction system may provide explicit transaction read and write operations from disk, and the transaction system takes care of rolling back transactional writes to disk on an a transactional abort.

1.4 Thesis Overview

In this section, I describe the primary design goals of usability and portability for Libxac, a C library for supporting memory-mapped transactions. Finally, I conclude this chapter with an outline of the rest of this thesis.

Design Goals for Libxac

I designed Libxac with two primary goals in mind:

1. Libxac should provide an interface for concurrently accessing data on disk that is simple and easy to use, but still expressive enough to provide reasonable functionality. Programming with memory-mapped transactions, I argue in this thesis, satisfies this requirement of usability.

2. Libxac should be portable to a variety of systems. For this reason, Libxac is written as a user-level C library and tested on a Linux operating system. Libxac relies only on generic memory mapping and signal-handling routines, not on the special features of research operating systems or special hardware.

Outline

In this thesis, I argue that memory-mapped transactions provide a convenient interface for concurrent and persistent programming. I also present evidence that suggests that it is possible to support a memory-mapped transactional interface portably and efficiently in practice.

Programming Interface

First, I demonstrate that an interface based on memory-mapped transactions has a well-defined and usable specification by describing the specification for LIBXAC, a C library supporting memory-mapped transactions. LIBXAC provides an `xMmap` function that allows programmers to memory-map a file transactionally. Programmers can then easily specify transactions that access this file by enclosing the relevant code between two function calls, `xbegin` and `xend`. Programming with memory-mapped transactions is easy because the runtime automatically detects which pages in memory a transaction accesses.

LIBXAC's memory model guarantees that transactions are "serializable," and that aborted transactions always see a consistent view of the memory-mapped file. This property leads to more predictable program behavior and in principle allows read-only transactions to always succeed. Programs that use LIBXAC, I argue, are more modular, and thus easier to write, debug, and maintain than code that with explicit I/O or locking operations.

The specification for the interface and the memory model both simplify the writing of programs that have multiple processes that concurrently access disk. With LIBXAC, I was able to easily convert existing serial memory-mapped implementations of a B^+-tree and a cache-oblivious tree (CO B-tree) into a parallel version supporting concurrent searches and insertions. This conversion took little time and required few changes to the code, demonstrating the ease of using memory-mapped transactions to code a concurrent, disk-based data structure.

In Chapter 1.4, I describe the LIBXAC specification and memory model. In Chapter 5, I describe the conversion of the serial search tree implementations into parallel versions.

Implementation

Next, I demonstrate that it is possible to implement a memory-mapped transaction system by describing a prototype implementation of the LIBXAC specification on

Linux. The implementation itself uses memory mapping, thereby using Linux's virtual memory subsystem to buffer pages from disk in main memory. The Libxac prototype also supports durable transactions on a single process by logging enough information on disk to restore a memory-mapped file to a consistent state after a program crash. The prototype does have several drawbacks; for example, it uses a centralized control mechanism that limits concurrency, and it does not include the routine for recovery yet. I argue that these drawbacks can be overcome, however, and that this prototype shows that providing support for memory-mapped transactions is feasible in practice.

Although the prototype has many shortcomings, it does have the advantage of being portable. The prototype relies primarily on the memory-mapping function `mmap` and the ability to specify a user-level handler for the `SIGSEGV` signal to support nondurable transactions. This implementation is more portable than transaction systems that rely on special features of research operating systems [8, 11, 12, 19, 46].

In Chapter 3, I describe these details of the Libxac implementation.

Experimental Results

The last step is to determine whether memory-mapped transactions can be supported efficiently in practice. I describe results from several experiments designed to measure the prototype's performance on both small nondurable and durable transactions that fit into main memory. A durable transaction incurs additional overhead compared to a nondurable transaction because the runtime logs enough information to be able to restore the memory-mapped file to a consistent state in case the program crashes.

I first used the experimental data to construct a performance model for memory-mapped transactions. For a small nondurable transaction that reads from R pages and writes to W pages, this model estimates that the additive overhead for executing the transaction is roughly of the form $aR + bW$, where a is between 15 to 55 microseconds and $b \approx 2a$. In some cases, at least 50% of this runtime overhead is spent entering and exiting fault handlers and calling `mmap`. The majority of the remaining time appears to be spent handling cache misses and page faults. The performance model for a small durable transaction, is roughly $aR + bW + c$, where a is tens of microseconds, b

is hundreds to a few thousand microseconds, and c is between 5 and 15 milliseconds. The single-most expensive operation for a durable transaction is the time required to synchronously write data out to disk on a transaction commit.

I then ran experiments to estimate the potential concurrency of independent transactions using Libxac. The results suggest that for a simple program that executes independent, nondurable transactions on a multiprocessor machine, when the work each transaction does per page is two orders of magnitude more than the per-page runtime overhead, the program achieves near-linear speedup on two processes compared to one process. The synchronous disk write required for each transaction commit appears to be a serial bottleneck that precludes any noticeable speedup when running independent, durable transactions on multiple processes, however. An implementation that is modified to work with multiple disks might admit more concurrency for durable transactions.

Finally, I measured the time required to insert 250,000 elements with randomly chosen keys into a Libxac search tree. The Libxac B^+-tree and CO B-tree were both competitive with Berkeley DB, when a single process performed each insertion as a durable transaction. On modern machines, the performance of the Libxac B^+-tree and CO B-tree ranged from being 4% slower to 67 % faster than insertions done using Berkeley DB. The fact that a program using the unoptimized Libxac prototype actually runs faster than a corresponding program using a high-quality transaction system such as Berkeley DB is quite surprising. This promising result suggests that it is possible for memory-mapped transactions to both provide a convenient programming interface and still achieve good performance in an actual practical application.

I describe the construction of the performance model and the experiments for estimating the concurrency of independent transactions in Chapter 4. I describe the experiments on Libxac search trees in Chapter 5.

Future Work

My treatment of memory-mapped transactions in this thesis is certainly not comprehensive. I conclude this thesis in Chapter 6 by describing possible improvements

to the implementation and possible directions for future work. In particular, I discuss the possibility of using a transaction system such as LIBXAC to help support unbounded transactional memory [1]. One weakness of the prototype is the overhead required to execute nondurable transactions. By combining a hardware transactional memory mechanism with a software transactional memory implementation such as LIBXAC, it may possible to provide an interface for programming with nondurable transactions that is both efficient and easy to use.

In summary, in this thesis, I argue that memory-mapped transactions simplify the writing of programs with multiple processes that access disk. I then present evidence that suggests that a memory-mapped transaction system can be efficiently supported in practice.

Chapter 2

The Libxac Interface

In this chapter, I present the specification for LIBXAC, a C library supporting memory-mapped transactions. LIBXAC demonstrates that a programming interface based on memory-mapped transactions can have a well-defined and usable specification.

In Section 2.1, I illustrate how to write programs with memory-mapped transactions. I present the prototypes for LIBXAC's basic functions for nondurable transactions and exhibit their use in a complete program. LIBXAC's interface, modeled after ordinary memory mapping, provides an `xMmap` function that allows a programmer to memory-map a file transactionally. A programmer can easily specify a transaction by enclosing the relevant code in between `xbegin` and `xend` function calls, and the runtime automatically detects which pages a transaction accesses. Section 2.1 also illustrates how LIBXAC supports nested transactions by subsuming inner transactions into the outermost transaction, and describes additional functions for durable transactions.

In Section 2.2, I show that memory-mapped transactions can have well-defined semantics by describing LIBXAC's memory model. This model guarantees that both committed and aborted transaction instances are "serializable," and that aborted transactions always see a consistent view of the memory-mapped file. Transactions abort synchronously at the `xend` call, and only changes to the `xMmap`ed file are rolled back on an abort; any changes that the transaction makes to local variables remain. These restrictions on the behavior of aborted transactions, I argue, lead to more

predictable program behavior and thus simpler programs.

Because memory-mapped transactions provide a simple interface, opportunities for additional program optimizations exist. I discuss three functions for optimizing Libxac programs in Section 2.3. First, Libxac provides a function for explicitly validating a transaction in the middle of execution. A program can prematurely abort a transaction if this function reports that the runtime has already detected a conflict. Second, Libxac uses a multiversion concurrency control scheme to provide special functions for specifying read-only transactions that never generate transaction conflicts. Finally, Libxac provides an advisory function that reduces the overhead of automatic detection of pages accessed by a transaction.

I explain how memory-mapped transactions fit in the context of other work in Section 2.4. I briefly describe other systems that provide mechanisms for simplifying concurrent and/or persistent programming, focusing on three areas: transaction systems for databases, persistent storage systems, and transactional memory.

In Section 2.5, I conclude with a summary of the main advantages of an interface based on memory-mapped transactions. Programs and data structures written using memory-mapped transactions are modular because they separate the concurrency structure of the program from the specific implementation. Because memory-mapped transactions hide details such as I/O operations and locking, programmers can easily code complex but algorithmically efficient data structures. Finally, an interface based on memory-mapped transactions is flexible because it can provide features such as multiversion concurrency control and support for durable transactions.

2.1 Programming with Libxac

In this section, I illustrate how to write programs with memory-mapped transactions using Libxac. First, I present the prototypes for Libxac's basic functions for non-durable transactions and demonstrate their use in two complete sample programs. Next, I describe how Libxac's supports nested subsumed transactions with another sample program. Finally, I describe Libxac's functions for supporting durable trans-

`int xInit(const char *path, int flags);`	This function initializes Libxac. The `path` argument specifies where Libxac stores its log and control files. The flag specifies the kind of transaction to support (either `NONDURABLE` or `DURABLE`).
`int xShutdown(void);`	This function shuts down Libxac. This function should be called only after finishing all transactions on all processes.
`void* xMmap(const char *name, size_t length);`	The `xMmap` function memory-maps the first `length` bytes of the specified file transactionally. Length must be a multiple of the system page size. The function returns a pointer to the transactionally-mapped file, or `MAP_FAILED` on an error.
`int xMunmap(const char *name);`	The `xMunmap` unmaps the specified file.
`int xbegin(void);`	The `xbegin` function marks the beginning of a transaction.
`int xend(void);`	The `xend` function marks the end of a transaction. Returns `COMMITTED` (`ABORTED`) if the transaction completed successfully (unsuccessfully). For a nested transaction, `xend` returns `PENDING` if no conflict has been detected, and `FAILED` otherwise.

Table 2.1: The Libxac functions for nondurable transactions.

actions and some restrictions to the Libxac interface.

The Libxac Specification

Table 2.1 gives the prototypes for Libxac's basic functions. All functions except `xMmap` and `xend` return 0 if they complete successfully and a nonzero error code otherwise.

```
1      int fileLength = 10;
2      int main(void) {
3        int* x;
4
5        xInit(".", NONDURABLE);
6        x = (int*)xMmap("input.db", 4096*fileLength);
7
8        while (1) {
9          xbegin();
10         x[0]++;
11         if (xend() == COMMITTED) break;
12       }
13
14       xMunmap("input.db");
15       xShutdown();
16       return 0;
17     }
```

Figure 2-1: LIBXAC program that increments the first integer of a memory-mapped file.

A Simple Libxac Program

Figure 2-1 illustrates a simple program using LIBXAC that increments the first integer stored in the file `input.db`.

Line 5 calls `xInit` to initialize LIBXAC. The second argument is a flag specifying whether transactions should be durable or nondurable. For durable transactions, LIBXAC writes enough information to disk to guarantee that the data can be restored to a consistent state, even if the program crashes during execution.

Line 6 calls `xMmap` to transactionally memory-map the first 10 pages of the file `input.db`. This function returns a pointer to a *shared-memory segment* that corresponds to the appropriate pages in the shared file. The second argument to `xMmap` must be a multiple of the system's page size. The function prototype for `xMmap` is effectively a version of the normal `mmap` with fewer arguments.[1]

Lines 8–12 contain the actual transaction, delimited by `xbegin` and `xend` function

[1]Memory protections and sharing are handled by LIBXAC, eliminating the need for those extra arguments. The `xMmap` function does not use an offset argument because the prototype currently allows only mappings that start at the beginning of the file. A more general specification for `xMmap` would behave more like `mmap`, handling multiple shared-memory segments and mappings of only parts of a file.

calls. After calling `xMmap`, programs may access the shared-memory segment inside a transaction (Line 10).[2] The transaction appears to either execute atomically or not at all. The `xend` function returns `COMMITTED` if the transaction completes successfully, and `ABORTED` otherwise.

A transaction may abort because of a conflict with another concurrent transaction. This program encloses the transaction in a simple loop that immediately retries the transaction until it succeeds, but in a real application, the programmer may want to specify some algorithm for backoff (i.e, waiting) between transaction retries to reduce contention.

LIBXAC's memory model guarantees that transactions are *serializable* with respect to the shared-memory segment. In other words, when the program executes, there exists a serial order for all committed transactions such that execution is consistent with that ordering. For example, if two copies of the program in Figure 2-1 run concurrently, the execution always appears as if one transaction happens completely before the other. In particular, the interleaving shown in Figure 1-4 can never occur. As I describe later in Section 2.2, LIBXAC actually makes a stronger guarantee, that aborted transactions see a consistent view of the shared-memory segment as well.

Line 14 calls `xMunmap`, the transactional analog to `munmap`. This function should not be called by a process until all transactions on that process have completed. Line 15 calls `xShutdown` to shuts down LIBXAC, guaranteeing that all changes made to files that have been `xMmap`ed have been stored on disk. After `xShutdown` completes, it is safe to modify those files via normal means, such as `mmap` or `write`.

Programs with Complex Transactions

LIBXAC's interface is easy to use because specifying a block of code as a transaction is independent of that code's complexity. Even a long and complicated transaction in between `xbegin` and `xend` still appears to execute atomically. For example, recall the program using memory-mapped transactions that reads the first 4-byte integer from

[2]Attempting to access the shared-memory segment outside a transaction results in unspecified program behavior (usually a fatal program error).

```
// Serial version using mmap.

1  int fileLength = 100000;
2  int main(void) {
3    int i, j, sum = 0;
4    int fd;
5    int* x;
6    fd = open("input.db",
7              O_RDWR,
8              0666);
9    x = (int*)mmap(x,
10                  4096*fileLength,
11                  PROT_READ|PROT_WRITE,
12                  MAP_SHARED,
13                  fd,
14                  0);
15
16
17   for (j = 0; j < 5; j++) {
18      i = rand() % fileLength;
19      sum += x[1024*i];
20   }
21
22   i = rand() % fileLength;
23   x[1024*i] = sum;
24
25
26
27   munmap(x, fileLength);
28   close(fd);
29   return 0;
30 }
```

```
// Concurrent version using Libxac
//  and xMmap

1  int fileLength = 100000;
2  int main(void) {
3    int i, j, sum = 0;
4    int* x;
5
6    xInit(".", NONDURABLE);
7
8
9    x = (int*)xMmap("input.db",
10                   4096*fileLength);
11
12
13
14
15   while (1) {
16     xbegin();
17     for (j = 0; j < 5; j++) {
18        i = rand() % fileLength;
19        sum += x[1024*i];
20     }
21
22     i = rand() % fileLength;
23     x[1024*i] = sum;
24     if (xend()==COMMITTED) break;
25   }
26
27   xMunmap("input.db");
28   xShutdown();
29   return 0;
30 }
```

Figure 2-2: A side-by-side comparison of the program in Figure 1-2 and the parallel version written with LIBXAC.

each of 5 randomly selected pages in a file and stores their sum on a randomly selected 6th page (see Figure 1-6). Figure 2-2 compares the original serial memory-mapped version of this program to the complete LIBXAC version. The only significant changes in the transactional version are the addition of `xbegin` and `xend` calls and the while loop to retry aborted transactions.

LIBXAC also supports simple nested transactions by subsumption, i.e., nested inner transactions are considered part of the outermost transaction. This feature is necessary for transactions that involve recursion. Consider the method in Figure 2-3 that recursively walks down a tree and computes the sum of all the nodes in the

```
1      int sum(tree* t) {
2         int answer = 0;
3         while (1) {
4           xbegin();
5            if (t == NULL) answer = 0;
6            else {
7              answer = t->value + sum(t->left) + sum(t->right);
8            }
9           if (xend() != ABORTED) break;
10        }
11        return answer;
12    }
```

Figure 2-3: A recursive function that uses nested transactions.

tree. An `xend` call nested inside another transaction returns `FAILED` if the runtime detects that the outermost transaction has encountered a conflict and will abort, and `PENDING` otherwise.[3] In a recursive function, the programmer should only retry the transaction if the status returned is `ABORTED`, i.e., when the outermost transaction reaches its `xend`. LIBXAC must support at least subsumed nested transactions if it allows a transaction to call a subroutine that contains another transaction. This feature is desirable for program modularity: a transaction should not care whether its subroutines are themselves implemented with transactions.

Interface for Durable Transactions

Programmers using LIBXAC can choose for all transactions to be durable by calling the `xInit` function with the `DURABLE` flag. When a durable transaction commits, LIBXAC forces enough data and meta-data out to a log file on disk to ensure that the changes made by committed transactions are not lost if the program or system crashes.[4] There are three library functions specific to durable transactions:

1. `xRecover`: If the program crashes, then calling `xRecover` on the `xMmap`ed file

[3]A side note: the program in Figure 2-3, cannot return `answer` from inside the transaction because control flow will skip the `xend` function call. In general, every `xbegin` call must be paired with a corresponding `xend`.

[4]This guarantee assumes that the hardware (i.e., the disk) has not failed.

restores the file to a consistent state. After recovery has been run, LIBXAC guarantees that all changes made by committed transactions have been restored, in the same order as before. Recovery is done by scanning the log files and copying the changes made by committed transactions back into the original file.

2. `xCheckPoint`: Checkpointing reduces the number of transactions that have to be repeated during recovery by forcing the runtime to copy changes made by committed transactions into the original file.[5]

3. `xArchive`: A log file is no longer needed for recovery once all the changes recorded in that log file have been copied back to the original file. This function identifies all such log files that are safe to delete.

For both nondurable and durable transactions, `xShutdown` automatically executes a `xCheckPoint` operation, ensuring that the original file contains consistent data after LIBXAC has been shut down. Similarly, the specification of the `xInit` requires that LIBXAC verify the integrity of the file and run `xRecover` if necessary. The description of these functions for durable transactions, completes the specification for all the basic functions that a fully functional version of LIBXAC provides.[6]

Restrictions on Libxac

The implementation inevitably imposes some restrictions on the LIBXAC interface. The most significant one is that programs using LIBXAC can have only one transaction per process. Because LIBXAC supports concurrency control between processes, not threads, transactions on one thread should not run concurrently with conflicting code

[5]Because of the multiversion concurrency control, after a checkpoint completes, it is **not** true that the data from every committed transaction has been copied back into the file. A transaction that is still running may need to access older values stored in the original file. The checkpoint operation copies as much data as possible, however. If no other transactions are being executed, then all changes is copied into the original file.

[6]As a caveat, although I have devised a specification for the recovery, checkpoint, and archive functions, in the LIBXAC prototype, in the prototype, I have not actually implemented the recovery or archive routines, and I have only implemented the implicit `xCheckpoint` in `xShutdown`. See Chapter 3 for implementation details.

on other threads in the same process. This restriction is difficult to remove because Linux supports memory protections at a per-process level, not a per-thread level.

Another restriction is that every `xbegin` function call must be properly paired with an `xend`. In other words, the control flow of a program should never jump out of a transaction without executing `xend`. See Appendix A for a more detailed discussion of restrictions to Libxac.

2.2 Semantics of Aborted Transactions

In this section, I argue that memory-mapped transactions can have well-defined semantics by describing Libxac's memory model. Libxac guarantees that both committed and aborted transactions are "serializable," and that aborted transactions always see a consistent view of the memory-mapped file. Libxac specifies that transactions abort synchronously upon reaching the `xend` function, and that on an abort, only changes to the shared-memory segment are rolled back, and any changes that the transaction makes to local variables remain. I argue that these restrictions on the behavior of aborted transactions lead to more predictable program behavior.

Libxac guarantees that transactions on the shared-memory segment appear to happen atomically or not at all. The committed transactions are *serializable*, meaning there exists a total order of all transactions such that the system appears to have executed transactions in that order.[7] This definition of serializability is intuitive and fairly straightforward. For a more formal, textbook treatment of serializability theory, in both the single-version and multiversion contexts, see [6].

The behavior of a transaction that commits is straightforward because a transaction completes successfully in only one way. Aborted transactions, however, can be handled in multiple ways. In this section, I discuss several design decisions for aborted transactions.

[7] In this situation, I use the term transaction to refer to a particular transaction instance, i.e., the instructions that execute between an `xbegin` and an `xend`. When a transaction is aborted and retried, it counts as a different transaction instance.

Asynchronous vs. Synchronous Aborts

Once the runtime detects a conflict and decides to abort a transaction, it can either abort the transaction immediately i.e., *asynchronously*, or it can continue to execute the transaction until it reaches a specified point where it can be safely aborted *synchronously*. This specified point can be in the middle of a transaction, or it can simply be the `xend` function call. A transaction that will abort but has not reached a specified point is said to have `FAILED`.

Libxac synchronously aborts a transaction once the `xend` function is reached because asynchronous aborts are more difficult to implement cleanly and portably.[8] Unfortunately, performing synchronous aborts may be inefficient for a program with many levels of nested transactions. If the outermost transaction aborts, the runtime must still return from each nested transaction. An asynchronous abort might allow the program to jump immediately to the outermost transaction.

Consistent vs. Inconsistent Execution

A system that performs synchronous aborts may specify what kinds of values an `FAILED` transaction can see in the shared-memory segment. A transaction's execution is *consistent* if a `FAILED` transaction never sees intermediate data from other transactions. A system that guarantees consistent execution typically requires a multiversion concurrency control protocol.

If a system performs synchronous aborts but does not guarantee a consistent execution, then a transaction may enter an infinite loop or cause a fatal error because it read a corrupted value in the shared memory segment. In this case, the runtime must be capable of handling these exceptional cases.

Libxac supports consistent execution, ensuring that aborted transactions always see a consistent view of the shared-memory segment. In other words, if one considers only the operations each transaction does on the shared-memory segment, then all transactions, whether committed or aborted, are serializable. If a transaction aborts,

[8] I have not fully explored using `setjmp` and `longjmp` to do asynchronous aborts in Libxac.

its changes to the shared-memory segment are discarded. If the transaction commits, however, its changes are made visible to transactions that come later in the order.

In LIBXAC, the combination of synchronous aborts and consistent execution has an interesting implication because the point of abort is at the `xend` function call. Since all transactions see a consistent view of the shared-memory segment, read-only transactions, in principle, can always succeed.

Transactional vs. Nontransactional Operations

LIBXAC only enforces transactional semantics for memory operations that access the shared-memory segment. We refer to these as *transactional operations*, while other operations that access process-local variables or other memory are *nontransactional operations*.[9] By default, it is unclear how these two types of operations should interact with each other. For example, suppose that the two programs in Figure 2-4 run concurrently. Program 1 modifies local variables `b`, `y`, and `z` inside the transaction. If the initial value of `x[0]` is `42`, what are the possible final values for `a, b, y` and `z`?

The answer depends on how the system deals with local variables. If the runtime does a *complete rollback*, then all nontransactional operations get rolled back to their original value, before the transaction started executing. With this approach, variables `b`, `y`, and `z` are always rolled back to 0 on a transaction abort. Therefore, after the transaction in Program 1 commits, `y` is 1 and `a` is the number of times the transaction tried to execute. Both `b` and `z` will be `41` if the transaction in Program 2 executes first, and both will be `42` otherwise. Unfortunately, without additional compiler support for detecting local variables and backing up their original values, it seems difficult to support complete rollback with only a runtime library. Semantically, complete rollback works equally well with synchronous or asynchronous aborts and with consistent or inconsistent execution, provided that it is possible to rollback all nontransactional operations. Nontransactional operations such as `printf` may be

[9]Note that this definition is based on the memory location, not whether the operation happened in a transaction. The LIBXAC prototype disallows transactional operations outside of a transaction, but it does specify the behavior of some nontransactional operations inside a transaction.

```
// Program 1

1   int main(void) {
2     int y = 0, z = 0; a = 0, b = 0;
3     int* x;
4     xInit(".", NONDURABLE);
5     x = xMmap("input.db", 4096);
6
7     while (1) {
8       a++;
9       xbegin();
10      b += x[0];
11      y++;
12      x[0]++;
13      z += (x[0] - 1);
14      if (xend()==COMMITTED) break;
15    }
16
17    munmap(x, 4096);
18    xShutdown();
19    return 0;
20  }
```

```
// Program 2

1   int main(void) {
2
3     int* x;
4     xInit(".", NONDURABLE);
5     x = xMmap("input.db", 4096);
6
7     while (1) {
8
9       xbegin();
10
11
12       x[0]--;
13
14      if (xend()==COMMITTED) break;
15    }
16
17    munmap(x, 4096);
18    xShutdown();
19    return 0;
20  }
```

Figure 2-4: A transaction that that accesses local variables inside a transaction.

impossible to roll back however, if the output has already been sent to the user's terminal.

Alternatively, the system can roll back only transactional operations. More specifically, LIBXAC, reverses changes to the shared-memory segment, but not to local variables. There are several cases to consider:

1. If the runtime does not guarantee consistent execution of transactions, then arbitrary values may get stored into `b` and `z`.

2. If the runtime performs asynchronous aborts and guarantees consistent execution, after Program 1 completes, the variable `a` stores the number of times the transaction was attempted, while `y` stores the number of times the transaction made it past the increment of `y` before aborting or committing. Similarly, `b` and `z` may have different values, depending on how often and when the transaction was aborted.

3. With synchronous aborts and consistent execution, after Program 1 completes,

`a` and `y` will always both equal the number of different transaction instances executed on process 1. Also, `b` and `z` will always have the same value (`41` if the transaction in program 2 completes first, and `42` otherwise).

LIBXAC satisfies Case 3, the case that most cleanly specifies the behavior of nontransactional operations inside a transaction. The example program demonstrates that Case 3 leads to the most predictable behavior for aborted transactions. Conceptually, an aborted transaction is similar to a committed transaction. First, a transaction modifies its own local copy of the shared-memory segment. After the `xend` completes, these changes atomically replace the actual values in the shared-memory segment only if the transaction commits.

One final method for handling nontransactional operations is to simply ignore them, leaving their behavior completely unspecified. This option is undesirable, however, as the program in Figure 2-4 demonstrates that it is possible to have well-defined semantics for some nontransactional operations inside a transaction. nontransactional operations provide the programmer with a loophole to strict serializability. For example, the programmer can use local variables to log what happens in aborted transaction instances. Obviously, such a loophole should be used cautiously.

Related Work

Serializability theory is discussed for both single-version and multiversion concurrency control in [5]. These concepts are also described in [6]. Many researchers have proposed other correctness criteria for concurrent systems. One such definition is the concept of *linearizability* for concurrent objects, proposed by Herlihy and Wing in [27]. In this model, each object has a set of operations it can perform. To perform an operation, an object makes a request, and later it receives a response. Every response matches a particular request. If the objects are transaction instances (committed or aborted), then the request and response are the beginning and end of the transaction, respectively. Linearizability guarantees that each transaction appears as though the execution happened instantaneously between the request and response. The LIBXAC

memory model can be thought of as a particular case of linearizability.

2.3 Optimizations for Libxac

The simplicity of an interface based on memory-mapped transactions creates opportunities for additional program optimizations. This section discusses three functions for optimizing LIBXAC programs. The `xValidate` function explicitly validates a transaction in the middle of execution, facilitating quicker detection of transaction conflicts. The `xbeginQuery` and `xendQuery` functions specify a read-only transaction that never generates transaction conflicts. Finally, the advisory function, `setPageAccess` informs the runtime that a transaction wishes to access a certain page, reducing the overhead of automatically detecting accesses to that page.

Transaction Validation

Since LIBXAC synchronously aborts transactions, a transaction that fails because of a conflict keeps running until reaching `xend`. Continuing to run a long transaction that has already failed is inefficient, however. To avoid unnecessary work, a program can periodically call `xValidate` inside a transaction to check if it has failed. This function returns `FAILED` if the runtime has detected a conflict and will abort the transaction, and returns `PENDING` otherwise. Based on the result, the program can use `goto` to jump to a user-specified label at `xend` to abort the transaction.[10]

Read-Only Transactions

Read-only transactions in LIBXAC can, in principle, always succeed because transactions always see a consistent view of the shared-memory segment. LIBXAC assumes that all transactions both read and write to the segment, however. Thus, `xend` may return `ABORTED` even for a read-only transaction that could have safely committed. A

[10]The `xValidate` function has a one-sided error: if it returns `FAILED`, then there is a conflict, but when it returns `PENDING`, there can still be a conflict. This specification allows the runtime to simply query a status flag, instead of actively checking for new conflicts.

programmer that knows a transaction is read-only could safely ignore the return value, but this approach is susceptible to error. Instead, LIBXAC provides the `xbeginQuery` and `xendQuery` functions for explicitly specifying a read-only transaction.

As a replacement for `xbegin` and `xend`, `xbeginQuery` and `xendQuery` provide three advantages. First, LIBXAC performs slightly less bookkeeping for read-only transactions because they always succeed. Second, even if LIBXAC is in durable-transaction mode, the runtime does not need to force data out to disk when a read-only transaction commits. Finally, LIBXAC can report an error if a program writes to the shared-memory segment inside a read-only transaction (by immediately halting the program, for example). Note that it is legal to nest a read-only transaction inside a normal transaction, but not a normal transaction inside a read-only transaction.

Advisory Function

A programmer can reduce the overhead incurred when a transaction accesses a page in the shared-memory segment for the first time by calling the *advisory function*, `setPageAccess`. Without the advisory function, LIBXAC automatically detects a page access by handling a `SIGSEGV`. A programmer can use the advisory function to inform LIBXAC that the current transaction plans to access a specified page with a specified access permission (read or read/write), thereby avoiding the `SIGSEGV`.

Using the advisory function affects only the performance of a program, not its correctness. If the programmer uses the advisory function on the wrong page, then this only hurts concurrency by generating a possibly false conflict. On the other hand, if the programmer forgets to call the advisory function on a page, the access will be caught by the default mechanism.

2.4 Related Work

In this section, I discuss memory-mapped transactions in the context of related work on mechanisms for simplifying concurrent and/or persistent programming. I focus

primarily on three areas: transaction systems for databases, persistent storage systems, and transactional memory.

The idea of virtual memory and memory-mapping is decades old. For example, in the 1960's, Atlas [31] implemented paged virtual memory and Multics [37] implemented a single-level store. Appel and Li in [2] survey many applications of memory-mapping, including the implementation of distributed-shared memory and persistent stores. Transactions, described in [18, 33], are a fundamental concept in database systems. See [20] for an extensive treatment of database issues.

Transaction Systems

Countless systems implement transactions for databases,[11] but in this thesis, I only compare LIBXAC primarily to two similar transaction systems: McNamee's implementation of transactions on a single level store, the Recoverable Memory System (RMS), [35], and Saito and Bershad's implementation of the Rhino transactional memory service [41].

Like LIBXAC, both RMS and Rhino provide a memory-mapped interface: a programmer calls functions to attach and detach a persistent segment of memory. Programmers should not store address pointers in this persistent area, as the base address for the segment changes between different calls to `xMmap`. Some persistent storage systems perform *pointer swizzling*, i.e., runtime conversion between persistent addresses on disk and temporary addresses in memory, to eliminate this restriction. This method further simplifies programming, but incurs additional overhead. An alternative to pointer swizzling, adopted by the μDatabase, a library for creating and memory-mapping multiple memory segments [9], is to always attach persistent areas at same address. This scheme allows a program to access only one memory-mapped file at a time, however.

Both RMS and Rhino provide two separate functions for committing and aborting a transaction, while LIBXAC provides one single `xend` function which returns a status

[11] There are many transaction systems in both research and practice. Some examples (but certainly not all) include [8, 12, 13, 35, 38, 41, 44, 45, 46, 47].

of `COMMITTED` or `ABORTED` and automatically aborts the transaction. Although the programmer can control if and when rollback of the shared-memory segment occurs with the first option, LIBXAC's automatic rollback is more convenient for the default case.

Like LIBXAC, both RMS and Rhino automatically detect the memory locations accessed by a transaction, and both specify the same memory model. Aborts only happen synchronously, when the programmer calls the appropriate function. This abort only rolls back the values in the shared-memory segment. The authors of both RMS and Rhino ignore the issue consistent execution because they do not discuss concurrent transactions. Instead, they assume that the programmer or the system designer uses a conventional locking protocol such as two-phase locking [20].

McNamee describes an implementation that could run on a Linux operating system, but Saito and Bershad describe two implementations: one on an extensible operating system, SPIN [7], and one on Digital UNIX. The Digital UNIX implementation appears to rely on the ability to install a callback that runs right before a virtual-memory pageout. Other examples of operating systems with built-in support for transactions on a single-level store are [8, 45, 46].

Persistent Storage Systems

The goal of many persistent storage systems is to provide a single-level store interface. LIBXAC, RMS, and Rhino all provide persistent storage by having a *persistent area* of memory that the programmer can attach and detach. Other systems choose to maintain persistent objects in terms of *reachability*: any object or region that is accessible through a pointer stored anywhere in the system is considered persistent.

Persistent stores are sometimes implemented with compiler support and a special language that allows programmers to declare whether objects should be persistent. Others provide orthogonal persistence (persistence that is completely transparent to the programmer) by implementing a persistent operating system ([40] is one example).

Inohara, et al. in [28] describe an optimistic multiversion concurrency control algorithm for a distributed system. In [28], persistent objects are memory-mapped

shared-memory segments, at the granularity of a page. The programming interface is reversed compared to LIBXAC: first, the programmer calls a function to begin a transaction, and then calls a function to open/attach each object/segment inside the transaction before using it.

Transactional Memory

LIBXAC's programming interface is based on work on transactional memory. Unlike databases, transactional memory supports nondurable transactions on shared-memory machines. Herlihy and Moss described the original hardware mechanism for transactional memory in [24], a scheme that builds on the existing cache-coherency protocols to guarantee that transactions execute atomically. Ananian, Asanović, Kuszmaul, Leiserson, and Lie in [1] describe a hardware scheme that uses `xbegin` and `xend` machine instructions for beginning and ending a transaction, respectively. Instructions between `xbegin` and `xend` form a transaction that is guaranteed to execute atomically.

Although hardware transactional memory systems usually track the cache lines accessed by a transaction, recent implementations of software transactional memory (STM) work with transactional objects. Fraser in [14] implements a C library for transactional objects (FSTM), while Herlihy, Luchangco, Moir, and Scherer implement a dynamic STM system in Java [25], DSTM.

As in [28], the interface of DSTM and FSTM requires the programmer to explicitly open each transactional object inside a transaction before that transaction can access the object. Thus, these systems do not automatically detect the memory locations a transaction accesses. Although both DSTM and FSTM do not handle nesting of transactions, both describe modifications for supporting subsumed nested transactions.

The authors of DSTM describe a `release` function that allows a transaction in progress to drop a transactional object from its read or write set. Using this feature may lead to transactions that are not serializable, but it provides potential performance gains in applications where complete serializability is unnecessary. None

of the authors of DSTM or FSTM focus on the interaction between transactional and nontransactional objects.

DSTM, like Libxac, performs incremental validation for transactions, checking for violations of serializability on a transaction's first access to a transactional object, (i.e., opening a transactional object). DSTM maintains an old copy and a new copy of every transactional object. Transactions always execute consistently: after opening an object, a transaction either accesses the correct copy, or it aborts by throwing an exception.

On the other hand, FSTM uses an optimistic validation policy, with a combination of both synchronous and asynchronous aborts. A transaction is validated when it attempts to commit, and aborts synchronously if there is a conflict. With this scheme, it is possible for transactions to read inconsistent data, causing a null pointer dereference or an infinite loop. Therefore, the system detects these cases by catches faults and by gradually validating the objects touched by a transaction during execution. After detecting these exceptional conditions, the transaction encountering these exceptional conditions, the transaction aborts asynchronously by using the `setjmp` and `longjmp` functions.

In Libxac, `xMmap` returns a pointer to a transactional memory segment. Programming dynamic data structures in this segment is somewhat cumbersome however, as the Libxac prototype does not provide a corresponding memory allocation routine. Object-based transactional interfaces do not suffer from this problem.

2.5 Advantages of the Libxac Interface

In this section I summarize the main advantages an interface based on memory-mapped transactions. Programs that use memory-mapped transactions are more modular than programs that perform explicit I/O or locking operations. With memory-mapped transactions, programmers can easily parallelize existing serial code and code complex but algorithmically efficient data structures. Finally, a memory-mapped transaction system is flexible enough to provide features such as multiversion concur-

rency control and support for durable transactions.

LIBXAC implements `xMmap`, a transactional version of the `mmap` function. Memory-mapping provides the illusion of a single-level storage system, allowing programs to access data on disk without explicit I/O operations. A programmer can easily specify a transaction in LIBXAC by enclosing the relevant code between `xbegin` and `xend` function calls. The runtime automatically detects which pages a transaction accesses, eliminating the need for explicit locking operations. Programs written with LIBXAC are modular because the concurrency properties of a program or data structure are independent of the specific implementation.

Later, in Chapter 5, I describe how I used LIBXAC to easily parallelize existing serial, memory-mapped implementations of a B^+-tree and a cache-oblivious B-tree (CO B-tree). This process using LIBXAC was considerably easier than it would have been using locks, as it was unnecessary for me to understand all the details of the specific implementation. The fact that I was able to easily parallelize a cache-oblivious B-tree shows that memory-mapped transactions facilitate the programming of complex but algorithmically efficient data structures.

Finally, an memory-mapped transaction system is flexible enough to provide extra useful features. Since Libxac uses a multiversion concurrency control algorithm to guarantee that aborted transactions always see a consistent view of the memory-mapped file, read-only transactions can always succeed. LIBXAC could also support transactions that are recoverable after a program or system crash.

The simplicity of a memory-mapped transactional interface is both an advantage and a disadvantage. With memory-mapped transactions, programmers do not have fine-grained control over I/O or synchronization operations. There is a tradeoff between simplicity and performance. In this chapter, I have argued that LIBXAC provides a simple programming interface. Later, in Chapters 4 and 5, I investigate the cost in performance of using this interface.

Chapter 3

The Libxac Implementation

In this chapter, I describe the prototype implementation of the Libxac specification. This prototype demonstrates that it is possible to implement portable memory-mapped transaction system that supports multiversion concurrency control.

The Libxac prototype has the advantage of being portable. In Section 3.1, I present Libxac's system requirements and provides a high-level description of how Libxac uses standard system calls in Linux to support transactions. Since the prototype relies primarily on the `mmap` and `fsync` system calls and the ability to specify a user-level `SIGSEGV` handler, the implementation is more portable than a transaction system that is built on a special research operating system.

In Section 3.2, I explain in greater detail how Libxac executes memory-mapped transactions on a single process. In particular, I explain how Libxac uses the virtual-memory subsystem to buffer pages from disk in main memory, and how RMS [35] and Rhino [41] support memory-mapped transactions on a single process.

I explain how Libxac supports memory-mapped transactions executed on multiple processes in Section 3.3. I describe Libxac's centralized control mechanism, the consistency tree data structure that Libxac uses to ensure transactions are serializable, and related work on concurrency control by Inohara, et al.[28].

In Section 3.4, I summarize the main shortcomings of the Libxac prototype and explain how the implementation might be improved. First, Libxac's centralized control is a potential bottleneck that limits concurrency. Second, although Libxac in

principle writes enough data to disk to recover from crashes, the recovery mechanism has not been implemented yet and some minor changes to the structure of LIBXAC's log files need to be made before LIBXAC can fully support recoverable transactions running on multiple processes. Finally, the prototype uses several unsophisticated data structures which impose unnecessary restrictions on the LIBXAC interface.

3.1 Overview

In this section, I explain how a memory-mapped transaction system can be implemented portably. I first describe the system requirements of LIBXAC, and then present a high-level description of how LIBXAC uses the `mmap` system call and user-level `SIGSEGV` handlers in Linux to execute a simple, nondurable transaction.

System Requirements

The LIBXAC prototype is portable because it is designed as a user-space C library for systems running Linux. It does not rely on any special operating system features or hardware, and can be adapted to any system that provides the following functionality:

1. *Memory mapping*: The operating system must support memory mapping of pages, i.e., `mmap` and `munmap`, with read/write, read-only, and no-access memory protections. Programs must be able to change the mapping of a particular page. The OS must also support multiple mappings, each with different memory protections, for the same page in a file. The `mmap` function must also support the `MAP_FIXED` argument, which allows the programmer to force a memory map to begin at particular address. Durable transactions also require `msync` to flush changes from a memory-mapped file to disk.

2. *File management*: LIBXAC uses the `open` system call to open a file and get its file descriptor. For durable transactions, the operating system must support `fsync` or a similar method that forces changes made to a file out to disk.[1]

[1]The `man` page states that when `fsync` returns, the data may not actually be written to disk if the harddrive's write-cache is enabled.

3. *`SIGSEGV` handler*: the system must allow programs to run a user-specified fault handler when the memory-protection on a page is violated.

4. *Locking primitives*: Since LIBXAC requires must execute some runtime methods atomically, I resort to using simple spin locks in the implementation. Ideally, the runtime could also be implemented with non-blocking synchronization primitives such as compare-and-swap (CAS) instructions.

Aside from these important system-dependent components, the runtime is implemented using standard C libraries.

Executing a Simple Transaction

In LIBXAC, every transaction that executes has the following state associated with it:

- *Readset and Writeset*: The readset and writeset are the set of pages in the shared-memory segment that the transaction has read from and written to, respectively.[2]

- *Status*: A transaction in progress is either `PENDING` or `FAILED`. This status changes from `PENDING` (`FAILED`) to `COMMITTED` (`ABORTED`) during an `xend`.

- *Global id*: During a call to `xbegin`, every transaction is assigned a unique integer id equal to the current value of a global transaction counter. This counter is incremented after every `xbegin`.

- *Runtime id*: Every transaction is also assigned a runtime id that is unique among all *live* transactions. A transaction is alive until the transaction manager has determined that it can safely kill it (i.e delete its state information).

The prototype stores the transaction state in a data structure that is globally accessible to all processes that have `xMmap`ed the shared-memory segment.

Since I argued in Section 1.1 that programs that use memory mapping are simpler than those using explicit I/O operations, it is not surprising that LIBXAC runtime

[2] The readset and writeset are defined to be disjoint sets.

itself uses memory mapping. As a transaction executes, LIBXAC modifies both the global state information and the memory-map for each process. Figure 3-1 illustrates the steps of the execution of a simple transaction.

1. The `xMmap` call initially maps the entire shared-memory segment (`x` in this example) for the current process with no-access memory protection (`PROT_NONE`). The `xbegin` call starts the current transaction with a status of `PENDING`.

2. Line 6 causes a segmentation fault when it attempts to read from the first page of mapped file. Inside the `SIGSEGV` handler, LIBXAC checks the global transaction state to determine whether one or more transactions need to be `FAILED` because this memory access causes a conflict. As described in Section 2.2, a failed transaction always sees a consistent value for `x[0]`, and keeps executing until the `xend` call completes and the transaction becomes `ABORTED`.

   Whether or not the transaction is `PENDING` or `FAILED`, LIBXAC `mmap`'s the correct version of the page with read-only permission, `PROT_READ`.

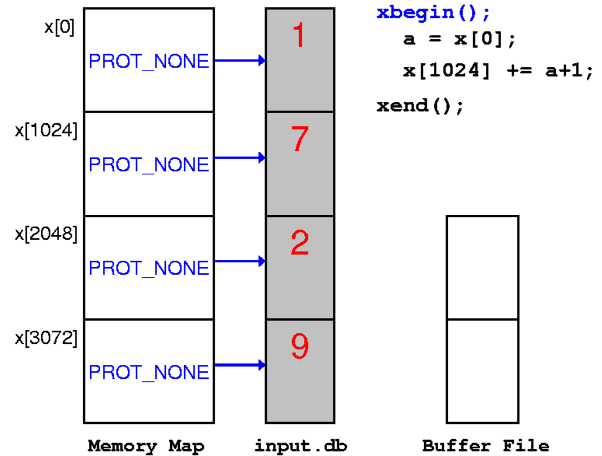
3. Line 7 causes two segmentation faults when it tries to write to the second page of the file.[3] LIBXAC handles two segmentation faults. LIBXAC handles the first `SIGSEGV` as in Step 2, checking for conflicts and mapping the page as read-only. On the second fault, LIBXAC checks for conflicts again, possibly failing one or more transactions. The runtime then creates a copy of the page and `mmap`'s the new copy with read/write access, `PROT_READ|PROT_WRITE`.

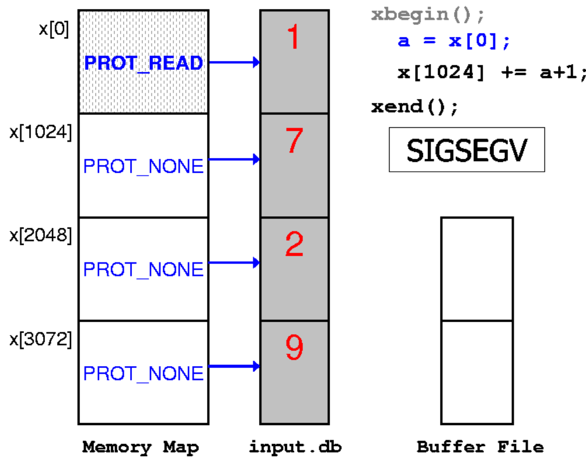
4. After mapping the second page with read/write access, the transaction updates the value of `x[1024]`. When the `xend` function is reached, the transaction commits if its status is pending and aborts if the status is failed. All pages that the transaction touched are mapped with no-access protection again.

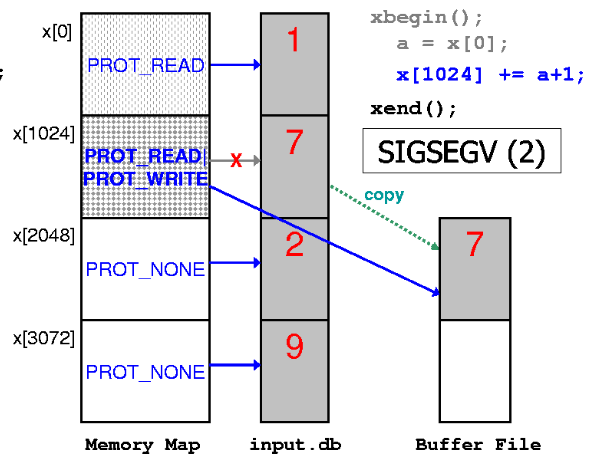
5. It may be incorrect to immediately replace the original version of a page because failed transactions may still need to read the older versions. Eventually, dur-

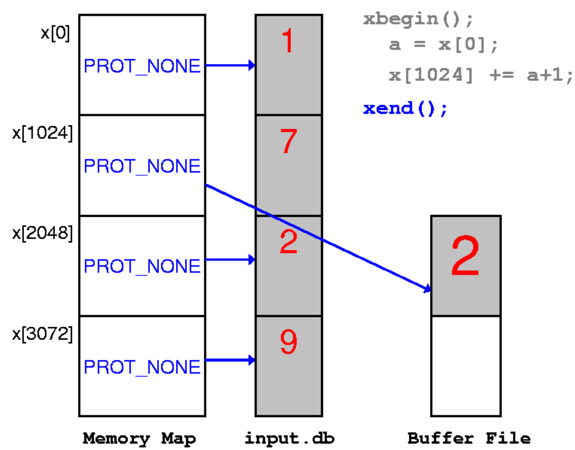
[3]To my knowledge, Linux does not provide any mechanism for differentiating between a `SIGSEGV` caused by a read and one caused by a write.

```
1  int main(void) {
2     int a; int* x;
3     x = (int*)xMmap("input.db",
4                     4*4096);
5     xbegin();
6       a = x[0];
7       x[1024] += (a+1);
8     xend();
9
10    xMunmap("input.db", 4*4096);
11    return 0;
12 }
```









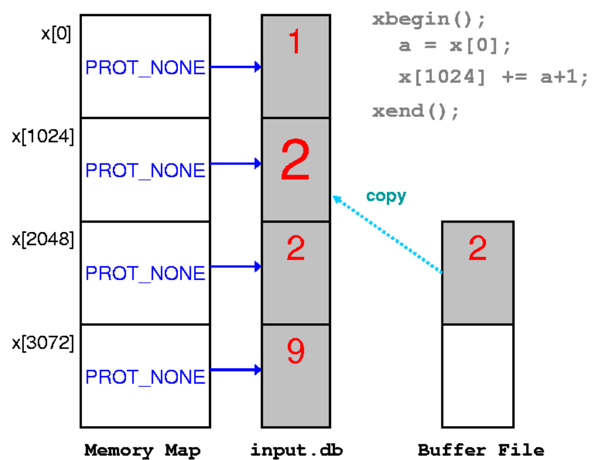


Figure 3-1: Changes to the memory map for a simple transaction.

ing execution of a later transaction or during a checkpoint operation, LIBXAC garbage collects new versions of pages by copying them back into the original file when it determines that the older versions are no longer necessary.

These first four steps (`xbegin`, the first read of a page, the first write to a page, and `xend`) represent the basic actions handled by the runtime. In Section 3.3, I describe how LIBXAC handles these actions with multiple, concurrent transactions.

3.2 Transactions on a Single Process

In this section, I explain in more detail how LIBXAC supports memory-mapped transactions executed on a single process. In particular, I describe how LIBXAC uses the virtual-memory subsystem to buffer pages from disk in memory and how the transaction systems in [35, 41] support memory-mapped transactions on a single process.

LIBXAC buffers the pages touched by a transaction in virtual memory by using `mmap`. For nondurable transactions, LIBXAC stores the copies of pages made by transaction writes in a memory-mapped *buffer file* on disk. Pages from this buffer file are allocated sequentially, as transactions write to new pages. When a transaction reads from a page, the runtime `mmap`'s the correct page from the original user file or the buffer file with read-only access. The runtime either creates a new buffer file when the existing buffer file is full, or reuses an old buffer file once all the new versions of pages in that file have been copied back into the original file.

For durable transactions, LIBXAC stores the pages written by transactions in memory-mapped log files. Since log information must be maintained until the programmer explicitly deletes it, the runtime does not reuse log files. For durable transactions, the log file also contains the following types of metadata pages:

- `XBEGIN`: On an `xbegin` function call, the runtime reserves a page in the log for recording the the transaction's global id. On an `xend` call, this page is updated with the list of pages in the log file that the transaction wrote to. This list may spill over onto multiple pages for large transactions.

- `XCOMMIT`: The runtime writes this page in the log when a transaction commits. This page stores the global id and also a checksum for all other the pages (data and metadata) written by this transaction.

- `XABORT`: This page records the global id of an aborted transaction.

- `XCHECKPT_BEGIN` and `XCHECKPT_END`: The runtime writes these pages when a checkpoint operation starts and finishes, respectively.

When a durable transaction commits, the runtime performs an asynchronous `msync` followed by an `fsync` on the log file(s) to force the transaction's data and metadata out to disk. After a program crash, the recovery routine uses the checksum to determine whether a transaction successfully wrote all of its pages to disk.[4]

When the user calls `xCheckPoint`, the runtime determines which pages will be garbage collected, i.e., copied back to the original file, and records this list in a `XCHECKPT_BEGIN` page. This page is synchronized on disk using `fsync`, and then the new versions are copied back to the original file. Those pages are forced to disk with `fsync`, and a `XCHECKPT_END` page is written and synchronized with a final `fsync`.

The `XCHECKPT_BEGIN` page also contains a list of pointers to the `XBEGIN` pages of all transactions whose new pages could not be garbage-collected. After a `XCHECKPT_END` page appears in a log, the only committed transactions whose updates may need to be copied back to the original file are those transactions whose `XBEGIN` pages are either referenced in the `XCHECKPT_BEGIN` page or appear after the `XCHECKPT_BEGIN`.

Although Libxac in principle writes enough data to disk to do recovery, I have not yet implemented the recovery module, and I need to cleanly separate metadata and data pages in the log file to correctly support recovery when multiple processes execute transactions concurrently. One way of achieving this separation is to write the metadata and data into separate log files. This change should not hurt performance if

[4]The prototype does not yet compute this checksum. The data in Table C.8 in Appendix C and in Chapter 4 show that using a hash function such as `md5` is not too expensive compared to the cost of writing the page out to disk. Alternatively, the runtime could force the data out to disk with an `fsync`, write the `XCOMMIT` metadata page, and then perform a second `fsync`. This method ensures that a transaction's `XCOMMIT` page never makes it to disk before any of its data pages.

the system can use multiple disks. See Appendix B for a more detailed discussion of how transaction recovery in Libxac might be supported. Since transaction recovery has been extensively studied, there are likely to be many ways of handling recovery in Libxac. For example, many transaction systems in the literature derive their recovery mechanism from the ARIES system [36].

Comparison with Related Work

In this section, I compare and contrast Libxac with the RMS [35] and Rhino [41] transaction systems. See [20] for a more general, extensive treatment of transaction systems.

Transaction systems for databases traditionally maintain explicit buffer pools for caching pages accessed by transactions. Whenever a transaction accesses an unbuffered page, the system brings the page into the buffer pool, possibly evicting another page if the buffer is full. Since paging is also done by an operating system, researchers have explored the integration of transaction support into operating systems. In [35], McNamee argues that in an environment where other programs are competing for memory, a transaction system that maintains an explicit buffer pool does not perform as well as one that integrates buffer management with the operating system. The primary reason is the phenomenon of *double paging* [17], the fact that a page cached by the buffer pool may have been paged out by the operating system.

McNamee also argues that most commercial operating systems do not provide support for transactions, and most research operating systems use special OS features to integrate buffer management with virtual memory. For example, the Camelot distributed transaction system uses the external pager of Mach, a research operating system [46]. This pager allows users to specify their own routines for moving pages between disk and main memory. Other examples of transaction systems implemented on top of special operating systems include [8, 11, 12, 19]. Because integration usually happens only on special operating systems, McNamee presents a hybrid transaction system, RMS, that is compatible with commercial operating systems. Like Libxac, this system works by manipulating process memory maps and virtual memory pro-

tections.

Saito and Bershad in [41] also implement Rhino, a transaction system in both Digital UNIX and also on SPIN [7], an extensible operating system. Their system also memory-maps the database files to avoid double paging, and uses virtual memory protections to automatically detect which pages transactions write to. One main point of [41] is that an extensible operating system such as SPIN can support automatic write detection more efficiently than Digital UNIX because SPIN requires fewer user-kernel boundary crossings to handle a page fault.

There are several interesting comparisons to make between LIBXAC and the RMS and Rhino. First, LIBXAC, like RMS and Rhino, integrates buffer pool management with the virtual-memory system by using memory mapping. Also, both RMS and Rhino automatically detect transaction writes by memory-mapping the shared-memory segment with read-only protection by default. When a transaction writes to the page, a `SIGSEGV` handler creates a before-image, (a copy of the old data), and then `mmap`'s the existing page with read/write protection.

Both RMS and Rhino must guarantee that the before-image is written on disk before this call to `mmap`. Otherwise, the database will be corrupted if the transaction modifies the page, the OS writes this temporary page out to disk, and the program crashes, all before the before-image is saved to disk. McNamee's scheme synchronously forces this before-image out to disk before calling `mmap`. Therefore, this scheme require a synchronous disk write every time a transaction writes to a new page, even when nothing needs to be paged out. The experimental results in Chapter 4 indicate that a synchronous disk write in a modern system is quite expensive. Saito and Bershad avoid this problem because SPIN, like Mach, allows users to specify their own procedures for pageouts. Before a new version is paged out, the system has a chance to write the before-image to disk first.

LIBXAC can avoid performing synchronous disk writes in the middle of a transaction or using a special OS feature such as an external pager because the runtime maintains redo records instead of undo records. On a transaction write, LIBXAC creates a copy of the original page, but `mmap`'s the new copy of the data instead of

the original. Thus, the before-image on disk is never overwritten if the new version is paged out. This policy is similar to a *no-steal* buffer replacement, because the before-image always remains in the database before the transaction commits.[5] Alternatively, in a *steal* policy, the before-image is overwritten during a pageout. See [20] for a textbook discussion of buffer replacement policies.

Since LIBXAC has a complicated multiversion concurrency control algorithm, it is natural to `mmap` the new copies of a page instead of the old copy: there is only one committed version of a page, but multiple working copies. In contrast, the systems described in [35, 41] maintain at most two copies of a given page at any one time. Both McNamee and Saito and Bershad do not discuss concurrency control since both assume that a standard locking protocol such as two-phase locking is used.

Because LIBXAC maintains multiple versions of a page, transactions must find the correct version to `mmap` before every page access. Pages that are contiguous in the shared-memory segment may actually be mapped to discontiguous pages in the log file. Eventually, however, garbage collection will copy the pages back to the original file, and the original ordering. This problem of fragmented data occurs in database systems that use shadow files instead of write-ahead-logging. Write-ahead logging, the mechanism used by [35, 41], is the technique of writing undo or redo information to a log on disk before modifying the actual database. A system that uses shadow files constantly switches between two versions of a page: one version that is the committed version, and one that is the working version that active transactions modify. Two examples of systems that use shadow files are [13, 19]. When there are n processes attached to the shared-memory segment, Libxac's multiversion concurrency control could be implemented by reserving enough virtual address space to have n extra shadow copies of the segment, one for each process.

Finally, one idea for a future implementation of LIBXAC is to separate the data and metadata pages in the log into separate files, ideally, on two different disks. This design allows metadata entries in the log to be smaller, as we would not need to waste

[5] The steal/no-steal definitions tend to assume single-version concurrency control, so they not be completely applicable for LIBXAC.

an entire page to store a global transaction id for an XBEGIN or XCOMMIT. This scheme would also write less data to disk on a commit if the runtime logged only the diffs of the pages that a transaction wrote. The authors of [41] in their study concluded that computing page-diffs provided better performance than page-grain logging for small transactions. Using page diffs had previously been proposed in [47].

3.3 Concurrent Transactions

In this section, I explain how Libxac supports memory-mapped transactions executed on multiple processes. The runtime's centralized control mechanism uses locks to ensure that the four primary events, the xbegin function call, a transaction's first read from a page, a transaction's first write to a page, and the xend function call, are all processed atomically. I also describe the consistency tree data structure that Libxac uses to ensure transactions are serializable, and one example of related work on concurrency control, [28].

The Libxac Runtime

In Section 3.1, I described the four primary events that the runtime handles: xbegin, a transaction's first read from a page, first write to a page, and xend. With multiple concurrent transactions, the runtime uses locks to process each event atomically.

The Libxac prototype is implemented using centralized control, storing all control data structures in a control file which is memory-mapped by a process during a call to xMmap. This control file stores four main pieces of information: the transaction state described in Section 3.1, transaction page tables for recording which transactions are reading or writing a particular page, the log information required to manage the buffer/log files described in Section 3.2, and finally a *consistency tree* used for concurrency control between transactions.

The prototype obeys a relatively simple locking protocol of holding a global lock while processing a transaction event. To improve concurrency, the runtime does not hold the global lock while changing the memory map with mmap or munmap or during

calls to `msync` or `fsync`. LIBXAC decouples log file manipulation and transaction state modification by using a separate global lock for managing the log files.

LIBXAC's centralized control is easy to implement, but represents a bottleneck that limits scalability to systems with many processes. Since the primary target system for the LIBXAC prototype is symmetric multiprocessor systems with only 2, 4, or 8 processors, a centralized control mechanism may be tolerable. A scalable solution, however, would have an efficient distributed control mechanism. Using a fine-grained locking scheme could also improve system performance. A nonblocking implementation of the runtime using synchronization primitives such as compare-and-swap or load-linked-store-conditional instructions may also be a complex but efficient alternative to using global locks.

Consistency Tree

LIBXAC supports the memory model presented in Section 2.2 by maintaining a *consistency tree* of transactions. Every transaction in the system is represented by a node in this tree. The root is a special committed transaction T_0 that represents main memory (T_0's writeset is the entire shared-memory segment). A transaction T is said to *own a version of a page* x if it writes to x. LIBXAC uses the tree to determine which version of a page a transaction should read when it executes.

An edge in the consistency tree captures potential dependencies between transactions, i.e., if a transaction T is an ancestor of T', then T comes before T' in some serializable schedule of transactions. Recall that every transaction can be in one of four states: `PENDING`, `FAILED`, `COMMITTED`, and `ABORTED`. A valid consistency tree must satisfy the following invariants:

Invariant 1: For every page x in the readset of a transaction T, T reads the version from the closest ancestor of T in the tree that owns x.

Invariant 2: Only `COMMITTED` transactions have children, and a transaction has at most one `COMMITTED` child.

Figure 3-2 exhibits one example of a consistency tree. By Invariant 1, T_3 can read page T_1's version of x only if both T_5 and T_6 do not write to x. Invariant 1 guarantees

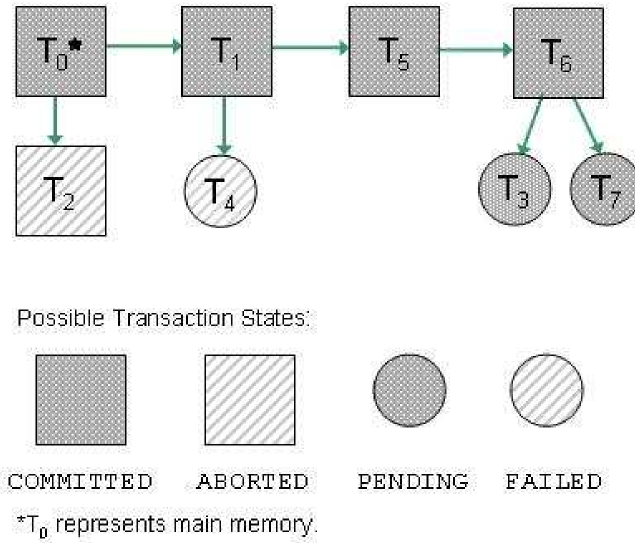


Figure 3-2: An example of a consistency tree.

that a parent-child relationship between two committed transactions corresponds to a valid serial ordering of the two transactions. Because T_1 is the parent of T_5, it is correct to order T_1 before T_5. This ordering is the only correct one if T_5 reads any page written by T_1.

By Invariant 2, T_2, T_3, T_4 and T_7 cannot have children, and T_0, T_1, T_5, and T_6 can each have at most one committed child. Invariant 2 is required for LIBXAC's memory model. Suppose a new transaction T_8 could read a page x from an uncommitted transaction T_3. Since the runtime traps only T_3's first write to a page, T_8 sees an inconsistent view of x if T_8 reads x and then T_3 writes to x again. This invariant implies that the tree is an ordered list of unordered lists. Each unordered list is a committed transaction with some number of uncommitted children.

It can be shown that if transactions read and write pages in a way that maintains Invariants 1 and 2 on the consistency tree, then the schedule of transactions is serializable. The correct serialization order for transactions corresponds to a pre-order traversal of the tree, where the committed children of a transaction are visited last.

The consistency tree is simplification of a serialization graph data structure [6]. With a complete serialization graph, a schedule of transaction reads and writes is seri-

alizable if and only if the graph does not have a cycle. The consistency tree maintains less information and only permits a subset of all possible serializable schedules. Every valid consistency tree allows some, but not all serializable schedules of transactions.

I have only specified LIBXAC's memory model for committed and aborted transactions. The prototype implementation however, maintains an additional invariant for pending and failed transactions:

Invariant 3: If the readsets and writesets of all pending (failed) transactions do not change (i.e., all transactions stop reading from or writing to new pages), then all pending (failed) transactions can be committed (aborted).

Invariant 3 states that LIBXAC performs incremental validation of transactions. Since the runtime checks that serializability is maintained after every page access, during an `xend`, a transaction can automatically commit if its status was still `PENDING`.

Implemented Policies

LIBXAC uses the consistency tree to implement the following generic concurrency control algorithm:

1. On an `xbegin`, LIBXAC inserts a new `PENDING` transaction as a child of some `COMMITTED` transaction in the tree. This step satisfies Invariant 2.

2. Whenever a transaction reads from (or writes to) a page for the first time, the runtime updates the transaction's readset (or writeset) and checks the transaction page tables for a possible transaction conflict. A conflict occurs if at least one other `PENDING` transaction is already accessing that page, and one of those transactions or the current transaction is writing to the page. If there is a conflict, LIBXAC may fail some transactions in order to preserve Invariant 3.

   Whether the transaction is `PENDING` or `FAILED`, on a page read, the runtime walks up the consistency tree to determine which version of the page to read. On a page write, the system copies the version of the page that was previously being read. This step satisfies Invariant 1.

3. On an xend, LIBXAC either commits a PENDING transaction or aborts a FAILED transaction.

4. In steps 2 or 3, the runtime may change the parent of a PENDING transaction to be a different COMMITTED transaction as long as Invariants 1 and 3 are satisfied.

This framework is general enough to support most reasonable concurrency control algorithms.[6] In the prototype, however, I have implemented only two simple policies for concurrency control. Both policies satisfy two additional constraints on the consistency tree, that all PENDING transactions must be children of the last COMMITTED transaction in the tree, and that transactions that have FAILED never become PENDING again. The first constraint implies that any particular page has either multiple PENDING readers or one PENDING writer.[7] The second constraint means that the transaction page table can ignore page accesses by a FAILED transaction because that transaction never generates conflicts by becoming PENDING again.

With these two constraints, when we have both reading transactions and a writing transaction for a page, we need to decide whether to abort the writer or all the readers. I arbitrarily chose to implement two abort policies:

- *Self-Abort*: A transaction aborts itself whenever it conflicts on a page x. More specifically, when a transaction T tries to read x, it aborts if there is already a writer for x. Similarly, when T tries to write to x, it aborts if there is already a reader or writer for x.

- *Oldest-Wins*: Always abort the transaction(s) with larger global id. Since LIBXAC assigns global transaction ids according to an increasing counter, this id acts as a timestamp. A new reader aborts the existing writer only if the reader has a smaller id. A new writer aborts an existing writer or all existing readers only if it has the smallest id of all the transactions accessing the page.

[6] A consistency tree can also support optimistic concurrency control if we omit Invariant 3.

[7] In the more general case, it is possible to have T_1 reading page x and T_2 writing to x simultaneously, as long as T_1's parent is earlier in the chain of committed transactions than T_2's parent. For example, if T_1 is a read-only transaction specified using xbeginQuery and xendQuery, it never fails.

Under this policy, livelock is impossible because the transaction with the smallest time stamp never fails. Starvation is still possible however, because LIBXAC assigns a new global id to a transaction after an abort.

LIBXAC could also support the two opposite policies:

- *Selfish-Abort*: Whenever a transaction T discovers a conflict with a transaction T', it aborts T'.
- *Youngest-Wins*: Always abort the transaction(s) with the smaller global id.

The last two policies are *obstruction-free* [26]: a transaction always succeeds if all other transactions stop running. The self-abort and oldest-wins policies do not have this property; if one process crashes while executing its transaction, the other transactions end up waiting indefinitely on that transaction. Some questions that I have not explored include what the best policies are to use in different situations, and whether some policies are more efficient to implement than others, particularly with a distributed control system for the runtime.

Garbage Collection

LIBXAC also uses the consistency tree to determine when it is safe to garbage-collect pages. It is safe to delete a transaction's version of a page if no `PENDING` or `FAILED` transactions can access that version of the page.

In the consistency tree, we say a chain of committed transactions can be *collapsed* if all transactions in the chain except the last have no `PENDING` or `FAILED` children. LIBXAC collapses the chain of transactions by transferring ownership of the latest version of each page to the first transaction in that chain. Transferring ownership of a page back to T_0 corresponds to copying the version back into the original file.

In the example in Figure 3-2, T_5 and T_6 can be collapsed together, leaving only T_5 with T_3 and T_7 as its children. T_6 can then be safely deleted from the consistency tree. LIBXAC does garbage collection of transactions only when the number of transactions in the tree goes above a fixed threshold or during a checkpoint operation.

Comparison with Related Work

The textbook concurrency control algorithm for database systems is two-phase locking (2PL) [20]. In 2PL, transactions first enter an expanding phase when they can only acquire locks, and then a shrinking phase when they can only release locks. For 2PL in a multiversion system, the shrinking phase may occur at the end of the transaction, and may also involve acquiring certification locks to validate the transaction. Because LIBXAC does incremental validation of transactions, its concurrency control can be thought of as 2PL, except that transactions never wait to acquire a lock. Instead, either the transaction waiting on the lock or the transaction holding the lock gets aborted immediately.

Alternatively, LIBXAC could use an optimistic concurrency control algorithm like the one originally proposed in [32]. Optimistic algorithms execute the entire transaction and then check for conflicts once, during commit. One way to implement an optimistic policy using a consistency tree is to never switch the parent for a `PENDING` transaction until that transaction tries to commit.

The LIBXAC prototype is not scalable, partly because accesses to the consistency tree occur serially. One possible improvement to LIBXAC is to use a multiversion concurrency control algorithm designed for distributed systems. The authors of [28] present one such algorithm, the page-based versioned optimistic (VO) scheme. First, they describe the VO scheme for a centralized system. When a transaction begins, it is assigned a timestamp that is 1 more than the timestamp of the last committed transaction in the system. When a transaction T writes to a page for the first time, it creates a version with its timestamp. When T reads a page x, it finds the version of x with the greatest timestamp less that T's timestamp. If T is read-only, it always commits. T aborts if, for some page x that T wrote to, some other transaction T' has written a newer version of x. In the VO scheme, a read-only transaction can be serialized before or after a committed transaction, but a read/write transaction can only be serialized after all other committed transactions.

The consistency tree framework can in theory implement a VO scheme. Trans-

action T' is a child of T in consistency tree if and only if in the VO scheme, T''s timestamp is one more than T's timestamp. The fact that a transaction's timestamp never changes in the VO scheme implies that a `PENDING` transaction in the consistency tree never switches parents until it tries to commit. The VO scheme validates a transaction T by checking whether T can be serialized after all committed transactions. This approach is equivalent to checking whether a `PENDING` transaction can be the child of the last committed transaction during `xend`.

3.4 Conclusion

In this section, I summarize the main shortcomings of the LIBXAC prototype, and explain how the implementation might be improved in a more complete system supporting memory-mapped transactions.

The primary drawbacks to the LIBXAC prototype are:

1. LIBXAC is implemented with centralized control data structures. One possible improvement is to applying ideas from distributed systems and work such as [28] to create a more decentralized control for LIBXAC.

2. LIBXAC's control data structures have relatively naive implementations that some impose unnecessary restrictions on transactions (see Appendix A).

3. The structure of LIBXAC's log files for durable transactions does not fully support recovery when transactions are executed on multiple processes. As I discuss in Appendix B, one possible improvement is to separate the metadata pages and data pages into different files.

Although these problems with the current implementation are significant, I believe none of them are fatal. In Chapter 5, I present results from experiments doing random insertions on search trees using LIBXAC. When each insertion was done as a durable transaction, the performance of LIBXAC search trees ranged from being 4% slower to actually 67 % faster than insertions done on Berkeley DB's B-tree. In light of the

issues I have described, this result is quite promising. If even a simple implementation can achieve reasonable performance in some cases, then there is hope that a more sophisticated and optimized version can support Libxac's specification efficiently in practice.

Chapter 4

A Performance Study

Memory-mapped transactions are easy to use because the underlying system automatically handles I/O operations and concurrency control, but this convenience comes at a cost. In this chapter, I describe several experiments designed to measure the performance of memory-mapped transactions in LIBXAC. I use these results to construct approximate performance models for both nondurable and durable transactions whose working sets fit into main memory.

In Section 4.1, I describe the four different machines on which I tested the LIBXAC prototype: a 3.0 GHz Pentium 4, a 2-processor 2.4 GHz Xeon, a 4-processor, 1.4 GHz AMD Opteron, and a 300 MHz Pentium II.

In Section 4.2, I estimate the cost of executing a nondurable memory-mapped transaction by measuring the time required for expensive operations: entering and exiting a `SIGSEGV` handler and executing the `mmap` system calls. For a nondurable transaction that reads from R different pages and writes to W different pages, I estimate the additive overhead on a modern machine for executing the transaction is roughly of the form $aR + bW$, where $b \approx 2a$ and a is in the range of 15 to 55 μs. Using the advisory function to inform the runtime which pages a transaction accesses reduces this overhead on modern machines by anywhere from 20% to 50%.

Section 4.3 describes similar experiments for estimating the cost of executing a durable memory-mapped transaction. The single-most expensive operation for a durable transaction is the call to `fsync` that forces data to be written out to disk

on a transaction commit. For durable transactions that write to only a few pages, I conjecture a performance model of the form $aR + bW + c$, where a is tens of microseconds, b is a few hundred microseconds, and c is fixed cost of anywhere from 5 to 15 ms.

Section 4.4 presents results from experiments designed to measure the potential concurrency of Libxac. On a multiprocessor machine, when two different processes ran independent nondurable transactions, Libxac achieved near-perfect linear speedup when the work done on the page touched by each transaction required time approximately two orders of magnitude greater than the overhead of accessing that page. With durable transactions, however, the same Libxac program did not exhibit any speedup on two processors, most likely because the time to synchronously write data to disk during a transaction commit represents a serial bottleneck.

Since the overhead of automatically detecting pages accessed is significant for nondurable transactions, and the synchronous writes to disk are a serial bottleneck for concurrent, durable transactions, the current Libxac prototype seems best suited for executing durable transactions on a single process. Improving the implementation to work with multiple disks may improve the performance of concurrent, durable transactions.

4.1 The Hardware

This section describes the four machines I used for testing the Libxac prototype.

1. A 3.0GHz Pentium 4 with hyperthreading and 2GB of RAM. The system has a RAID 10 configured with 4 120GB, 7200-rpm SATA disks with 8MB cache and a 3ware 8506 controller. The drives were mounted with an ext3 filesystem. The system runs Fedora Core 2 with Linux kernel 2.6.8-1.521-smp.[1]

2. A machine with 2, 2.4GHz Intel Xeon processors and 1GB of RAM. The harddrive is a 80GB, 7200rpm Barracuda ATA IV with a 2MB cache. The drive was

[1] The kernel was modified to install `perfctr`, a package for performance monitoring counters.

mounted with an ext3 filesystem. The system was running Fedora Core 1 with Linux kernel 2.4.22-1.2197.nptlsmp.

3. A machine with 4, 1.4GHz AMD64 Opteron processors and 16 GB of RAM. The system was running SUSE 9.1, with Linux kernel 2.6.5-7.147-smp. This machine has two different disks that I ran tests on:

   (a) A 73.4GB, 10,000rpm IBM Ultrastar 146Z10 with 8MB cache and an Ultra320 SCSI interface, mounted with the ReiserFS filesystem.

   (b) A 146.8GB, 10,000rpm IBM Ultrastar disk with 8MB cache and an Ultra320 SCSI interface, mounted with an `ext3` filesystem.

4. A 300 MHz Pentium II with 128 MB of RAM, running Redhat 8.0 with Linux Kernel 2.4.20. The machine has a 4.3 GB ATA disk drive.

The first three machines are relatively modern machines; the first is a standard single-processor machine while the second and third are more expensive multiprocessor machines. The fourth is a relatively old system with a slow processor and limited amount of memory. Testing Libxac on these different systems ensures I am not obtaining results that are specific to a single machine.

Each aspect of the system has a different effect on Libxac's performance. The type and number of processors affects how quickly transactions can execute and how much concurrency we can expect to get. See Table 4.1. The amount of RAM determines how many pages can be buffered in main memory. A nondurable transaction that accesses data that is entirely in memory should not incur the cost of disk writes. Changes made between Linux kernel 2.4 and 2.6 may affect the performance of system calls such as `mmap`. Finally, for durable transactions, the cost of `msync` and `fsync` depends on the rotational latency, seek times, and write speed of the disk drives.

Unless otherwise noted, all times for nondurable transactions in Section 4.2 were measured by reading the processor's cycle counter twice using the `rdtsc` instruction, and recording the time difference. For durable transactions, in Section 4.3, I instead used `gettimeofday`. See Appendix C.1 for details on the resolution of the timers.

Machine	Processor Speed	Time per Cycle
1	3.0 GHz	0.33 ns
2	2.4 GHz	0.42 ns
3	1.4 GHz	0.71 ns
4	300 MHz	3.33 ns

Table 4.1: Processor speeds and time per clock cycle for the test machines.

4.2 Performance of Nondurable Transactions

In this section, I present a rough performance model for small nondurable transactions. The additive overhead on a modern machine for executing a nondurable transaction that reads from R distinct pages and writes to W distinct pages is approximately $aR + bW$, where a is a constant between 10 and 60 μs and $b \approx 2a$. Using an advisory function to inform the runtime which pages a transaction accesses reduces this overhead by anywhere from 20 to 50%. I also describe several experiments that suggest that a significant fraction of the overhead for a transaction is spent entering and exiting the `SIGSEGV` handler, calling `mmap`, and handling page faults.

4.2.1 Page-Touch Experiment

For a transaction that reads from R pages and writes to W pages, I conjecture a performance model of $aR + bW$ and in this section I describe experiments designed to estimate the constants a and b. From the results, I conclude that for small nondurable transactions, a is on the order of tens of microseconds, and $b \approx 2a$. Whenever a transaction reads from or writes to a new page, the LIBXAC runtime must `mmap` the appropriate page in memory. During a transaction commit or abort, the runtime must change the memory protection of every accessed page back to no-access. The performance model is reasonable because the time for these operations is proportional to the number of pages accessed.

I tested the accuracy of this model with an experiment that ran the transactions shown in Figure 4-1. Transaction (a) reads from n consecutive pages, while (b) writes to n consecutive pages. Figure 4-2 shows the results on Machine 1 for transactions

```
// Transaction that reads
//   from n pages
int* x = xMmap("input.db",
               n*PAGESIZE);
int i, value;
   ...

xbegin();
for (i = 0; i < n; i++) {
  value = x[i*PAGESIZE/sizeof(int)];
}
xend();
```

(a)

```
// Transaction that writes
//   to n pages
int* x = xMmap("input.db",
               n*PAGESIZE);
int i;
   ...

xbegin();
for (i = 0; i < n; i++) {
  x[i*PAGESIZE/sizeof(int)] = i;
}
xend();
```

(b)

Figure 4-1: A simple transaction that (a) reads from n consecutive pages, and (b) writes to n consecutive pages.

(a) and (b) as n varies between 1 and 1024. Transaction (a) took an average of about 110,000 clock cycles (37 μs) per page read. The data was less consistent for (b); for $n \geq 2^4$, the average time per page written was roughly 200,000 clock cycles.

The sharp peak for transaction (b) in Figure 4-2 is somewhat typical behavior for long running LIBXAC programs. Although the average and median times to execute a small transaction are fairly stable, the maximum time is often one or more orders of magnitude greater than the average time. Since Linux does not generally provide any bound on the worst-case time for `mmap` or other system calls, this result is not too surprising. Also, because I measure real time in all experiments, any time when the test program is swapped out is also included. One of these phenomena may possibly explain the sharp peak.

4.2.2 Page-Touch with an Advisory Function

In this section, I present data that suggests that using the advisory function described in Section 2.3 for the page-touch experiment reduces the cost per page access on a modern machine by anywhere from 20 to 50%. LIBXAC detects which pages a transaction accesses by handling the `SIGSEGV` signal caused by the transaction first attempt to read from or write to a page. The advisory function, `setPageAccess` informs the runtime that a transaction is about to access a page. This optimization

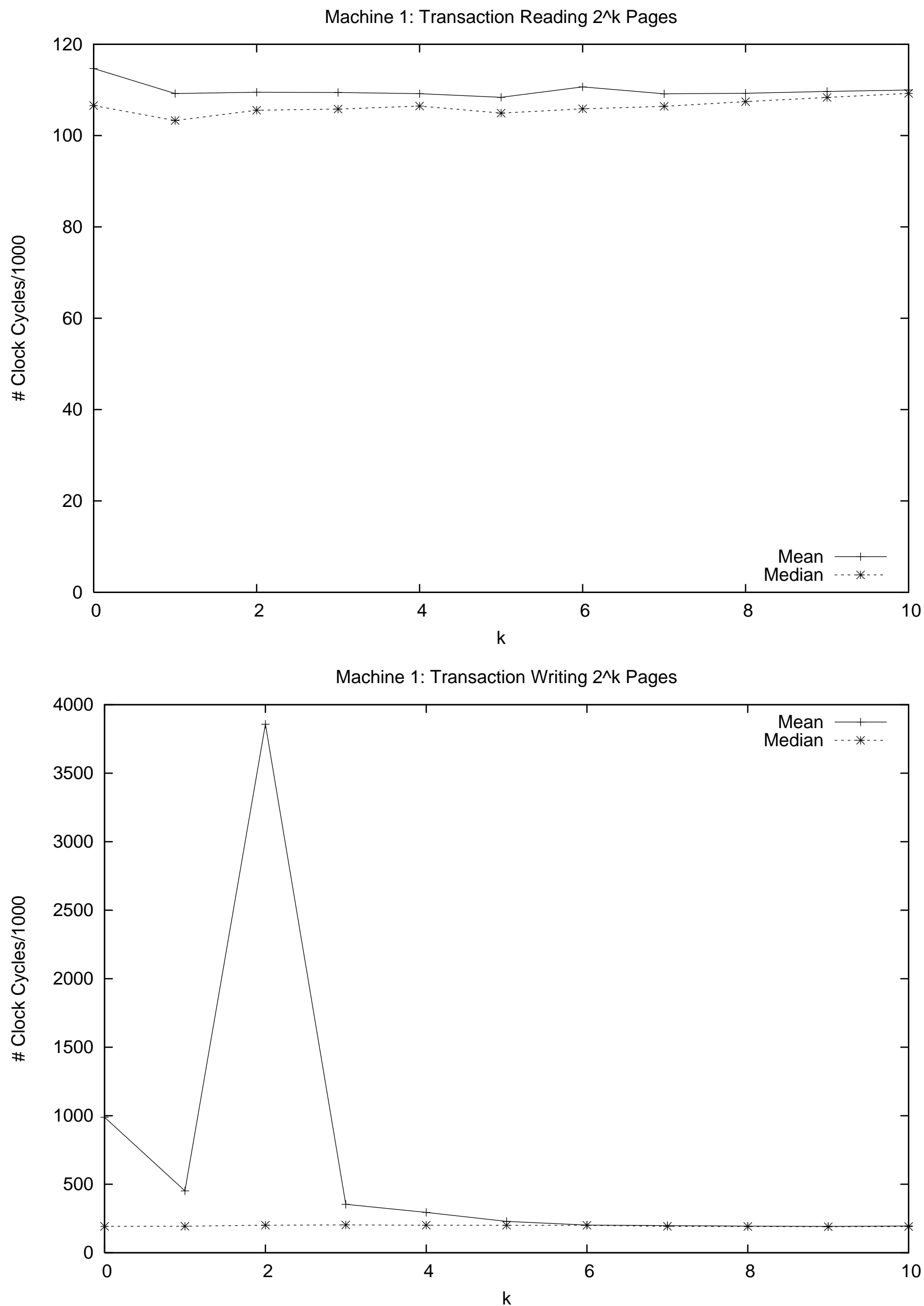


Figure 4-2: Average time per page to execute the transactions shown in Figure 4-1 on Machine 1. For each value of n, each transaction was repeated 1000 times.

```
// Transaction that reads
//   from n pages
int* x = xMmap("input.db",
               n*PAGESIZE);
int i, value;
   ...

xbegin();
for (i = 0; i < n; i++) {
  int j = i*PAGESIZE/sizeof(int);
  setPageAccess(&(x[j]),
                READ);
  value = x[j];
}
xend();
```

(a)

```
// Transaction that writes
//   to n pages
int* x = xMmap("input.db",
               n*PAGESIZE);
int i;
   ...

xbegin();
for (i = 0; i < n; i++) {
  int j = i*PAGESIZE/sizeof(int);
  setPageAccess(&(x[j]),
                READ_WRITE);
  x[j] = i;
}
xend();
```

(b)

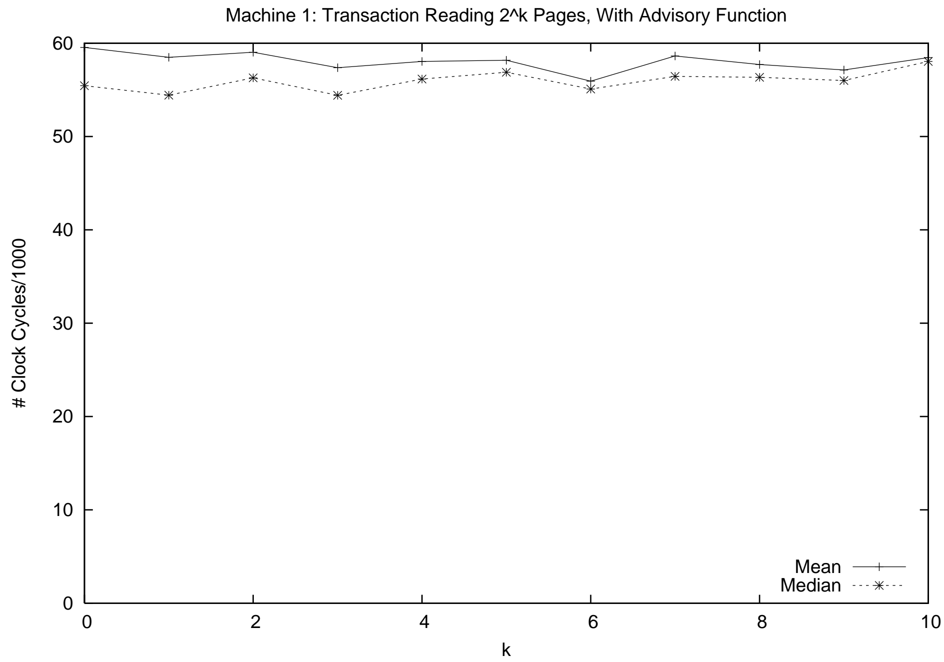
Figure 4-3: The transactions in Figure 4-1 written with the advisory function. The transaction in (a) reads from n consecutive pages, the transaction in (b) writes to n consecutive pages.

replaces the triggering and handling of a SIGSEGV with a function call.

Figure 4-3 exhibits the programs from the page-touch experiment, modified to call the advisory function before each page access. Figure 4-4 plots the average access times per page as n varies from 1 to 1024 on Machine 1. The advisory function reduced the average time per page read and write from 110,000 to about 60,000 clock cycles and from 200,000 to just under 100,000 clock cycles, respectively. Thus, on Machine 1, the advisory function cut down the per-page overhead by almost a factor of 2. Also, with the advisory function, the plots do not have any sharp peaks, suggesting that eliminating the need to handle SIGSEGVs reduced the variability of the experiment.[2]

Table 4.2 summarizes the average access times per page for all machines for $n = 1024$. The advisory function improved performance on Machine 1 by approximately a factor of 2, but the improvement was less significant on the other machines. For example, on Machine 3, the speedup was only 26% and 20% for page reads and writes, respectively. One explanation that I discuss in Section 4.2.3 is that the cost of entering a SIGSEGV handler appears to be particularly expensive on Machine 1

[2]Similar plots for other machines are shown in Appendix C.2.



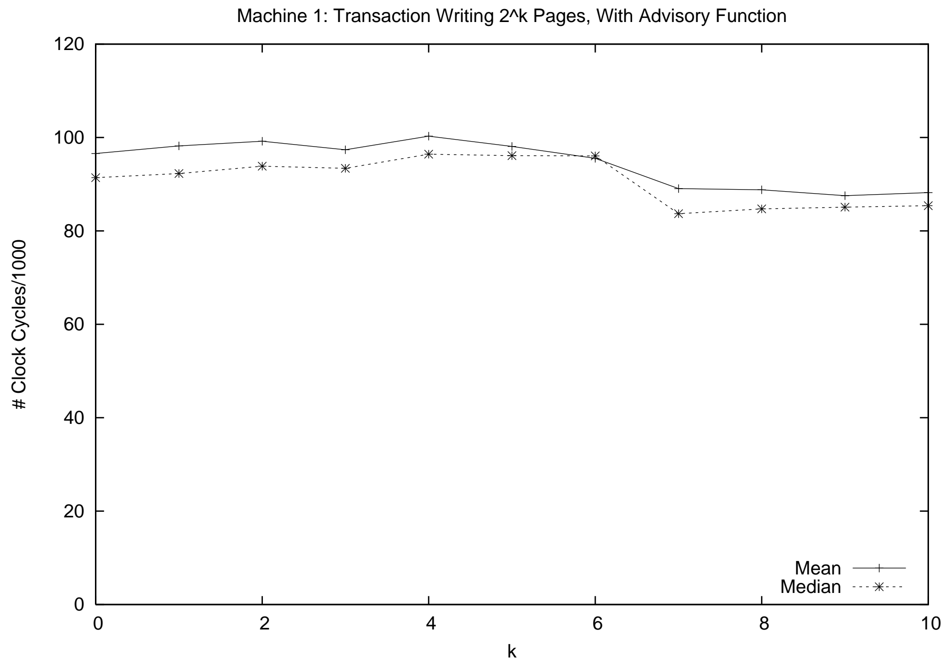


Figure 4-4: Average time per page to execute the transactions shown in Figure 4-3 on Machine 1. For each value of n, each transaction was repeated 1000 times.

Machine #	t_{read} Normal	With Adv.	% Speedup	t_{write} Normal	With Adv.	% Speedup
1	110.0	58.5	47%	195.5	88.2	55%
2	44.4	31.1	31%	91.6	63.4	31%
3	21.7	16.0	26%	43.5	34.8	20%
4	16.1	10.1	37%	149.4.	143.8	4%

Table 4.2: Average # of clock cycles per page access for transactions touching 1024 pages, with and without the advisory function. Numbers are in thousands of cycles. Percent speedup is calculated as $100\left(\frac{\text{Normal - With Adv}}{\text{Normal}}\right)$.

compared to the other machines. The speedup for writes on Machine 4 was only 4%. Since Machine 4 has only 64 MB, it is possible that this experiment no longer fits into main memory. Machine 4's slow processor speed might be another factor.

Converting the time per page read from clock cycles to microseconds, we find that a, the time per page read, ranges from 15 μs on Machine 3 to 54 μs on Machine 4. On all but Machine 4, b, the time per page write, is slightly less than $2a$.

4.2.3 Decomposing the Per-Page Overhead

In this section, I attempt to account for the per-page overheads observed in Sections 4.2.1 and 4.2.2 by estimating the times required for individual runtime operations. The experimental results suggest that in most cases, over half the overhead can be explained by the time required to handle a `SIGSEGV` signal, to call `mmap`, to copy a page on a transaction write, and the time to handle a page fault. I conjecture that most of the remaining time is spent handling cache misses and additional page faults.

Memory Mapping and Fault Handlers

As I described in Section 3.1, the runtime detects which pages a transaction accesses by mapping pages with no-access or read-only memory protections and handling `SIGSEGV` signals. I describe results from an experiment that measured the time required for the three expensive operations in this process: entering the fault handler, calling `mmap`, and exiting the handler.

Machine #	Entering `SIGSEGV` Handler (Cycles)	Exiting `SIGSEGV` Handler (Cycles)	`mmap` Clock Cycles	μs
1	32,216	29,323	15,156	5.05
2	8,032	9,723	10,054	4.19
3	3,489	4,745	3,228	2.31
4	3,078	3,140	2,282	7.61

Table 4.3: Number of clock cycles required to enter `SIGSEGV` handler, call `mmap`, and exit handler (average of 10,000 repetitions).

In the experiment, I ran a loop that captures the basic actions that the runtime executes when a transaction tries to read from or write to a new page for the first time. The times required for the loop operations give a lower bound on the overhead of a nondurable transaction. Starting with a memory-mapped file that is initially unmapped, I timed 10,000 repetitions of the following loop:

1. Attempt to increment the first `int` stored on the first page of the file. Measure the time required to enter the `SIGSEGV` handler for this operation.

2. Inside the `SIGSEGV` handler, `mmap` the same page with read-write protection. Record the time for this operation.

3. Measure the time required to exit the `SIGSEGV` handler.

4. Unmap the first page in the file and repeat.

Table 4.3 reports the average number of clock cycles required to do these operations on each machine. The main observation is that entering and exiting fault handlers and the `mmap` system call all took several thousand to tens of thousands of clock cycles. These times are one or two orders of magnitude greater than a cache miss that takes a hundred clock cycles. Of the three modern machines, Machine 1, the Pentium 4 required the most time for these operations. In summary, a transaction must incur at least several microseconds in overhead for every new page accessed.

Although I present only average values here, there is some variability in the measured values. For example, on Machines 2 and 3, the maximum time for an `mmap`

Machine	Avg. # of Clock Cycles	σ	Time in μs
1	16,851	945	5.6
2	5,310	139	2.2
3	3,311	211	2.3
4	3,995	1,133	13.3

Table 4.4: Clock cycles required to write to a page for the first time after memory mapping that page. Each experiment was repeated 5 times.

operation was about 5 times the average value. See Table C.3 in Appendix C for more complete statistics for this experiment.

First Access to an `mmap`ed Page

Linux does not actually bring an `mmap`'ed page into memory until it is first accessed. The time to handle this page fault represents another expensive component of non-durable transactions. In this experiment, I `mmap`'ed a single page of a file with read/write protection and then recorded the time required to increment the first integer on the page for the first time. Table 4.4 shows that these times are comparable to the time to enter and exit a fault handler.

Test of `memcpy`

Before a transaction writes to a page for the first time, Libxac first makes a copy of the old page. In this experiment, I measured the time required for a `memcpy` between two 4K character arrays. In Table 4.5, the first value is the time for the program's first `memcpy`, the second is the average time of the 2nd through 5th `memcpy`'s, and the third is the average time for 1,000 `memcpy`'s between the two arrays.[3]

The data suggests that optimizing the Libxac runtime to avoid cache misses and page faults could significantly improve performance. For the first two machines, a `memcpy` between two pages that are already in cache (the third value) costs approximately an order of magnitude less than the `mmap` from Table 4.3. The difference is less for the other two machines, but a `memcpy` is still faster than an `mmap`. On the

[3] The first and second values are averages from 5 repetitions of the experiment. The third is an average from 1000 repetitions of the experiment. See Table C.4 for detailed data.

Machine	First memcpy	Avg. of Ops 2 through 5	Avg. of 1000 Ops.
1	44,067	2,493	1,122
2	19,497	1,718	1,052
3	9,578	676	608
4	7,324	3,320	932

Table 4.5: Clock cycles required for a `memcpy` between two 4K character arrays in memory.

other hand, a `memcpy` when the arrays are not cached is a factor of about 2 to 3 more expensive than an `mmap` call.

Predicting Per-Page Overhead

I use the data in Tables 4.3, 4.4, and 4.5 to try explain the per-page overheads observed in the page-touch experiments from Sections 4.2.1 and 4.2.2.

For transaction (a) in Figures 4-1 and 4-3, recall that every time a transaction tries to read from a page for the first time, the runtime must handle a `SIGSEGV` and change the memory map from a page mapped as no-access to a (possibly different) page mapped read-only. At least one page fault occurs when the transaction actually reads from the page. Finally, when the transaction commits, the memory protection on that page must be changed back to no-access. Thus, transaction (a) involves handling one `SIGSEGV`, two calls to `mmap` and one page fault. Using the advisory function eliminates the cost of entering and exiting the fault handler.

On a given machine, let $t_{\text{segvEnter}}$ and t_{segvExit} be the average time to enter and exit a `SIGSEGV` handler, respectively, and let t_{mmap} and $t_{\text{pageFault}}$ be the average times to call `mmap` and to handle a page fault, respectively. Then, based on this model, a lower bound on t_{read} and t'_{read}, the time to read a new page inside a transaction, without and with the advisory function, is[4]

$$\begin{aligned} t_{\text{read}} &\geq t_{\text{segvEnter}} + t_{\text{segvExit}} + 2t_{\text{mmap}} + t_{\text{pageFault}} \\ t'_{\text{read}} &\geq 2t_{\text{mmap}} + t_{\text{pageFault}} \end{aligned} \tag{4.1}$$

[4]To get a more accurate lower bound, I actually subtract $t_{\text{timerDelay}}$, the delay in our timer, once from every measured value measured (in this case, $5t_{\text{timerDelay}}$).

Machine #	t_{read} Est.	Actual	Δ	% Diff.	t_{write} Est.	Actual	Δ	% Diff.
1, no Adv.	108.2	110.0	1.8	**1.6%**	185.7	195.5	9.8	**5.0%**
1, w Adv.	46.9	58.5	11.6	19.8%	47.9	88.2	40.3	45.7%
2, no Adv.	43.2	44.4	1.3	**2.8%**	72.0	91.6	19.6	**21.3 %**
2, w. Adv.	25.4	31.1	5.7	18.2%	26.5	63.4	26.5	58.2 %
3, no Adv.	18.0	21.7	3.8	**17.4%**	30.0	43.5	13.5	**31.0 %**
3, w. Adv.	9.7	16.0	6.3	39.3%	10.3	34.8	24.4	70.3 %
4, no Adv.	14.6	16.1	1.6	**9.7%**	23.9	149.4.	125.6	**84.0 %**
4, w. Adv.	8.4	10.1	1.7	16.4%	9.3	143.8.	134.5	93.5 %

Table 4.6: Average # of clock cycles per page access for transactions touching 1024 pages. All numbers are in thousands of clock cycles.

When a transaction writes to a page, the runtime handles two `SIGSEGV`'s: the first for reading the page, and the second to copy the page and `mmap` the new copy with read/write protection. Thus, a page write requires one extra fault handler, an additional call to `mmap`, and one `memcpy`.[5] Using the advisory function eliminates the entering and exiting of the `SIGSEGV`, and also one call to `mmap`, as the runtime does not need to map the original version of the page as read-only first. Thus, I estimate t_{write} and t'_{write}, the time required for a transaction to write to a new page, without and with the advisory function, as

$$\begin{aligned} t_{\text{write}} &\geq 2t_{\text{segvEnter}} + 2t_{\text{segvExit}} + 3t_{\text{mmap}} + t_{\text{pageFault}} + t_{\text{memcpy}} \\ t'_{\text{write}} &\geq 2t_{\text{mmap}} + t_{\text{pageFault}} + t_{\text{memcpy}} \end{aligned} \tag{4.2}$$

After repeating the transactions in Figures 4-1 and 4-3 1,000 times, with $n = 1024$, I obtained the data shown in Table 4.6. I use the average values from Table 4.3 as estimates for $t_{\text{segvEnter}}, t_{\text{segvExit}}$, and t_{mmap}, and I use the data in Table 4.4 and 4.5 as estimates for $t_{\text{pageFault}}$ and t_{memcpy}, respectively.

On modern machines, a majority of the overhead for a nondurable transaction comes from handling the `SIGSEGV` signal, calling `mmap`, copying the page, and handling page faults. On Machines 1, 2, and 4, the lower bound in (4.1) accounted for all but 10% of t_{read}. The bound in (4.2) was less accurate for predicting t_{write}, however.

[5]For t_{memcpy}, I use the value the smallest value, value 3, from Table 4.5.

Generally, the predictions were most accurate for Machine 1, which has the greatest overhead in clock cycles for the fault handler and the `mmap` operations. With the exception of t_{write} on Machine 4, the predictions for t_{read} and t_{write} account for at least 50% of the time spent per page access.

The lower bounds are even less successful for predicting the overheads when the advisory function is used. Equation (4.1) accounts for at least 50% of t'_{read}, but equation (4.2) significantly underestimates t'_{write}. One reason for this discrepancy is that Equations (4.1) and (4.2) assume the time spent updating control data structures is negligible. Although the computation required to modify the transaction state or maintain the consistency tree is not significant for a small transaction running on a single process, any cache misses or page faults incurred when accessing the control data structures may noticeably increase the overhead. For example, when running the transaction in Figure 4-1(a) with $n = 1$ on Machine 3, I noticed that the LIBXAC runtime spent 21,000 cycles inside the fault handler, but actually spent about 11,500 cycles adding the first page to the transaction's readset. Since this operation only involves adding an element to an empty list stored in an array, this operation is probably expensive only because a page was not cached in main memory.

I conjecture that most of the unexplained overhead is due to poor caching behavior. LIBXAC's data structures for a transaction's readset and writeset and for the transaction page tables are both implemented as large, fixed-size arrays. These data structures are unlikely to exhibit significant locality, since they are stored in a 59 MB control file. This phenomenon may be even more noticeable on Machine 4, which is only 300 MHz and has only 64 MB of RAM. With a more efficient implementation of the control structures, the discrepancies may be lower.

Although Equations (4.1) and (4.2) underestimate the overhead, for small transactions that fit into memory, the simple linear performance model of $aR + bW$ should still be reasonable. Failing to account for all the page faults or cache misses should only increase the constants a and b.

Machine #	Avg. Time (μs) Page Read	Page Write
1, no Adv.	39	1,569
1, w. Adv.	22	1,297
3(a), no Adv.	15	235
3(a), w. Adv.	14	179
3(b), no Adv.	16	227
3(b), w. Adv	15	182

Table 4.7: Average Access Time (μs) per Page, for Transactions Touching 1024 Pages.

4.3 Durable Transactions

In this section, I attempt to construct a performance model for durable transactions and test this model by repeating some of the experiments described in Section 4.2 for durable transactions.[6] Durable transactions incur extra overhead when they commit because the runtime must synchronously force data to the log file on disk using an `fsync`. This disk write should add both a large constant overhead to all transactions for the latency of accessing disk, and a roughly constant overhead per page written for updating the log file. Thus, I conjecture that a small durable transaction that reads from R pages and writes to W page incurs an additive overhead of $aR+bW+c$, where a is tens of microseconds, b is hundreds to a few thousand microseconds, and c is between 5 and 15 milliseconds.

Page Touch Experiments

Table 4.7 presents data from page-touch experiments for durable transactions, when $n = 1024$.[7] Since each transaction runs for tens to hundreds of milliseconds, the latency of accessing disk is not significant and I can use this data to construct slight over-estimates of a and b.

From the data, we see that the per-page overhead for nondurable and durable read-only transactions are roughly the same. Since `fsync` for a read-only transaction

[6]For these tests, I disabled the write-cache to ensure that `fsync` actually writes all data to disk. I omit results from Machine 2, because I had insufficient permissions to disable the write-cache.

[7]Table C.5 gives a more complete version of this table.

writes only meta-data to disk, the latency to access disk is amortized over 1024 pages. The time required per page write, however, increases by several orders of magnitude. Machine 1, which only has SATA drives, performs page writes 6 or 7 times slower than Machine 3, which has SCSI drives. It is possible, however, that other effects may explain this difference.

Synchronizing a File

To estimate c, the latency of a write to disk, I measured the time required to synchronize a memory-mapped file after modifying a one randomly selected page. This benchmark repeats the following loop:

1. Pick a page uniformly at random from the 10,000 page memory-mapped file. Increment the first `int` on that page.[8]

2. Measure the time for an asynchronous `msync` on that page in the file.

3. Measure the time for an `fsync` on the file.

The time required for the `msync` operation was on average less than 10 μs on all machines. In Table 4.8, I report the times required to do `fsync`. The average time for to write a page out to disk seems to be between 5 and 15 ms on average. Again, the harddrives on Machine 3 are faster than on Machine 1. The maximum time required for an `fsync` is 0.8 seconds on Machine 3, and almost 6 seconds on Machine 1. This result is consistent with the fact that the operating system does not provide any guarantees on the worst-case behavior of system calls. Expensive operations occur, but only infrequently.

4.4 Testing Concurrency in Libxac

I conclude this chapter by describing several experiments that test LIBXAC's performance when executing independent transactions on two concurrent processes. In

[8] I had arbitrarily set the default size for LIBXAC's log files to 10,000 pages.

Operation	Mean	St. Dev	Min	Median	99th %tile	Max
1: `fsync`	**13.6**	184.3	2.4	8.0	12.7	5,836.5
3(a): `fsync`	**4.8**	24.0	0.7	4.0	7.0	761.6
3(b): `fsync`	**4.7**	23.1	0.8	3.9	6.9	731.9

Table 4.8: Time required to call `msync` and `fsync` on a 10,000 page file with one random page modified, 1000 repetitions. All times are in ms.

an experiment for nondurable transactions, the system achieved near-linear speedup when the work done on a page touched by a transaction was about two orders of magnitude greater than the overhead of accessing that page. The same program did not exhibit significant concurrency with durable transactions, however, most likely because the writes to disk during a transaction commit represents a serial bottleneck.

Test Programs

Figure 4-5 shows the transactions used to test the concurrency of LIBXAC. The single-process version of Test A executes a simple transaction 10,000 times, while the two-process version runs 5,000 independent transactions on each process.[9] Since Test A does little work incrementing the first integer on a page, I also tested two other versions of this program, shown in Figure 4-6. Test B increments every integer on the page inside the transaction, and Test C repeats B's loop 1000 times. In Figure 4-6, I only show the single-process version, as the two-process version is similar.

Nondurable Transactions

Table 4.9 exhibits the results with nondurable transactions for Tests A, B, and C.[10] From this data, we can make several observations. First, there is no significant speedup on Machines 1 or 4. Since both machines have a single processor, this result is not surprising. The slight speedup for Tests A and B on Machine 1 may be due to the Pentium 4's hyperthreading.

Machine 2, which has 2 processors, exhibited speedup on all three tests, ranging

[9] Although this code does not show it, I use the advisory function for these concurrency tests.
[10] See Table C.9 for more detailed data.

Test A

Single process version:

```
for (i = 0; i < 10000; i++) {
  xbegin();
    x[0]++;
  xend();
}
```

Two process version:

Process 1

```
for (i = 0; i < 5000; i++) {
  xbegin();
    x[0]++;
  xend();
}
```

Process 2

```
for (i = 0; i < 5000; i++) {
  xbegin();
    x[PAGESIZE/sizeof(int)]++;
  xend();
}
```

Figure 4-5: Concurrency Test A: Each transaction increments the first integer on a page 10,000 times.

Test B

```
xbegin();
for (j = 0;
     j < PAGESIZE/sizeof(int);
     j++) {
  x[j]++;
}
xend();
```

Test C

```
xbegin();
for (k = 0; k < 1000; k++) {
  for (j = 0;
       j < PAGESIZE/sizeof(int);
       j++) {
    x[j]++;
  }
}
xend();
```

Figure 4-6: Concurrency Tests B and C: Test B increments every integer on the page. Test C repeats the transaction in Test B 1,000 times. I omit the outermost for-loop, but as in Figure 4-5, each transaction is repeated 10,000 times.

Machine	Test	Mean Time per Xaction 1 proc.	2 proc.	Speedup
2	A	26.2	23.0	1.14
2	B	28.3	24.2	1.16
2	C	1,786	903	1.98
3(a)	A	22.9	24.3	0.94
3(a)	B	28.1	27.5	1.02
3(a)	C	2,259	1,132	2.00
3(b)	A	24.3	24.9	0.98
3(b)	B	28.2	26.3	1.07
3(b)	C	2,248	1,130	1.99

Table 4.9: Concurrency tests for nondurable transactions. Times are μs per transaction. Speedup is calculated as time on 1 processor over time on 2 processors.

from about 12% for Test A to almost 50% is Test C. On the other hand, Tests A and B actually run faster on one process than on two on Machine 3. Test C, which performs a lot of work on one page, does manage to achieve near-perfect linear speedup. It is unclear whether the differences between Machines 2 and 3 are due to the different architectures or due to some other factor.

These results suggest that on Machine 3, LIBXAC only exhibits concurrency for independent transactions if each transaction does significantly more work per page than the overhead for a page access. Thus, the prototype implementation of LIBXAC may not be efficient for small concurrent nondurable transactions.

Durable Transactions

Table 4.10 presents the data for Tests A, B, and C, repeated with durable transactions. For a durable transaction, the cost of forcing data out to disk at a transaction commit is significant. These results are consistent with the data from Table 4.8 for the times required to complete `fsync` on the various machines.

In these experiments, we do not observe any speedup on the multiprocessor machines. Since we are running transactions using only one disk, the `fsync` is likely to be a serialization point. Thus, it may not be possible to achieve significant speedup with only one disk without an implementation that supports group commits, i.e.,

Machine	Test	Mean Time per Xaction		Speedup
		1 proc.	2 proc.	
3(a)	A	6.12	6.16	0.99
3(a)	B	6.11	6.20	0.99
3(a)	C	6.32	6.63	0.95
3(b)	A	6.21	6.26	0.99
3(b)	B	6.22	6.21	1.00
3(b)	C	6.39	6.62	0.97

Table 4.10: Concurrency tests for durable transactions. Times are milliseconds per transaction.

committing multiple transactions on different processes with the same synchronous disk write. One possible reason for observing slowdown is that having multiple processes accessing the same log file simultaneously may cause slightly more disk head movement compared to having a single process access the file.

Chapter 5

Search Trees Using Libxac

In this chapter I describe how memory-mapped transactions can be used in a practical application, specifically in search trees that support concurrent searches and insertions. I also present experimental results comparing the performance of search trees written using LIBXAC to the B-tree of Berkeley DB [44], a high-quality transaction system.

For data that resides on disk, B-trees are the canonical data structure for supporting dictionary operations (search, insertion, deletion, and range queries). In Section 5.1, I describe the Disk Access Machine Model, the performance model primarily used to analyze B-trees. In this model, computation is free, but moving a block between main memory and disk has unit cost. I then describe the concept of a cache-oblivious algorithm, an algorithm that tries to minimize the number of memory transfers per operation without knowing the actual block size. Finally, I briefly describe how a cache-oblivious B-tree (CO B-tree) supports dictionary operations with an asymptotically optimal number of memory transfers.

In Section 5.2, I investigate the practical differences between two different B-tree variants by presenting experimental results comparing the performance of a serial B^+-tree and a serial CO B-tree, both written using memory mapping, but without LIBXAC.[1] The data demonstrates that a CO B-tree can simultaneously support ef-

[1]Sections 5.1 and 5.2 describes joint work with Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul.

ficient searches, insertions, and range queries in practice. Random searches on the CO B-tree ran only 3% slower than on a tuned B$^+$-tree on one machine and ran 4% faster on a newer machine.

In Section 5.3, I describe the ease of using memory-mapped transactions to convert the serial implementations of the B$^+$-tree and CO B-tree into parallel versions.

I present experimental results in Section 5.4 that suggest that small, durable memory-mapped transactions using Libxac are efficient. In an experiment where a single process performed random insertions, each as a durable transaction, the Libxac B$^+$-tree and CO B-tree are both competitive with Berkeley DB. On the three newer machines, the performance of the B$^+$-tree and CO B-tree ranged from being 4% slower than Berkeley DB to actually being 67 % faster. This result is quite surprising, especially in light of the fact that I am comparing an unoptimized prototype with a sophisticated, commercial transaction system.

Finally, in Section 5.5, I conclude by describing possible future experiments for evaluating the performance of Libxac. I also discuss potential improvements to the implementation that are motivated by the experimental results.

5.1 Introduction

The DAM Model

In today's computer systems, there is significant disparity between the time required to access different memory locations at different levels of the memory hierarchy. In Chapter 4, we saw examples of this phenomenon. A single clock cycle on the newer test systems is less than 1 nanosecond. A `memcpy` between two arrays in memory takes a few microseconds, while using `fsync` to force data out to disk typically requires several milliseconds. The rotational latency of a 10,000 rpm disk is 6 ms, creating a lower bound on the worst-case time to read data from disk. Because the time to access disk is at least 6 orders of magnitude larger than the time for a single clock cycle, the cost of actual computation for a program that performs many disk accesses can often

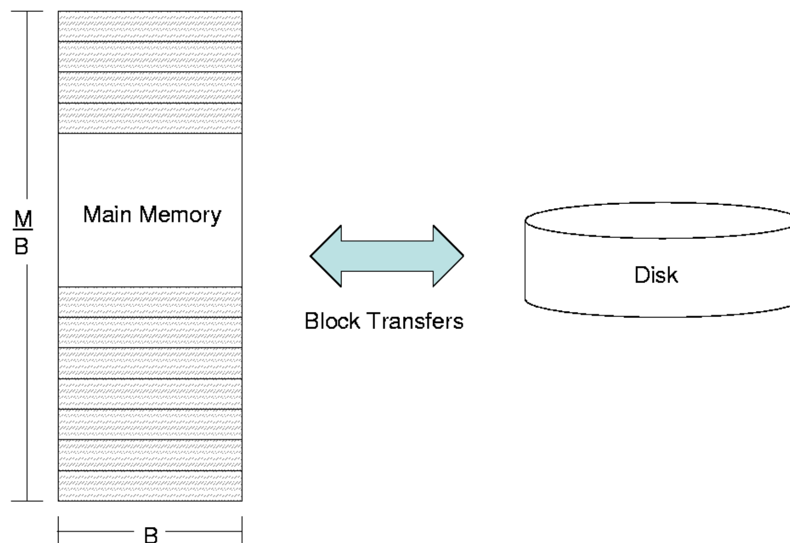


Figure 5-1: An illustration of the Disk Access Machine (DAM) model.

be ignored. Instead, the performance model traditionally used to analyze programs that access large data sets on disk is the *Disk-Access Machine* (DAM) model [3]. The DAM model assumes a system has two levels of memory, as illustrated in Figure 5-1. Main memory has size M, disk has infinite capacity. In this model, computation on data in main memory is free, but each transfer of a size-B block between main memory and disk has unit cost.

Using the DAM model we can analyze the cost of doing a single query on a B$^+$-tree. A B-tree can be thought of as a normal binary-search tree, except with a branching factor of $\Theta(B)$ instead of 2. A search on a B-tree storing N keys requires $O(\log_B N)$ block transfers: a constant number of transfers at every level of the tree. An information-theoretic argument proves a lower bound on the worst-case time for a dictionary operation of $\Omega(\log_B N)$. Thus, searches on B-trees use an asymptotically optimal number of memory transfers. A B$^+$-tree is similar to a B-tree, except that the data is stored only at the leaves of the tree, minimizing the number of block

transfers by putting as many keys in a single block as possible. The fact that B-trees or variants of B-trees are widely used in practice corroborates the validity of the DAM model.

Cache-Oblivious B-Trees

The optimality of the B$^+$-tree in the DAM model requires that the implementation know the value of B. Unfortunately, in a real system, it is not always clear what the exact value of B is. For example, on a disk, the cost of accessing data in a block near the current position of the disk head is cheaper than accessing a block on a different track. There are multiple levels of data locality: two memory locations may be on the same cache line in L1 cache, the same line in L2 cache, the same page in memory, the same sector on disk, or the same track on disk. In a real system, there may not be a single "correct" block size B.

An alternative to the DAM model is the cache-oblivious model of computation [16, 39]. An algorithm is said to be *cache-oblivious* if it is designed to minimize the number of memory block transfers *without knowing the values of B or M.* A fundamental result for cache-oblivious algorithms is that any algorithm that performs a nearly optimal number of block transfers in a two-level memory model without knowing B and M also performs a nearly optimal number of memory transfers on any unknown, multilevel memory hierarchy [39].

A cache-oblivious B-tree (CO B-tree) [4] is a search tree structure that supports dictionary operations efficiently. The CO B-tree guarantees the following bounds on dictionary operations without needing to know the exact value of B:

1. Search: $O(\log_B N)$ memory transfers.
2. Range queries of R elements: $O(\log_B N + R/B)$ memory transfers.
3. Insertions and deletions: $O(\log_B N + \log^2 N/B)$ memory transfers.

The bounds for searches and range queries are asymptotically optimal. A CO B-tree achieves these bounds by organizing the tree in a *van Emde Boas layout* [39]. This

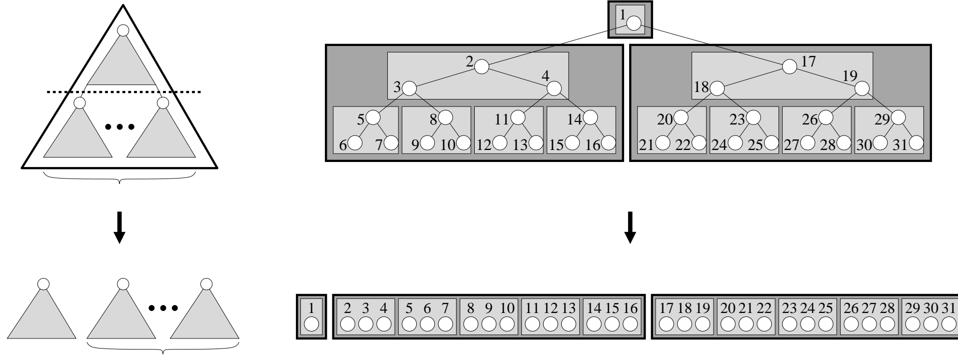


Figure 5-2: The van Emde Boas layout (left) in general and (right) of a tree of height 5.

layout is a binary tree recursively laid out in memory. A tree with N nodes and height h can be divided into one root tree with $\Theta(\sqrt{N})$ nodes and height approximately $h/2$, and $\Theta(\sqrt{N})$ child subtrees, each also with $\Theta(\sqrt{N})$ nodes and height $h/2$. In the van Emde Boas layout, each of these height $h/2$ subtrees is stored contiguously in memory, with the layout recursively repeated for each height $h/2$ tree. Figure 5-2 illustrates this layout for trees of height 5.

Intuitively, this layout is cache-oblivious because for any block size B, we can recurse until our layout eventually gets to a tree of height approximately $\Theta(\lg B)$. This tree fits entirely into one block, so any query from root to leaf in the original tree visits $O(\lg N)/\Theta(\lg B)$, or $O(\log_B N)$ blocks.

The van Emde Boas layout is sufficient for a static dictionary that does not support insertions or deletions. One method for creating a dynamic tree is to use this static tree as an index into a packed memory array [4]. A packed memory array stores N elements in sorted order in $O(N)$ space. The array leaves carefully spaced gaps in between elements and carefully maintains these gaps to satisfy certain density thresholds. These threshold ensure that large rearrangements of the array are be amortized over many insert or delete operations.

I have only sketched the details the the CO B-tree here. For a more thorough presentation of this data structure, I refer the reader to [4].

5.2 Serial B$^+$-trees and CO B-trees

In this section,[2] I present several experimental results that show the CO B-tree is competitive with the B$^+$-tree for dictionary operations. When doing random searches on a static tree, the CO B-tree ran 3% slower than the B$^+$-tree with the best block size on one machine and 4% faster than the B$^+$-tree on another machine. For dynamic trees, we observe that as the block size increases, the time to do random insertions in the B$^+$-tree increases, but the time for range queries and searches decreases. The CO B-tree is able to efficiently support all three operations simultaneously.

These experiments were conducted on Machine 3, which has 16 GB of RAM, and on Machine 4, which has only 128 MB of RAM.[3]

Random Searches on Static Trees

The first experiment performed 1000 random searches on a B$^+$-tree and a CO B-tree. On Machines 3 and 4, these static trees had 2^{29} and 2^{23} keys, respectively. These sizes were chosen to be large enough to require the machine to swap. For the B$^+$-tree, we tested block sizes ranging from 4 KB to 512 KB. In this test, we flushed the filesystem cache by unmounting and then remounting the file system before the first search.

From the results in Table 5.1, we see that the CO B-tree is competitive on Machine 4: the B$^+$-tree with the best block size only outperformed the CO B-tree by 3 %. For Machine 3, the CO B-tree was 4% faster than the B$^+$-tree with the best block size. This data also hints at the slight difficulty in finding the right block size B for the B$^+$-tree. On Machine 3, the best block size was 256 KB, while on Machine 4 it was 32 KB. Both values are significantly larger than the default operating system page size of 4 KB. For each machine, the B$^+$-tree needed to be tuned to find the optimal block size, while the CO B-tree was efficient without tuning.

[2] This section describes joint work with Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul

[3] These machines are described in Section 4.1. At the time of this test, however, Machine 3 was running a 2.4 kernel.

Data structure	Average time per search	
	Machine 4: small	Machine 3: big
CO B-tree	12.3ms	13.8ms
Btree: 4KB Blocks:	17.2ms	22.4ms
16KB blocks:	13.9ms	22.1ms
32KB blocks:	11.9ms	17.4ms
64KB blocks:	12.9ms	17.6ms
128KB blocks:	13.2ms	16.5ms
256KB blocks:	18.5ms	14.4ms
512KB blocks:		16.7ms

Table 5.1: Performance measurements of 1000 random searches on static trees. Both trees use 128-byte keys. In both cases, we chose enough data so that each machine would have to swap. On the small machine, the CO B-tree had 2^{23} (8M) keys for a total of 1GB. On the large machine, the CO B-tree had 2^{29} (512M) keys for a total of 64GB.

Dynamic Trees

The next experiment tested dynamic trees on the smaller machine, Machine 4. We compared the time to insert 440,000 and 450,000 random elements for the CO B-tree, respectively. We chose these data points because 450,000 is the point right after the CO B-tree must reorganize the entire data structure. We also compared this data to B$^+$-trees with different block sizes. For the B$^+$-tree experiments, allocation of new blocks was done sequentially to improve locality on disk. This choice represents the best possible behavior for the B$^+$-tree. In a real system, as the data structure ages, the blocks become dispersed on disk, possibly hurting performance. Finally, we compared this data to random insertions done into a Berkeley DB database using the `db_load` command. The buffer pool size for Berkeley DB was set to 64 MB.

The data in Table 5.2 demonstrates that the CO B-tree performs well, even at the pessimal point, just after reorganizing the entire array. For the B$^+$-tree, there is a tradeoff between small and large block sizes. Small block sizes imply that insertions are faster, but only at the cost of more expensive range queries and searches. The CO B-tree is able to efficiently support all three operations simultaneously. We did not observe any block size B where the B$^+$-tree was strictly better than the CO B-tree for all three operations.

Using Berkeley DB with the default tuning parameters, it took 20 minutes to load

	Block Size	insert random values	range query of all data	1000 random searches
CO B-tree 440,000 inserts		15.8s	4.6s	5.9s
CO B-tree 450,000 inserts		54.8s	9.3s	7.1s
B^+-tree				
Sequential	2K	19.2s	24.8s	12.6s
block	4K	19.1s	23.1s	10.5s
allocation:	8K	26.4s	22.3s	8.4s
(450,000	16K	41.5s	22.2s	7.7s
inserts)	32K	71.5s	21.4s	7.3s
	64K	128.0s	11.5s	6.5s
	128K	234.8s	7.3s	6.2s
	256K	444.5s	6.5s	5.3s
Random block allocation:	2K	3928.0s	460.3s	24.3s
Berkeley DB (default parameters):		1201.1s		
Berkeley DB (64 MB pool):		76.6s		

Table 5.2: Timings for memory-mapped dynamic trees. The keys are 128 bytes long. The range query is a scan of the entire data set after the insert. Berkeley DB was run with the default buffer pool size (256KB), and with a customized loader that uses 64MB of buffer pool. These experiments were performed on the small machine.

the data. Building a customized version of `db_load` with the buffer pool size set to 64 MB, however, we managed to improve Berkeley DB to run only 40% slower than the CO B-tree for insertions. Perhaps by tuning additional parameters, Berkeley DB could be sped up even further. Unfortunately, needing to optimize a large number of tuning parameters represents a disadvantage in practice.

In-Order Insertions

Finally, we ran an experiment testing the worst-case for the CO B-tree, when the data is inserted in order. Table 5.3 shows the time required to insert 450,000 elements in order into each search tree. In this case, the CO B-tree is about 65% slower than Berkeley DB. This behavior is reasonable considering we are testing the CO B-tree at a worst possible point.

In summary, these empirical results show that the performance of a serial CO B-tree for dictionary operations is competitive, and in some situations, actually faster than an optimally tuned B^+-tree or Berkeley DB.

Time to insert a sorted sequence of 450,000 keys	
Dynamic CO B-tree	61.2s
4KB Btree	17.1s
Berkeley DB (64MB)	37.4s

Table 5.3: The time to insert a sorted sequence 450,000 keys. Inserting sorted sequence is the most expensive operation on the packed memory array used in the dynamic CO B-tree.

5.3 Search Trees Using Libxac

This section explains how I parallelized the serial implementations of the B$^+$-tree and the CO B-tree tested in the previous section. This process was relatively painless and involved few changes to the existing code.

Parallelizing Search Trees Using Libxac

The serial implementations of a B$^+$-tree and a CO B-tree that I started with both stored data in a single file. Each tree opened and closed the database using `mmap` and `munmap`, respectively. I modified the open and close methods to use LIBXAC's `xMmap` and `xMunmap` instead. I supported concurrent searches and insertions by enclosing the search and insert methods between `xbegin` and `xend` function calls. In the implementation, no backoff method was specified; every transaction immediately retries after an abort until it succeeds.

Parallelizing these codes required only these few, simple changes. Table 5.4 gives a rough estimate of the size of the source code before and after modification with LIBXAC. Although counting the number of lines of code is, at best, an imprecise way to estimate code complexity, these numbers reflect the total programmer effort required to use LIBXAC. Less that two dozen LIBXAC function calls were required for each data structure. For the B$^+$-tree, 8 out of the 23 calls were actually optimizations, i.e., calls to the advisory function, `setPageAccess`. Also, most of the additional code for the B$^+$-tree was for testing concurrent insertions on the tree, not for supporting the data structure operations.

Because the conversion process was simple, I was able to successfully modify the CO B-tree structure in only a few hours, i.e., overnight. Since the CO B-tree was

Code	Serial Version	Using LIBXAC	LIBXAC Function Calls Added
B$^+$-tree	1122 lines	1527 lines	23
CO B-tree	1929 lines	2026 lines	17

Table 5.4: Changes in Code Length Converting B$^+$-tree and CO B-tree to Use LIBXAC.

previously implemented by another student [29], most of this time was spent actually understand the existing code. The serial B$^+$-tree was also coded by someone else.

My experience provides anecdotal evidence as to the ease of programming with LIBXAC. Using this library, it was possible to modify a complex serial data structure to support concurrent updates, knowing only a high-level description of the update algorithm. Unlike a program that uses fine-grained locks, the concurrency structure of the program with transactions is independent of the underlying implementation.

5.4 Durable Transactions on Search Trees

In this section, I describe experiments performing insertions on LIBXAC search trees, with each insertion as a durable transaction. On newer machines, I found that the search trees coded with LIBXAC were actually competitive with Berkeley DB's B-tree, running anywhere from 4% slower to 67% faster. Tables 5.5 and 5.6 summarize the results from this experiment. For more details on the experimental setup, see Section C.6.

On a single process, on an `ext3` filesystem (Machines 1 and 3(b)), the average time per insertion on the LIBXAC search trees was over 60% faster than on Berkeley DB. On the Reiser FS filesystem (Machine 3(a)), the LIBXAC search trees ran only 4% slower than Berkeley DB. These results demonstrate that durable memory-mapped transactions with LIBXAC can be efficient.

It is unclear exactly why Berkeley DB takes so long to perform random insertions as durable transactions. It is possible that I have not tuned Berkeley DB properly, or that I have not taken full advantage of its functionality. The fact that I cannot simply use Berkeley DB with default parameters is another argument in favor of

Machine	Search Tree	Avg. Time/Insert (ms) 1 Proc.	2 Proc.	% Speedup	Speedup
1	B^+-tree, w. adv.	18.3	14.6	20.2%	1.25
1	CO B-tree, no adv.	13.5	15.7	-16.3%	0.86
1	Berkeley DB	45.9	44.2	3.7%	1.04
3(a)	B^+-tree, w. adv.	7.7	7.7	0 %	1.00
3(a)	CO B-tree, no adv.	7.5	7.8	-4.0 %	0.96
3(a)	Berkeley DB	7.4	5.1	31.1 %	1.45
3(b)	B^+-tree, w. adv.	7.4	7.2	2.7%	1.03
3(b)	CO B-tree, no adv.	7.2	7.3	-1.4%	0.99
3(b)	Berkeley DB	22.4	17.7	21.0%	1.28
4	B^+-tree, w. adv.	82.0	–	–	–
4	CO B-tree, no adv.	66.5	–	–	–
4	Berkeley DB	57.7	–	–	–

Table 5.5: Time for 250,000 durable insertions into LIBXAC search trees. All times are in ms. Percent speedup is calculated as $\frac{100(t_1-t_2)}{t_2}$, where t_1 and t_2 are the running times on 1 and 2 processors, respectively.

Machine	B^+-tree vs. BDB 1 Proc.	2 Proc.	CO B-tree vs. BDB 1 Proc.	2 Proc.
1	60%	67%	71%	65%
3(a)	-4%	-51%	-1%	-53 %
3(b)	67%	59%	68%	59%
4	-42%	–	-15%	–

Table 5.6: The % speedup of LIBXAC search trees over Berkeley DB. Percent speedup is calculated as $\frac{100(t_L-t_B)}{t_B}$, where t_L and t_B are the running times on the LIBXAC and the Berkeley DB tree, respectively. Speedup is t_1/t_2.

simpler interfaces like the one provided by LIBXAC.

The LIBXAC search trees on Machine 3 achieve almost no speedup or slight slowdown going from one to two processes. These results are consistent with the previous data from the concurrency tests on durable transactions in Table 4.10: the simple transactions in concurrency Test A take about 6 ms on average, while the search tree inserts take about 8 ms. It is interesting that the B^+-tree achieves speedup about 20% speedup on Machine 1. One observation is that concurrency test A takes about 8 or 9 ms on Machine 1, while the B^+-tree inserts take about 18 ms. Thus, there may be more potential for speedup compared to Machine 3.

We can look a little more closely at the time required for individual inserts. Figure 5-3 plots the time required for the kth most expensive insert on Machine 3(a) and 3(b).

For all the search trees, only about 100 insertions require more than 100 ms. There is a sharp contrast between the LIBXAC search trees and Berkeley DB; the most expensive inserts for LIBXAC trees take over a second, while the most expensive inserts for Berkeley DB take on the order of a tenth of a second. The fastest inserts for LIBXAC tend to be faster than Berkeley DB however, taking on the order of a millisecond. The conclusion is that the Berkeley DB B-tree exhibits more consistent behavior than LIBXAC search trees, but on average the two systems are competitive.

Since LIBXAC relies more heavily on the operating system than Berkeley DB, the fact that some insertions with LIBXAC are expensive is not surprising. Also, since all results are real-time measurements, it is possible that some of these 100 expensive insertions include times when the program was swapped out for a system process.

Finally, although I do not present the detailed results here, I have observed that even when the write-cache on the harddrives are enabled, Berkeley DB and the LIBXAC search trees are comparable (see Appendix C, Table C.12). Although these transactions are not strictly recoverable, these results suggest that memory-mapped transactions using LIBXAC may still be efficient in other situations, (if our systems had harddrive caches with battery-backup, for example).

5.5 Summary of Experimental Results

In this chapter, I have presented experimental results testing the performance of search trees implemented with and without LIBXAC. The results in Section 5.2 show that a CO B-tree can simultaneously support efficient searches, insertions, and range-queries in practice. The CO B-tree is even competitive with a B-tree whose block size B has been carefully tuned. Section 5.4 shows that the LIBXAC B^+-tree and CO B-tree can support insertions as durable transactions efficiently. In the experiments I conducted, insertions using LIBXAC search trees ranged from being only 4% slower to

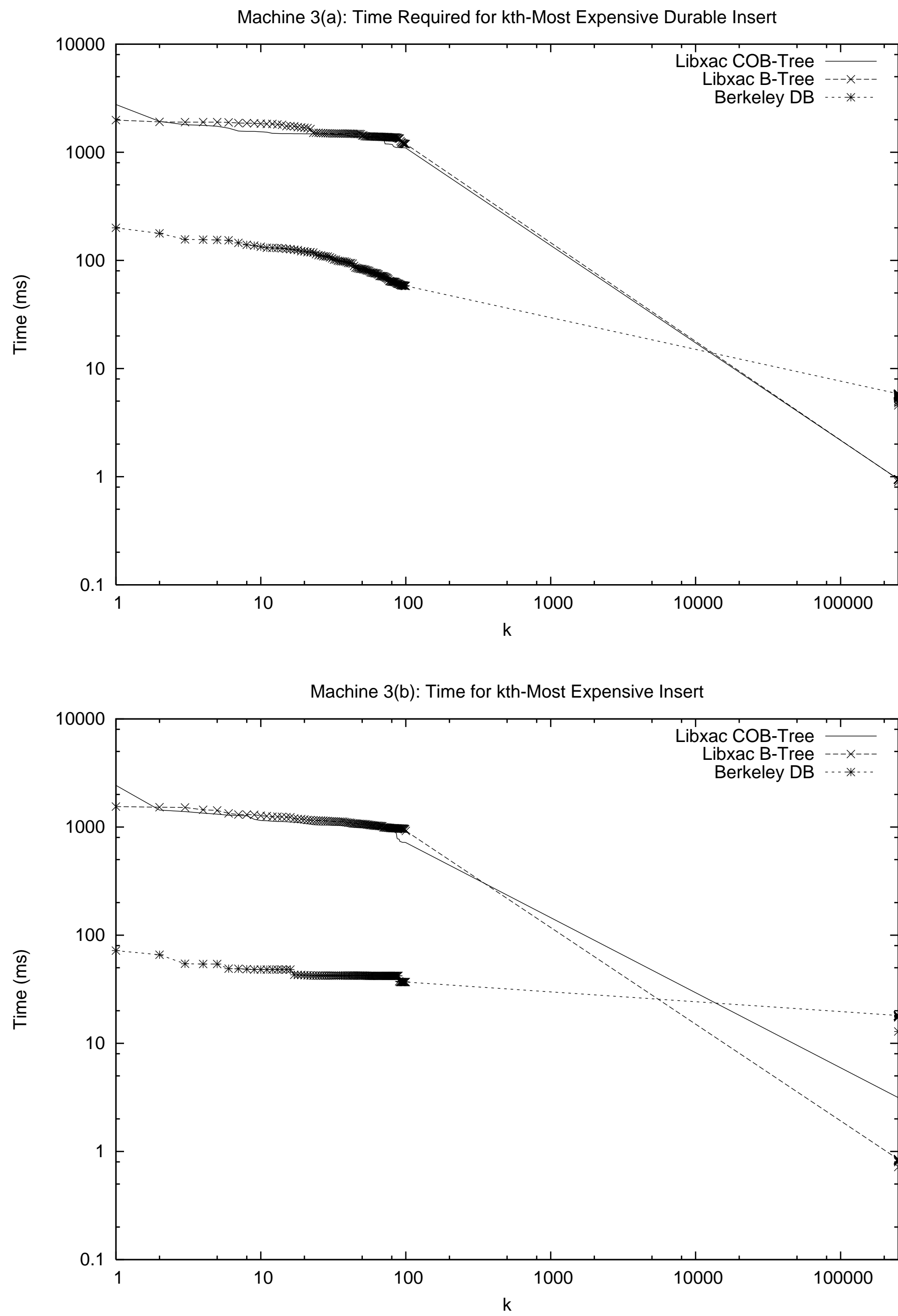


Figure 5-3: Machine 3: Time for kth most expensive insert operation.

about 67% faster than Berkeley DB. This last result is quite surprising, considering the fact that I am comparing an unoptimized prototype of LIBXAC with several significant flaws to a high-quality transaction system such as Berkeley DB.

Although this result is quite promising, I believe there is still significant work that needs to be done:

1. LIBXAC needs to be modified to fully support recovery on multiple processes, and a recovery process needs to be implemented and fully tested. Separating the log data and meta-data into separate files should accomplish this goal and not hurt performance if LIBXAC uses multiple disks, but it is impossible to know for sure without actual tests.

   In particular, one shortcut I took during implementation was to not calculate the checksum for each page during transaction commit. Appendix C, Table C.8 shows that calculating an MD5 checksum on a single page takes about 36,000 clock cycles on Machines 1 and 2 (about 12 to 15 μs). For small transactions that touch only a few pages, this cost seems fairly reasonable. For larger transactions, however, performing two calls to `fsync` to ensure that the commit record is written to disk after all the data pages is probably more efficient than computing a checksum. In some cases, I have noticed that the time for a second `fsync` is fairly small if it occurs soon after the first, possibly because the disk head does not move between the two writes. It would be interesting to more rigorously test whether performing two `fsync`'s during a transaction commit substantially impacts the performance of LIBXAC.

2. Berkeley DB supports group commits, i.e., allowing transactions on different threads or processes commit together with the same synchronous disk write. Modifying LIBXAC to support group commits may improve concurrency when the system uses a single disk.

3. The prototype currently limits a transaction's maximum size to around 64 or 128 MB.[4] Unfortunately, on the three newer machines, this constraint allows us

[4]In Linux, a process is allowed to have at most 2^{16} different `mmap`ed segments. The LIBXAC

to test only search trees that can fit into main memory. It would be interesting to test LIBXAC on large databases that do not fit into memory.

Another interesting experiment would be to test LIBXAC in a memory-competitive environment, with other applications running simultaneously.

4. Currently, LIBXAC maintains its logs at page granularity; the runtime saves a copy of every page that a transaction writes, even if the transaction modifies only a few bytes on a page. A single run of the experiment doing 250,000 insertions to a B^+-tree or CO B-tree, LIBXAC generates approximately 5 GB of log files. In contrast, Berkeley DB only creates 185 MB of log files. There is significant room for improvement in the way LIBXAC maintains its logs.

In addition to questions related to the LIBXAC implementation, there are also several theoretical questions that these experiments raise:

1. In all these experiments, I have used the oldest-wins abort policy with no backoff when transactions conflict. A backoff loop may improve performance for concurrent insertions in practice. It would be interesting to experiment with other policies for contention resolution, especially if the LIBXAC runtime is modified to be more decentralized.

2. Although I managed to "parallelize" the CO B-tree, it is unclear whether this data structure still performs an optimal number of memory-transfers per operation. For example, some CO B-tree insert operations must rebalance the entire packed memory array, leading to a transaction that conflicts with any other transaction that modifies the tree. Appropriate backoff in this situation may improve the performance of a concurrent version of the CO B-tree.

3. The serial version of the CO B-tree written without LIBXAC is cache-oblivious by construction. Since LIBXAC supports multiversion concurrency by memory-mapping multiple copies of pages in complex ways, it is unclear whether the

runtime ends up creating 1 or 2 segments for every page a transaction touches.

property of cache-obliviousness still holds. For example, in Libxac, it is possible for two adjacent pages in the user file to end up being mapped to two non-adjacent pages in Libxac's log file, and vice-versa. The behavior is even more complicated when operations are being done on multiple processors. One interesting research question to explore is whether a serial, cache-oblivious B-tree can be converted into a parallel cache-oblivious structure while still supporting multiversion concurrency.

In conclusion, I consider Libxac not as a finished product, but as work in progress. The Libxac prototype has some interesting features in its implementation, but there is much room for improvement. The fact that Libxac manages to support durable search-tree insertions as efficiently as the Berkeley DB B-tree in our experiments is a strong indication that memory-mapped transactions can be practical.

I have spent the majority of this chapter discussing performance, but arguably the most important result is the one I have spent the least time discussing. Libxac is intended to be a library that is easy to program with. For concurrent and persistent programs, the hope is that the ease of parallelizing serial data structures is the rule rather than the exception.

Chapter 6

Conclusion

In this thesis, I have argued that memory-mapped transactions provide a simple yet expressive interface for writing programs with multiple processes that concurrently access disk. I described LIBXAC, a prototype C library that demonstrates that efficient and portable support for memory-mapped transactions is feasible in practice. Using LIBXAC, I was able to easily converted existing serial implementations of a B^+-tree and a CO B-tree to support concurrent searches and insertions. In an experiment with a single process performing doing random insertions on these search trees, with each insertion as durable transaction, these search trees run anywhere from 4% slower to 67% faster than transactional inserts into a Berkeley DB B-tree. This result demonstrates that it is possible to use the simple interface based on memory-mapped transactions in a practical application and still achieve good performance in practice.

I have not fully explored every aspect of memory-mapped transactions however. In this chapter, I discuss possible improvements to the LIBXAC prototype and ideas for future research. In particular, I focus on the idea of combining a hardware transactional memory system and a memory-mapped transaction system to support efficient, unbounded nondurable transactions.

6.1 Ideas for Future Work

In this section, I discuss ways the Libxac prototype could more efficiently support memory-mapped transactions and list several aspects of memory-mapped transactions that I have not explored.

At the end of Chapters 3 and 5, I enumerated several ways of improving the Libxac implementation and other interesting research questions.

1. The centralized control data structures for Libxac represent a serial bottleneck that limits the scalability of the prototype. One topic for research is how to design and implement an efficient decentralized control mechanism.

2. Libxac's control data structures have relatively naive implementations that impose unnecessary restrictions on the interface. In Appendix A, I discuss possible improvements in more detail.

3. The log file for Libxac needs to be restructured to support durable transactions for programs with multiple processes, and the recovery program needs to be implemented. Appendix B describes these issues in greater detail.

4. Although I describe a memory model for memory-mapped transactions in Section 2.2, I do not formally show that this model has reasonable semantics. For example, I claim without proof that the interaction between transactional and nontransactional operations is well-defined when nontransactional operations modify only local variables. One idea for future work is to adapt the framework described in [15, 34] to handle memory-mapped transactions.

5. One topic to explore is what policies for handling transaction conflicts and what algorithms for backoff are efficient in theory and in practice. In Section 3.3, I describe several policies for handling conflicts between transactions, but I do not experiment with these different policies.

6. Because Libxac maintains multiple versions of a given page, it is not immediately clear whether the cache-oblivious property of a CO B-tree is maintained

when insertions are performed as memory-mapped transactions, or when insertions happen in parallel. It would be interesting to investigate whether cache-oblivious algorithms can retain their optimal caching behavior in a memory-mapped transaction system.

6.2 Libxac and Transactional Memory

By combining Libxac with other transactional memory systems, I believe it is possible to support efficient, unbounded nondurable transactions. I finish this thesis by sketching one possibility for combining Libxac with a hardware implementation of transactional memory.

Although the primary goal in designing Libxac was to create a convenient interface for programming concurrent, disk-based data structures, the inspiration for Libxac was actually to create a library for software transactional memory in C. Unfortunately, the results in Chapters 4 suggest that the per-page overheads incurred by the Libxac runtime may be too great for practical, nondurable transactions. A system such as Libxac may not be able to support efficient nondurable transactions by itself. By combining Libxac with other transactional memory systems, I believe it is possible to support efficient, unbounded nondurable transactions.

The term transactional memory was originally used by Herlihy and Moss to describe a hardware scheme for supporting atomic transactions, HTM [24]. The HTM scheme uses extra bits in the cache to mark when a cache line is accessed by a transaction. By modifying the existing cache-coherence protocols, HTM guarantees that transactions execute atomically. HTM is unable to handle transactions that did not fit completely into the transactional cache. Ananian, et. al. in [1] describe an improved hardware scheme, UTM, that support transactions of unbounded size and duration. UTM added two new machine instructions: `xbegin` and `xend`. Any instructions on that same thread between `xbegin` and `xend` form a transaction.

The fact that Libxac has function calls named `xbegin` and `xend` is not a coincidence. Libxac was originally intended to be an implementation of software transac-

tional memory at a page-level granularity. In terms of programming interface, Libxac is similar to transactional memory; the library is as easy to use for concurrent programming as the original hardware proposal for transactional memory was intended to be. In terms of performance, however, Libxac is not successful. Transactional memory is specifically designed with small, nondurable transactions in mind. Even with substantial improvements to the runtime, it is possible that a page-granular transaction system using memory mapping may not be as practical as other software transactional memory implementations for small transactions.

Nondurable transactions in Libxac may still be practical in other situations, however. As the authors of UTM argue, not all transactions are small. In an experiment where the Linux 2.4 kernel was "transactified", the authors notice that 99.9% of all transactions touched less than 54 cache lines, but the largest transaction touched over 7,000 64-byte cache lines. They advocate that hardware transactional memory implementations should support transactions of unbounded size. From the perspective of good software engineering, this principle makes sense. If the working set of a transaction is limited to the size of the hardware cache, for example, then transactional code that runs on one machine may not be supported on a machine with a smaller cache. Code complexity would increase as applications would now be system-dependent. Note that this problem is not only a performance issue. It is not correct to simply use hardware transactional memory for small transactions and use ordinary locks for large transactions; by default, these two mechanisms are incompatible with each other.

UTM supports unbounded transactions by spilling cache lines from transactions into main memory and handling that transaction in software. The mechanisms for doing this are fairly complicated however. With access to a system such as Libxac, however, it may be possible to support unbounded transactions with only the simpler hardware scheme of [24]. In what follows, I sketch one possible proposal for integrating the various transaction systems.

A Generic Transactional Memory Hierarchy

My proposal for integrating Libxac with other transactional memory systems is motivated by the idea of the memory hierarchy. Computer systems cache memory at different granularities. For example, machine instructions operate on memory stored in registers, threads often access memory stored in lines in cache, and a process usually accesses pages that are stored in RAM. A particular program can be thought of as having a particular level of the hierarchy at which it primarily operates; usually, most of the working set of a program can be completely cached at this level. Typically, moving data in and out of the cache at this level represents a performance bottleneck.

We can apply the same concept to parallel programs with transactions. A transaction with a large working set should operate at a higher level in the hierarchy than a transaction with a small working set. It does not make sense to handle concurrency control on a cache-line basis for two transactions that each touch a total of 10 pages of data. It seems equally bad to handle concurrency control at the page level for two transactions that each touch only 10 different cache lines. The concurrency control for transactional memory should operate at appropriate levels in the memory hierarchy.

Consider a two-level memory hierarchy, with block sizes B_1, B_2 and cache sizes M_1, M_2, respectively. We assume that $B_1 < B_2$, $M_1 < M_2$. $B_1 \ll M_1$ and $B_2 \ll M_2$. For any memory address x, let $f_1(x)$ and $f_2(x)$ be the corresponding size-B_1 and size-B_2 blocks that contain x. We assume that the caches are inclusive: for any memory address x, if $f_1(x)$ is cached at level 1, then $f_2(x)$ must also be cached at level 2.

In a simple transactional memory hierarchy, I propose that every transaction instance executes at only one level of the hierarchy. Concurrency control for all transactions at a particular level is handled by a hardware (or software) transactional memory (TM) scheme operating at that level. For example, the level-1 TM scheme guarantees level-1 transactions are atomic by monitoring accesses to blocks of size B_1. Similarly, the level-2 TM scheme tracks accesses to blocks of size B_2.

The tricky part is determining how the two levels can communicate with each other. I sketch a simple scheme that tries to keep the two levels as independent of

each other as possible. I propose that at every level of the hierarchy, a block can conceptually be in one of the following states: N, R, W, L, or U.

- **N**: Not in a transaction. This status must be consistent at all levels of the hierarchy. For example, if block q_1 is marked as N at level 1, then $f_2(q_1)$ (the block at level 2 containing q_1) must also be marked as N.

- **R**: Read by a transaction. At a particular level, multiple transactions may be reading the same block. If level 1 has block q_1 in state R, then $f_2(q_1)$ must be in state L at level 2. On the other hand, if level 2 has a block q_2 in state R, then all level-1 blocks in q_2 must be in state U if they are cached at level 1.

- **W**: Written by a transaction. Typically, only one transaction is allowed have a block in this state. Again, if level 1 has block q_1 in state W, the $f_2(q_1)$ must be in state L at level 2. If level 2 has a block q_2 in state W, then any level-1 blocks in q_2 cached at level 1 must be in state U.

- **L**: A lower level TM system is handling this block. If a block q_2 is in state L at level 2, then level 1 transactions are free to access the entire block q_2 without interference from the level 2 TM system. Level-2 transactions would not be allowed to access any blocks in state L.

- **U**: An upper level TM system is handling this block. For example, if block q_1 is in state U at level 1, then every small block in the larger block $f_2(q_1)$ that is in the level 1 cache must also be in state U. In this example, a level 1 transaction is not allowed to access any memory in block $f_2(q_1)$, but level 2 transactions can.

The following communication is required between the different layers (for simplicity, I only describe 2 layers, but this description can be generalized to multiple layers):

1. When a level-1 transaction tries to access an uncached block q_1 or a block in the U state, the runtime must communicate with the level-2 TM system:

- If level 2 has $f_2(q_1)$ in state L or N or uncached, then level 2 sets the state on that block to L, and level 1 can then access block q_1.
- If level 2 has $f_2(q_1)$ in the R or W state, then the level 1 TM system must signal to level 2 that it wishes to access block $f_2(q_1)$. The transaction at level 2 must either complete or abort before the level 1 transaction can continue. Alternatively, the level-1 transaction could simply abort.
- If level 2 has $f_2(q_1)$ in the U state, then level 2 must signal up to any higher levels and wait for them to either abort or complete their transactions involving the block $f_2(q_1)$.

2. When a level-2 transaction tries to access any level-2 block q_2 that is in the L state, it must communicate with the level-1 TM system and ask it to release all level-1 blocks in q_2 (i.e. flush them or set them in the U state).

In summary, a level-i TM system must communicate with level $i+1$ whenever it tries to access an uncached block or a block in the U state. Level i communicates with level $i-1$ when it tries to access a block in the L state. A particular transaction instance T always executes at a single level. Transactions in the system can be executing at both levels concurrently, but level-1 transactions are completely independent of level-2 transactions (at the granularity of level 2). Under certain conditions, transactions may be moved up to execute at a higher level (or even down to execute at a lower level), depending on the policies for dealing with aborted transactions. By extending this scheme to multiple levels of transactional memory, we can in principle support transactions of unbounded size.

A Specific Example

Imagine that level 1 corresponds to a hardware transactional memory (HTM) mechanism that operates on cache lines, as proposed in [24], and level 2 is actually a software mechanism such as LIBXAC that operates on pages. I propose one possible design for integrating these two systems. In this design, I attempt to use the HTM scheme largely as a black box, making a minimal number of modifications.

I describe a scheme that handles a restricted set of programs with multiple threads and processes. HTM works both on transactions running on different threads or on different processes, while LIBXAC only works for sharing data between different processes through a memory-mapped file. In the scheme I propose, every process must either be in HTM mode (level 1) or LIBXAC mode (level 2):

- In HTM mode, a process can have multiple threads, each possibly accessing the shared-memory segment. The restriction is, however, that no level-2 transactions can be executed on this process while in HTM mode.

- Similarly, in LIBXAC mode, the process must execute serially, and no level-1 transactions can be executed.

Each process can have a page in the shared memory segment in one of the four possible states: N, R, W and L. The system behavior depends on these states:

- For all processes in LIBXAC mode, pages that have state N or L are mapped with no-access protection. When a level-2 transaction tries to read or write to a page x for the first time, it causes a `SIGSEGV`. As long as that no process has x in the L state, LIBXAC handles conflicts normally.

  If the current process or any other process has x in the L state however, then there is a conflict between levels 1 and 2 on page x. One solution is to have the level-2 transaction always abort. The other extreme is to have LIBXAC tell HTM system to evict all cache lines on that page from all transactional caches, wait until this process finishes, and then mark the page as R or W as before. At this point, it is unclear whether one policy is better than the other.

- For all processes in HTM mode, pages that are in state N are mapped with no-access protection, but pages in state L are mapped with read/write protection. This choice means that level-1 transactions are free to modify pages in state L without incurring any LIBXAC overhead. Conflicts between level-1 transactions are handled automatically by the HTM layer.

When a level-1 transaction tries to access a page marked N, it is as though this page was not in memory initially. LIBXAC's SIGSEGV handler traps this access and then checks for conflicts. If no other process has this page set to R or W, then it changes the page state for this process to L.[1] If some other process is reading or writing to this page, then we again have a conflict between a level 1 and level 2 transaction. The two choices are to have the level-1 transaction abort, or have the HTM system signal to LIBXAC to release that particular page and mark it as L.

Unresolved Questions

This abstract description of the scheme is far from a complete. Many important details have not been worked out:

1. I have not specified the exact communication protocols between TM levels 1 and 2. In the current proposal, the HTM layer communicates only indirectly with level 2 by causing a SIGSEGV when accessing a page mapped with no-access protection. LIBXAC needs to have a way to ask the HTM system to release all cache-lines from a particular page, either by flushing them from the cache or putting them in the U state. We do not have to maintain an explicit U state in level 1 if we simply flush those lines from the transactional cache. In the worst case, if this selective flushing of cache lines is difficult to accomplish, LIBXAC could simply flush the entire transactional cache. More efficient solutions might also be feasible, however.

2. Although the requirement that each process either be in HTM mode or LIBXAC mode is restrictive, is seems necessary because there appears to be no easy way set memory protections on a per-thread basis instead of per-process. Having a level-1 transaction and a level-2 transaction running concurrently on the same process is difficult for the same reason that having multiple threads in LIBXAC

[1] To support multiversion concurrency for level-2 transactions, we may also copy the page before switching it to status L.

is. Solving the latter problem might remove this restriction from the former.

3. Efficiently switching a process between Libxac mode and HTM mode may be an inefficient operation. In the current proposal, to switch modes, we must flip the memory-protection of all the pages marked L between read/write access and no-access.

   This operation may be quite expensive if many pages are marked as L. In fact, we expect the common case to be that most transactions execute at level 1, so most pages should be marked as L by level 2. On the other hand, switches between Libxac and HTM mode should occur only infrequently.

4. My previous description of a transactional memory hierarchy implicitly assumes that all the TM levels do incremental validation of transactions, i.e., that two transactions will not simultaneously and optimistically modify a memory block and discover a conflict only at commit. One can imagine trying to design a hierarchy where some levels can execute transactions optimistically.

5. Similarly, Libxac actually supports multiversion concurrency. In our specific example, because Libxac represents the top of the hierarchy, this fact does not seem to be a problem. One can imagine however, having another layer on top of Libxac that handles transactions on a distributed-shared memory cluster. It may be possible to have a hierarchy with some levels supporting multiversion concurrency.

6. Finally, we might integrate Libxac and HTM in a different way by trying to use hardware transactional memory to implement the Libxac runtime system. With the current proposal, this approach may be problematic because Libxac is constantly doing system calls that involve context switches that may flush the transactional cache. Still, if this issue could be resolved, then Libxac might do concurrency control between transactions by creating a meta-transaction that is handled in HTM. In this approach, every page in the shared-memory segment gets mapped to a particular cache line. Every time a level-2 transaction reads

or writes from a new page, Libxac will read or write from the appropriate cache line. Thus, Libxac may be able to use the HTM mechanism to detect transaction conflicts at the page level.

In summary, I have attempted to sketch one possible description for a transactional memory hierarchy, based on two specific transactional memory implementations. This design is still in the earliest stages of completion. I believe, however, that it is a good first step towards having a unified programming interface for concurrent programming that simplifies code and works efficiently at all levels of the memory hierarchy.

Appendix A

Restrictions to the Libxac Interface

A.1 Restrictions to Libxac

This appendix enumerates the various programming restrictions when using LIBXAC, and discusses potential improvements to the implementation.

1. LIBXAC's most significant restriction is that only one transaction per process is allowed. LIBXAC uses the no-access, read-only, and read/write memory protections provided by the `mmap` function. If the OS maintains a single memory map for each process rather than for each thread, we can not use memory protections to map a page read-only for one thread and read/write for a different thread on the same process. In this case, LIBXAC could not support multiple concurrent transactions on the same process without using a different mechanism for detecting the pages accessed by a transaction.

2. LIBXAC currently supports having only one shared memory segment. In other words, all concurrent processes that `xMmap` the same file must call `xMmap` with the same filename and length arguments. The motivation for this restriction is that programs can often simply make one call to `xMmap` to map one large file for the entire shared memory space.

   The implementation described in Chapter 3 could theoretically be extended to support the full functionality of normal `mmap` (i.e allowing multiple `xMmap` calls

on different files, mapping only part of a file instead of the entire file, or mapping the same page in a file to multiple addresses). These extensions would require LIBXAC to maintain a more complicated map between the transactional page addresses returned by `xMmap` and the physical page address of files on disk, but these changes are, in principle, relatively straightforward. In this extension, accesses to pages in multiple files would be treated as though the multiple files were concatenated logically into one large shared file.

   This idea is different from the proposal of allowing a process access to multiple shared memory segments that are logically distinct in terms of the LIBXAC's atomicity guarantees. Having two distinct shared-memory segments (for example, A and B) has interesting semantics. LIBXAC would guarantee that transactions are serializable only with respect to operations on A, and also with respect to only operations on B. The serial order of transactions could be different for A and for B, however, so there is no guarantee of serializability if we consider all operations on A and B together.

3. The prototype arbitrarily sets the maximum size of the shared memory segment to 100,000 pages, and the maximum number of concurrent transactions to 16. This restriction allowed us to implement control data structures simply (and inefficiently) with large fixed-size arrays. Using dynamic control structures easily removes this limitation, and may also improve the caching behavior of the runtime system.

4. Linux limits a process to having at most 2^{16} different memory segments (virtual memory areas, or vma's in the kernel [10]) at any point in time. Whenever a transaction touches a page, LIBXAC calls `mmap` on that page and generates a new segment for that page. Thus, a single transaction can not possibly touch more than 2^{16} pages at once.

   One way of raising this limit, aside from modifying the Linux kernel, is to concatenate adjacent segments together when a transaction touches adjacent pages. This proposal does not fix the problem, as it is possible for a transaction

to touch every other page. In that situation however, Libxac should escalate its concurrency control and work at a larger granularity, treating multiple pages as a single large segment when it detects a large transaction.

5. Every `xbegin` function call must be properly paired with an `xend`. The control flow of a program should not jump out of a transaction. If we also require that every `xbegin` must be paired with exactly one `xend`, then it is possible to detect unmatched function calls with compiler support.

6. As described in Section 3.2, recovery for durable transactions has not been implemented. Also, the structure of the log file is incorrect for transactions on multiple processes if transactions write data pages to the log that can be confused as metadata pages. Separating the metadata and actual data pages into separate files solves this problem and also facilitates further optimizations such as logging only page diffs.

7. Libxac does not provide any mechanism for allocating memory from the shared memory segment. This shortcoming is not technically a restriction on the implementation, but it is quite inconvenient if the programmer wishes to dynamically allocate and work with transactional objects. Providing a `malloc` function that allocates memory from the shared segment might lead to simpler user programs.

Appendix B

Transaction Recovery in Libxac

For durable transactions the LIBXAC prototype writes meta-data pages in the log and synchronously force changes out to disk. This in principle is enough to recover from a program crash and restore the user data file back to a consistent state, assuming the disk itself has not failed.

Although LIBXAC writes enough data to disk to do recovery, I have not yet implemented the recovery module for the prototype. In this section, I sketch a possible algorithm for transaction recovery when transactions execute on a single process.

LIBXAC's log files have the following structure:

- The `XCOMMIT` pages appear in the log in the order that transactions are committed.
- The `XBEGIN` page points to all pages belonging to T, and also to any spill-over pages for this storing this list. For any transaction, T, all spill-over pages and data pages written by T appear between the T's `XBEGIN` and `XCOMMIT` (or `XABORT`) meta-data pages.
- All pages in a new log file are initialized to all zeros before the log file is used.

Figure B-1 illustrates two example layouts for the log file when transactions are executing on a single process and two processes, respectively. At a high level, the recovery algorithm is as follows:

	Log entry		
0	XBEGIN	1	[1, 2 3*]
1		1	
2		1	
3	XCOMMIT	1	
4	XBEGIN	2	[5, 6*]
5		2	
6	XABORT	2	
7	XBEGIN	3	[8, 10, 11*]
8		3	
9	CHECKPT BEGIN		[7**]
10		3	
11	XCOMMIT	3	
12	XBEGIN	4	[14, 15*]
13	CHECKPT END		
14		4	
15	XCOMMIT	4	
16	???		
17	???		
18	???		
19	XCOMMIT	5	

(a)

	Log entry		
0	XBEGIN	1	[1, 4, 6*]
1		1	
2	XBEGIN	2	[3, 5*]
3		2	
4		1	
5	XCOMMIT	2	
6	XCOMMIT	1	
7	???		
8	???		
9	XBEGIN	4	[11, 12, 13, 14**]
10	???		
11		4	
12		4	
13		4	
14	XCOMMIT	4	
15	XCOMMIT	3	

(b)

Figure B-1: An example of a LIBXAC log file when transactions execute (a) on one process, and (b) on two processes.

1. Go to log file containing the last `XCHECKPT_BEGIN` page that has a valid matching `XCHECKPT_END`.[1] Scan through the entire log starting at this point and compute which transaction each page in the log belongs to.

   We know that every valid page that comes after the `XCHECKPT_BEGIN` page is either (i) a meta-data page, (ii) a data page pointed to by an `XBEGIN` page in the list stored in the `XCHECKPT_BEGIN` page, or (iii) a data page pointed to by some `XBEGIN` page that comes after the `XCHECKPT_BEGIN` page. Therefore, we can match pages for all transactions after the last checkpoint. Any unmatched pages are considered to be invalid.

   For a transaction T, we may detect the following inconsistencies:

   - T has an `XBEGIN` page, but no `XCOMMIT` or `XABORT` page. This event means T had not completed at the time of the crash, or the `XCOMMIT` page did not make it out to disk.
   - T has both `XBEGIN` and `XCOMMIT` pages, but the checksum is wrong. Since writes of multiple pages are not guaranteed to happen atomically, the `XCOMMIT` page may get written to disk before one of the data pages. The checksum should detect this error.
   - T does not have an `XBEGIN` page. This situation can occur if a system crashes before the `XBEGIN` is flushed to disk. In this case, none of T's data pages will be pointed to by a valid transaction.

   In these three situations, the transaction T is considered to be aborted.

2. Once we have identified which transactions in the log were successfully committed, we can replay all those transactions in the correct serial order. This process is done by copying a transaction's pages into the original file. Alternatively, we

[1] We assume LIBXAC maintains a separate file on disk recording the location of all checkpoint meta-data pages. This file is synchronized a second `fsync` that occurs after the first `fsync` does the actual synchronization. Thus, the `XCHECKPT_END` page is not considered valid until this meta-meta-data appears on disk.

could also attempting a more clever algorithm that works from the end of the log and only copies the latest version of each page back into the file.

Note that in the actual implementation, the recovery process must itself keep a log of its changes so we can restore the data to a consistent state in case of a crash during the recovery process.

In the example in Figure B-1 (a), scanning through the log file, we discover that the last valid checkpoint started at page 9. We only need to repeat transaction 3 because it is pointed to by the `XCHECKPT_BEGIN` page, and transaction 4 because it comes after that page. Transaction 5 attempted to commit, and the `XCOMMIT` page made it to disk. Its checksum will be incorrect however, as the corresponding `XBEGIN` page did not make it successfully to disk. Note that the system must have crashed before `fsync` returned. Thus, LIBXAC did not see transaction 5 finish its commit, and no other transaction could have read values written by 5.

The example in Figure B-1 (b) shows transactions executing concurrently on two processes. In this case, pages from different transactions can be interleaved in the log file. For this example, transaction 3 crashed while attempting to commit, and thus its `XBEGIN` page at page 7 did not successfully get written to disk. Transaction 4 that was executing on the other process did successfully commit however. Transaction 4 could not have read pages from transaction 3 because the `fsync` that was writing transaction 3's data did not succeed.

When all transactions are executed on a single process (but checkpointing may be done by a different process), this recovery process works correctly because LIBXAC guarantees that the log will have at most two invalid data pages interleaved between valid log pages (the two pages reserved for `XCHECKPT_BEGIN` and `XCHECKPT_END`, in case the checkpointing process crashed). Since we record the pointers to the checkpoint meta-data pages in a separate file, we can always distinguish meta-data pages from data pages.

Unfortunately, this recovery algorithm does not always work when two or more processes execute transactions. In Figure B-1(b), when we see pages 9 and 14 in the log, these could either be meta-data pages for transaction 4, or they could theoreti-

cally be data pages for transaction 3 that crashed. This proposed recovery algorithm assumes that it is always possible to distinguish between data pages and meta-data pages. It seems quite unlikely that a programmer would accidentally execute a transaction that writes data pages that are exactly the meta-data plus data pages for a valid transaction. It is conceivable, however, that a programmer might call `xMmap` on a log file generated by Libxac, and perhaps copy a portion of this file as part of a transaction. In this case, the recovery process can no longer distinguish between data and meta-data.

When pages from transactions on different processes can be interleaved, it seems difficult to differentiate between data and meta-data without imposing additional structure on the log file. Possible solutions to this problem are to enforce some global structure in log (ex. all odd pages are meta-data, all even pages are data), to use separate log files for each process, or to use separate files for data and meta-data. The last two options are perhaps the most practical, although these implementations may require multiple disks to achieve good performance. Otherwise, the system may waste a significant amount of time doing disk seeks between two different files on disk.

Appendix C

Detailed Experimental Results

This appendix presents a more detailed description of some of the experiments described in Chapters 4 and 5. It also contains more detailed data collected from these experiments.

C.1 Timer Resolution

In this section, I describe an experiment to determine the resolution of the timers used in the empirical studies. Using the processor's cycle counter and `gettimeofday` is accurate at least for measuring times greater than 50 ns and 10 μs, respectively.

In all experiments, I measured the time for an event by checking the system time before and after the event and reporting the difference. For nondurable transactions, I typically used the processor's cycle counter, via the `rdtsc` instruction. For longer events, I used two calls to the `gettimeofday` function. To understand the resolution of this method, I measured the difference between two consecutive calls to check the timer, with no code in between. The results from repeating this experiment 10,000 times for `rdtsc` and `gettimeofday` are shown in Tables C.1 and C.2, respectively.

In Table C.1, the data on Machines 1 and 2 suggests that the delay when checking the cycle counter is about 100 clock cycles (less than 50 ns). The delay is even less for Machines 3 and 4. Although the maximum value on Machine 1 was about 12 μs, the 99th percentile was still under 50 ns, suggesting that this mechanism for

Machine	Mean	St. Dev	Min	Median	99th Percentile	Max
1	99.5	458	92	92	112	35332
2	92.0	0.92	92	92	92	184
3	9.0	3.2	5	8	23	36
4	41.0	0.16	41	41	41	51

Table C.1: Delay (in clock cycles) between successive calls to timer using `rdtsc` instruction, 10,000 repetitions.

Machine	Mean	St. Dev	Min	Median	99th Percentile	Max
1	2.87	2.24	2.0	3.0	7.0	201.0
2	0.87	0.39	0.0	1.0	1.0	20.0
3	1.3	0.50	1.0	1.0	2.0	8.0
4	1.55	2.99	1.0	2.0	2.0	271.0

Table C.2: Delay between successive calls to `gettimeofday` (in μs), 10,000 repetitions.

estimating times is reasonably accurate for measuring times to within a few tenths of a microsecond, provided we make repeated measurements.

From the data in Table C.2, we observe that the delay between `gettimeofday` calls on all four machines was less than 7 microseconds for 99% of measurements. This data suggests `gettimeofday` has a resolution of a few microseconds.

On Machine 1, the maximum delay was approximately 200 μs. If we observe the distribution of delay times, as shown in Figure C-1, then we see that this was a rare event. This behavior is not surprising, as it is impossible to stop basic system processes during our experiments. An interrupt or other operating system process may have caused the timer code to get swapped out. These rare but expensive delays must be kept in mind when interpreting the experimental results.

In summary, the data suggests we can use `gettimeofday` to measure times longer than approximately 10 μs with reasonable accuracy provided the time interval is long enough or we do enough repetitions.

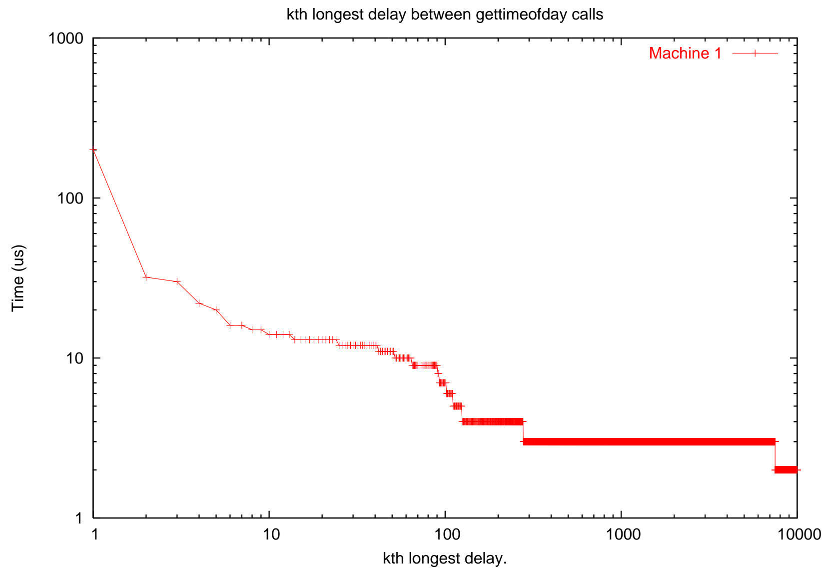


Figure C-1: Machine 1:Distribution of Delay Times Between Successive `gettimeofday` Calls.

C.2 Page-Touch Experiments

This section contains results from the page-touch experiments on Machines 2 through 4. These experiments were previously described in Section 4.2.1 and 4.2.2. Figures C-2 through C-4 plot the times per page read and write in the page-touch experiments with nondurable transactions.

C.3 Experiments on Various System Calls

In this section, I present detailed data from the microbenchmark experiments discussed in Section 4.2.3.

Memory Mapping and Fault Handlers

Table C.3 is a more complete version of Table 4.3.

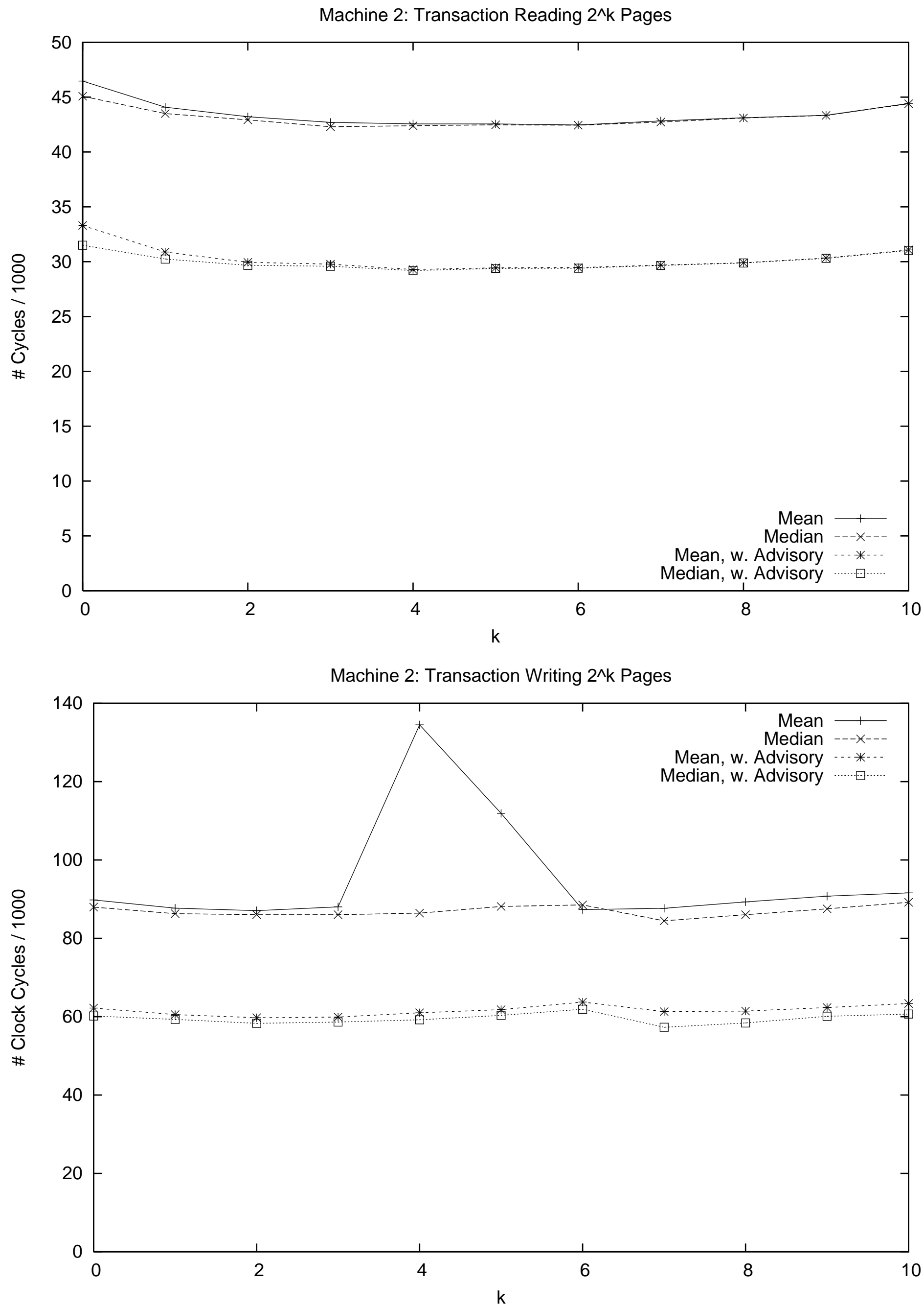


Figure C-2: Average time per page to execute the transactions shown in Figure 4-3 on Machine 2. For each value of n, each transaction was repeated 1000 times.

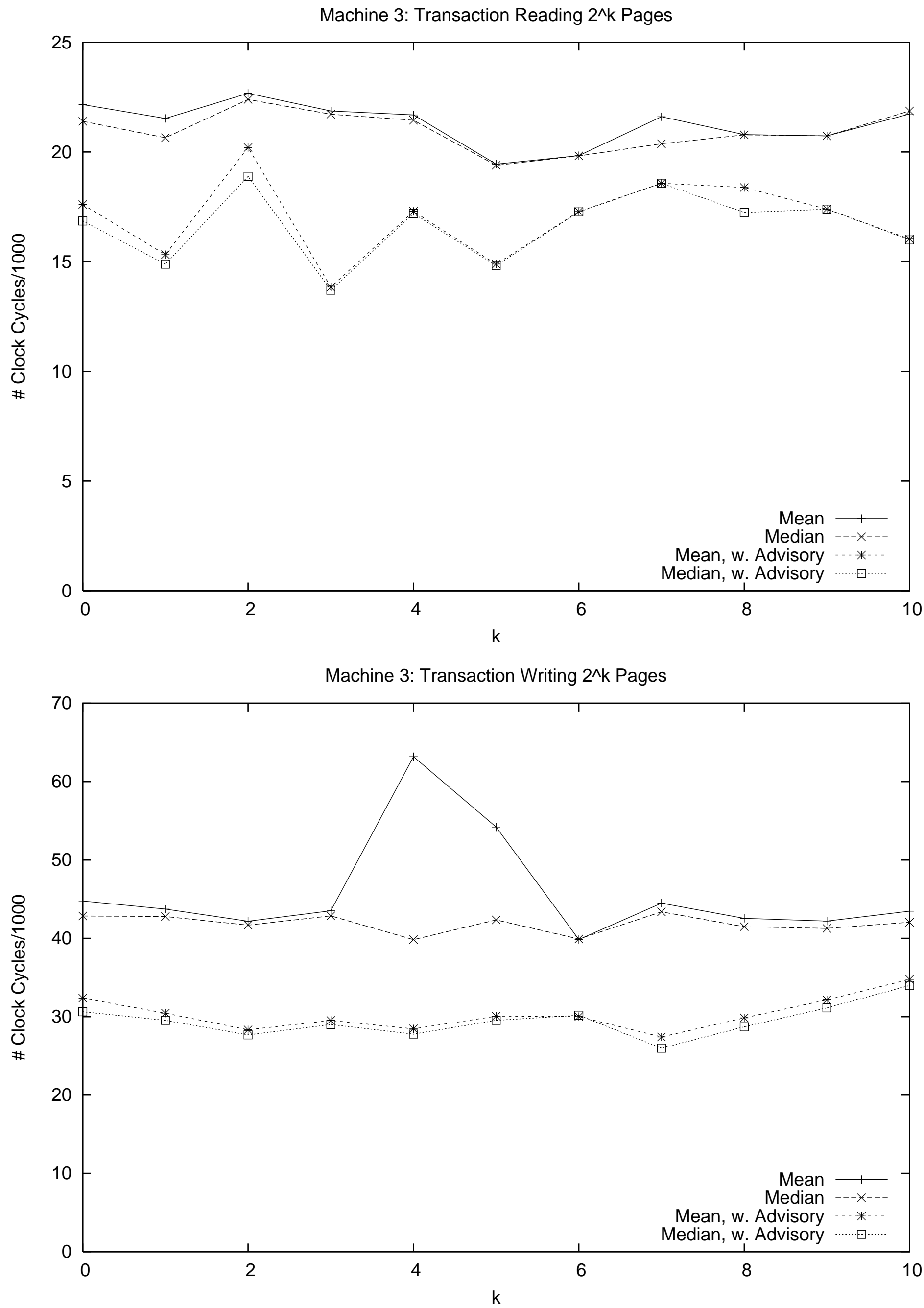


Figure C-3: Average time per page to execute the transactions shown in Figure 4-3 on Machine 3. For each value of n, each transaction was repeated 1000 times.

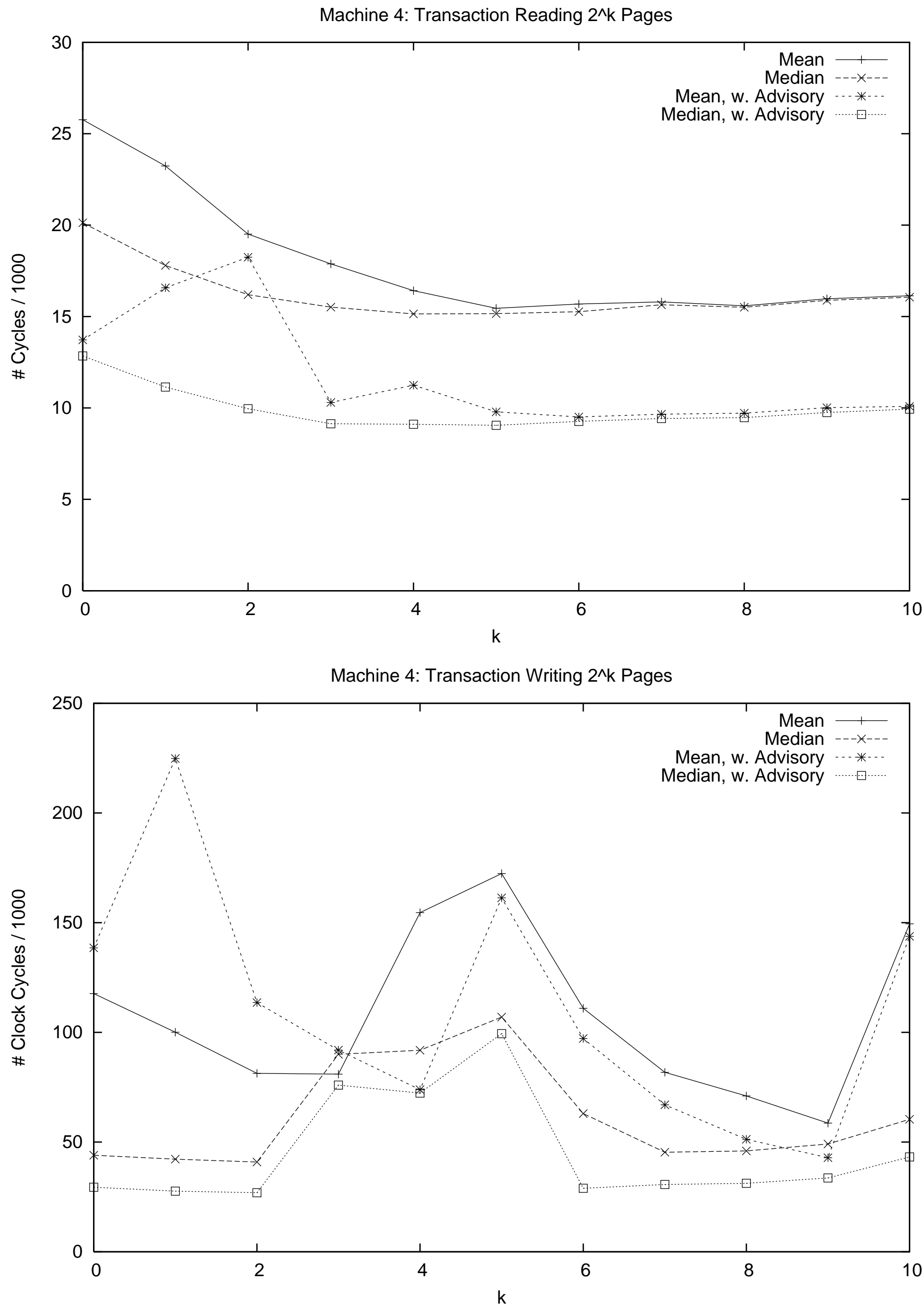


Figure C-4: Average time per page to execute the transactions shown in Figure 4-3 on Machine 4. For each value of n, each transaction was repeated 1000 times.

Operation	Mean	St. Dev	Min	Median	99th %tile	Max
1: Entering `SIGSEGV`	32,216	11,268	33,320	33,956	54,772	648,236
1: `mmap`	15,156	2,747	13,916	14,580	26,672	70,748
1: Exiting `SIGSEGV`	29,323	9,288	27,408	28,428	44,996	635,340
2: Entering `SIGSEGV`	8,032	546	7,884	8,032	8,132	45,720
2: `mmap`	10,054	639	9,920	10,024	10,380	49,520
2: Exiting `SIGSEGV`	9,723	830	9,632	9,704	9,872	46,960
3: Entering `SIGSEGV`	3,489	182	3,238	3,466	3,741	13,190
3: `mmap`	3,228	512	2,892	3,205	3,420	13,335
3: Exiting `SIGSEGV`	4,745	110	4,377	4,709	4,844	9,204
4: Entering `SIGSEGV`	3,078	613	2,971	3,051	3,155	30,968
4: `mmap`	2,282	711	2,142	2,259	2,340	56,322
4: Exiting `SIGSEGV`	3,140	537	3,055	3,114	3,267	21,255

Table C.3: Timing data for entering `SIGSEGV` handler, calling `mmap`, and leaving handler, 10,000 repetitions. All times are processor cycles.

Operation	Mean	St. Dev	Min	Median	99th %tile	Max
1: `memcpy`	1,122,167	129,167	1,064,328	1,076,620	1.803,956	2,204,000
2: `memcpy`	1,052,253	7,379	1,046,292	1,052,316	1,086,384	1,106,784
3: `memcpy`	608,482	8,049	605,017	608,112	612,237	791,724
4: `memcpy`	931,823	98,636	925,048	925,089	953,525	4,037,042

Table C.4: Clock cycles to do 1,000 calls to `memcpy` between two 4K character arrays in memory, 1,000 repetitions. times are in μs.

Test of `memcpy`

Table C.4 is a more detailed look at the time required to do `memcpy` between two arrays 1,000 times (i.e. the last column of Table 4.5).

C.4 Durable Transactions

All times in this section of the appendix were measured using `gettimeofday` as the clock.

Page-Touch Experiments

Table C.5 is a more complete version of Table 4.7 from Section 4.3.

Machine	Mean	σ	Min.	Median	99th %tile	Max
1, Page Read	38.6	65.0	34.6	35.5	52.7	2044.5
1, Page Read w. Adv.	22.4	75.0	18.1	18.8	36.0	2032.2
1, Page Write	1569.3	4146.0	560.4	731.3	9578.2	106,560
1, Page Write w. Adv.	1297.4	2808.0	524.4	842.5	8645.2	54,514
3(a), Page Read	15.4	33.1	14.1	14.4	14.7	1062.6
3(a), Page Read w. Adv.	14.0	32.5	11.8	13.0	13.5	1039.4
3(a), Page Write	235.0	276.3	108.7	126.0	1231.71	1800.13
3(a), Page Write w. Adv.	179.4	173.9	102.9	114.7	822.3	1612.6
3(b), Page Read	16.4	10.6	15.5	15.8	30.2	344.6
3(b), Page Read w. Adv.	15.1	25.2	12.7	13.8	25.9	636.2
3(b), Page Write	226.8	297.6	103.9	121.2	1510.3	1798.0
3(b), Page Write w. Adv.	182.2	203.8	97.8	115.3	1072.4	1674.3

Table C.5: Average Access Time (μs) per Page, for Transactions Touching 1024 Pages.

Synchronizing a File

Table C.6 is a more complete version of Table 4.8.

Write Speed

This benchmark measures the time to write 10,000 pages to a file, 1 page at a time, using the `write` system call. This operation was repeated 1000 times. The results from this test are shown below in Table C.7.

The data suggests that `lseek` is actually lazy, with the disk head not actually moving until an I/O operation executes. This test was done when the write-cache on all machines was enabled.

Checksum Calculations

Table C.8 shows the time required to calculate the SHA1 and MD5 hash functions on a single page. On Machines 1 and 2, the average time for MD5 is 12 and 15 μs, respectively. This data suggests that the overhead of computing a checksum of each page a transaction writes on a transaction commit is not too expensive for small durable transactions.

Operation	Mean	St. Dev	Min	Median	99th Percentile	Max
1: `msync`	8	1	7	8	16	27
1: `fsync`	13575	184338	2439	8015	12672	5836456
2: `msync`	11	3	8	9	19	25
2: `fsync`	5137	36919	294	2663	41061	1156245
3(a): `msync`	5	1	4	5	6	12
3(a): `fsync`	4794	24018	735	4002	7000	761552
3(b): `msync`	5	1	4	5	6	12
3(b): `fsync`	4698	23090	818	3949	6925	731910
4: `msync`	33	6	28	31	43	111
4: `fsync`	3531	46,721	592	632	32,576	1,460,662

Table C.6: Timing data for calling `msync` and `fsync` on a 10,000 page file with a random page modified, 1000 repetitions. All times are in μs.

Operation	Mean	St. Dev	Min	Median	99th %tile	Max
1: `lseek`	6	2	5	6	13	62
1: `write`	103,446	244,907	90,564	92,088	113,437	5,615,347
2: `lseek`	4	13	2	3	4	406
2: `write`	151,895	166,318	119,006	119,571	1,193,881	1,278,444
3(a): `lseek`	3	1	2	2	4	9
3(a): `write`	63,465	4,036	62,992	63,270	64,268	190,798
3(b): `lseek`	3	1	2	3	4	9
3(b): `write`	92,507	70,402	78,171	83,658	586,612	837,660
4: `lseek`	9	5	7	8	9	103
4: `write`	2,168,185	504,761	1,423,656	2,521,593	2,893,774	3,070,492

Table C.7: Time to write 10,000 pages to a file, 1,000 repetitions. All times are in μs.

Machine	Hash	Mean	σ	Min	Median	99th %	Max
1	SHA1	85.6	16.5	81.7	82.9	118.6	685.6
1	MD5	36.3	10.1	34.4	35.1	67.3	680.5
2	SHA1	84.5	2.3	83.6	84.2	86.4	177.5
2	MD5	35.5	5.7	34.4	35.5	35.9	582.0
4	SHA1	105.2	555.9	52.2	52.3	69.2	6,069.2
4	MD5	54.6	412.2	25.9	25.9	30.6	6,050.1

Table C.8: Time to compute SHA1 and MD5 hash functions on a single page. All times are in thousands of clock cycles.

C.5 Concurrency Tests

Table C.9 shows the average time required per nondurable transaction for the concurrency tests described in Section 4.4. This table represents a more complete version of Table 4.9. For transactions on two processes, I report the number of aborted transactions on each process. All times in this section are measured with `gettimeofday`.

Similarly, Table C.10 is a more complete version of the data for concurrency tests for durable transactions, originally presented in Table 4.10.

C.6 Search Trees using Libxac

I use `gettimeofday` as the timers for all experiments on the LIBXAC search trees. For insertions done on a single process, I measure and record the time required for every insertion. To provide a comparison on two processes, I use a call to `fork`, and did 125,000 insertions on each process. To estimate the time required for each insertion, I record the time to complete each insertion and compare that to the time before the call to `fork`. Note that this introduce a slight bias in favor of the single process because I am also including the time required to do the `fork` operation. On the other hand, each insertion on two processes requires only one call to `gettimeofday` instead of two.

Insertions as Nondurable Transactions

This section describes the details of the experiments done on the LIBXAC search trees and on Berkeley DB's B-tree and presents a complete table of results for nondurable and durable transactions. See Section 5.4 for details.

With the LIBXAC versions of the B^+-tree and the CO B-tree, I measured the time to insert 250,000 elements. Each search tree had 512-byte data blocks, each indexed by a 64-bit key. For the B^+-tree, the blocksize was 4K. The keys for the inserted elements were chosen at random using the `rand` function, with each insert being a separate transaction. I tested 2 versions of the B^+-tree: one unoptimized implementation, and

Machine	Test #	Avg. Time per Xaction (μs)	Standard Dev. (μs)	Speedup
1	A, 1 proc.	32.2	0.27	
1	A, 2 proc.	30.7	0.36	1.05
1	B, 1 proc.	33.6	0.25	
1	B, 2 proc.	32.2	0.87	1.04
1	C, 1 proc.	1,453	4.90	
1	C, 2 proc.	1,460	56.0	1.00
2	A, 1 proc.	26.2	0.10	
2	A, 2 proc.	23.0	0.15	1.14
2	B, 1 proc.	28.3	0.26	
2	B, 2 proc.	24.2	0.48	1.17
2	C, 1 proc.	1,787	30.3	
2	C, 2 proc.	903	36.3	1.98
3(a)	A, 1 proc.	22.9	0.63	
3(a)	A, 2 proc.	24.3	1.14	0.94
3(a)	B, 1 proc.	28.1	0.42	
3(a)	B, 2 proc.	27.5	1.05	1.02
3(a)	C, 1 proc.	2,259	3.90	
3(a)	C, 2 proc.	1,132	1.82	2.00
3(b)	A, 1 proc.	24.3	1.36	
3(b)	A, 2 proc.	24.9	1.07	0.98
3(b)	B, 1 proc.	28.2	1.67	
3(b)	B, 2 proc.	26.3	0.74	1.07
3(b)	C, 1 proc.	2,248	2.74	
3(b)	C, 2 proc.	1,130	1.30	1.99
4	A, 1 proc.	109	3.39	
4	A, 2 proc.	185	13.1	0.59
4	B, 1 proc.	121	3.39	
4	B, 2 proc.	190	11.5	0.64
4	C, 1 proc.	10,487	8.51	
4	C, 2 proc.	10,565	8.86	0.99

Table C.9: Concurrency tests for nondurable transactions. Times are μs per transaction.

Machine	Test #	Mean Time per Xaction (μs)	Standard Dev. (μs)	Speedup
1	A, 1 proc.	8,231	234	
1	A, 2 proc.	8,531	3500	0.96
1	B, 1 proc.	8,868	46.5	
1	B, 2 proc.	8,878	3133	1.00
1	C, 1 proc.	9,125	65.6	
1	C, 2 proc.	10,042	1452	0.91
3(a)	A, 1 proc.	6,116	8.23	
3(a)	A, 2 proc.	6,162	95.3	0.99
3(a)	B, 1 proc.	6,113	3.28	
3(a)	B, 2 proc.	6,201	93.8	0.98
3(a)	C, 1 proc.	6,315	6.57	
3(a)	C, 2 proc.	6,626	464	0.95
3(b)	A, 1 proc.	6,210	23.1	
3(b)	A, 2 proc.	6,264	64.8	0.99
3(b)	B, 1 proc.	6,215	18.1	
3(b)	B, 2 proc.	6,213	20.1	1.00
3(b)	C, 1 proc.	6,388	16.2	
3(b)	C, 2 proc.	6,619	438.4	0.97

Table C.10: Concurrency tests for durable transactions. Times are per transaction.

one that uses the advisory function. I only tested an unoptimized CO B-tree.

For the Berkeley DB B-Tree, I used the `DB_AUTO_COMMIT` feature to automatically make each `put` operation on the B-tree its own transaction. On Machines 1 and 3, I ran Berkeley DB version 4.2 (`-ldb-4.2`). Machines 2 and 4 had Berkeley DB version 4.1 (`-ldb-4.1`). On each machine, the cache size was set to be 2 caches, 1 MB in size.

To make transactions nondurable, for the machines Berkeley DB 4.2, I used the `DB_TXN_NOT_DURABLE` flag to turn off durability. For Machines 2 and 4, finding an appropriate point of comparison with Berkeley DB was challenging, as using Berkeley DB version 4.1 seemed to cause some incompatibility with this flag. Instead, I used the flag `DB_TXN_NOSYNC` on Machines 2 and 4.

Machine	Search Tree	# Proc.	Avg. Time (μs) per Insert	# Aborts
1	B^+-tree, no adv.	1	411	–
1	B^+-tree, no adv.	2	488	59,992, 56,193,
1	B^+-tree, w. adv.	1	240	–
1	B^+-tree, w. adv.	2	236	27,753, 28,041
1	CO B-tree, no adv.	1	490	–
1	CO B-tree, no adv.	2	455	3,370, 2,876
1	Berkeley DB	1	37	–
1	Berkeley DB	2	29	–
2	B^+-tree, no adv.	1	244	–
2	B^+-tree, no adv.	2	191	32,270, 33,397,
2	B^+-tree, w. adv.	1	189	–
2	B^+-tree, w. adv.	2	152	31,491, 27,238
2	CO B-tree, no adv.	1	260	–
2	CO B-tree, no adv.	2	189	3,733, 5,785
2	Berkeley DB	1	24	–
3(a)	B^+-tree, no adv.	1	266	–
3(a)	B^+-tree, no adv.	2	264	31,128, 27,250
3(a)	B^+-tree, w. adv.	1	232	–
3(a)	B^+-tree, w. adv.	2	229	26,877, 25,497
3(a)	CO B-tree, no adv.	1	338	–
3(a)	CO B-tree, no adv.	2	337	3,345, 5,166
3(a)	Berkeley DB	1	26	–
3(a)	Berkeley DB	2	20	–
4	B^+-tree, no adv.	1	18,019	–
4	B^+-tree, w. adv.	1	17,408	–
4	CO B-tree, no adv.	1	2,286	–
4	Berkeley DB	1	393	–

Table C.11: Time to do 250,000 nondurable insertions into LIBXAC search trees.

Machine	B^+-tree	CO B-tree	Berkeley DB B-Tree
1	2.6	2.2	2.1
2	6.0	4.8	4.6
3(a)	2.7	2.1	6.5
3(b)	2.0	1.7	14.1

Table C.12: Time to do 250,000 durable insertions on a single process into the various search trees, with write-caches on the harddrives enabled. All times are in ms.

Durable Insertions with Write-Cache Enabled

Table C.12 shows the average times per durable insert when the write-caches on the harddrives on all machines were enabled. It is unclear how to interpret these numbers, as these transactions are not strictly durable. The main point is, however, that the performance of LIBXAC search trees and Berkeley DB are still comparable under different hardware settings.

Bibliography

[1] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, February 2005.

[2] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *ASPLOS-IV*, pages 96–107, April 1991.

[3] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory. In *FOCS 1996*, pages 560–569, October 1996.

[4] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *FOCS 2000*, pages 399–409, 2000.

[5] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] Brian N. Bershad, Craig Chambers, Susan J. Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. SPIN – an extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop*, pages 68–71, 1994.

[8] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, 1990.

[9] Peter A. Buhr and Anil K. Goel. uDatabase annotated reference manual, version 1.0. Technical report, Dept. of Comp. Sci., Univ. of Waterloo, Ontario, Canada, September 1998.

[10] Rémy Card, Éric Dumas, and Franck Mével. *The Linux Kernel Book*. John Wiley and Sons, 1999.

[11] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[12] J. M. Cheng, C. R. Loosely, A. Shibamiya, and P. S. Worthington. IBM Database 2 performance: Design, implementation and tuning. *IBM Systems J.*, 23(2):189–210, 1984.

[13] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent object management system. *Software-Practice and Experience*, 14(1), January 1984.

[14] Keir Fraser. Practical lock-freedom. Technical Report 579, University of Cambridge, February 2004.

[15] Matteo Frigo. The weakest reasonable memory model. Master's thesis, MIT Department of Electrical Engineering and Computer Science, January 1998.

[16] Matteo Frigo. *Portable High-Performance Programs*. PhD thesis, MIT EECS, June 1999.

[17] R. Goldberg and R. Hassinger. The double paging anomaly. In *Proc. 1974 National Computer Conference*, pages 195–199, May 1974.

[18] Jim Gray. The transaction concept: Virtues and limitations. In *Seventh International Conference of Very Large Data Bases*, pages 144–154, September 1981.

[19] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–242, 1981.

[20] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[21] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming languages and Operating Systems*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[22] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[24] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[25] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*

'03: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pages 92–101, New York, NY, USA, 2003. ACM Press.

[26] Maurice P. Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, Providence, Rhode Island, May 2003.

[27] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[28] Shigekazu Inohara, Yoji Shigehata, Keitaro Uehara, Hajime Miyazawa, Kouhei Yamamoto, and Takashi Masuda. Page-based optimistic concurrency control for memory-mapped persistent object systems. In *HICSS (2)*, pages 645–654, 1995.

[29] Zardosht Kasheff. Cache-oblivious dynamic search trees. Master's thesis, MIT EECS, June 2004.

[30] Kevin Poulsen. Tracking the blackout bug. http://www.securityfocus.com, 2004.

[31] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level storage system. *IRE Trans. on Electronic Computers*, EC-11(2):223–235, April 1962.

[32] H.T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[33] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox PARC, April 1979.

[34] Victor Luchangco. *Memory Consistency Models for High Performance Distributed Computing*. PhD thesis, MIT, 2001.

[35] Dylan James McNamee. *Virtual Memory Alternatives for Transaction Buffer Management in a Single-level Store*. PhD thesis, Univ. of Wash., 1996.

[36] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[37] Elliot I. Organick. *The Multics System: An Examination of its Structure*. The MIT Press, Cambridge, MA, 1972.

[38] The PostgreSQL Global Development Group. *PostgreSQL 7.2.1 Documentation*, 2001.

[39] Harald Prokop. Cache-oblivious algorithms. Master's thesis, MIT EECS, June 1999.

[40] John Rosenberg, Alan Dearle, David Hulse, Anders Lindström, and Stephen Norris. Operating system support for persistent and recoverable computations. *Commun. ACM*, 39(9):62–69, 1996.

[41] Yasushi Saito and Brian Bershad. A transactional memory service in an extensible operating system. In *USENIX Annual Technical Conference*, pages 53–64, 1998.

[42] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[43] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, Fifth edition, 1998.

[44] Sleepycat Software. The Berkeley database. http://www.sleepycat.com, 2005.

[45] Frank G. Soltis. *Inside the AS/400*. Duke Press, Loveland, Colorado, 1997.

[46] Alfred Z. Spector, D. Thompson, R.F. Pausch, J.L. Eppinger, D. Duchamp, R. Draves, D.S. Daniels, and J.L. Bloch. Camelot: A distributed transaction facility for Mach and the Internet—An interim report. Technical Report CMU-CS-87-129, Carnegie Mellon University, 1987.

[47] Seth J. White and David J. DeWitt. QuickStore: A high performance mapped object store. *VLDB Journal: Very Large Data Bases*, 4(4):629–673, 1995.