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 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

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

int fileLength = 100000;
int main(void) {
  int i, j, sum = 0;
  int fd;
  int* x;

  fd = open("input.db",
            O_RDWR, 0666);
  x=(int*)malloc(4096*fileLength);
  read(fd, (void*)x,
       4096*fileLength);

  for (j = 0; j < 5; j++) {
      i = rand() % fileLength;


      sum += x[1024*i];
  }

  i = rand() % fileLength;

  x[1024*i] = sum;

  lseek(fd, 0, SEEK_SET);
  write(fd, (void*)x,
        4096*fileLength);
  close(fd);
  return 0;
}
```

A

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

int fileLength = 100000;
int main(void) {
  int i, j, sum = 0;
  int fd;
  int value;

  fd = open("input.db",
            O_RDWR, 0666);



  for (j = 0; j < 5; j++) {
     i = rand() % fileLength;
     lseek(fd, 4096*i, SEEK_SET);
     read(fd, &value, 4);
     sum += value;
  }

  i = rand() % fileLength;
  lseek(fd, 4096*i, SEEK_SET);
  write(fd, &value, 4);



  close(fd);
  return 0;
}
```

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.

```
// 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`.

```
// 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.

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.

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 instructions: 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

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.

```
//  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.

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.

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 *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.

[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.

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 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

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

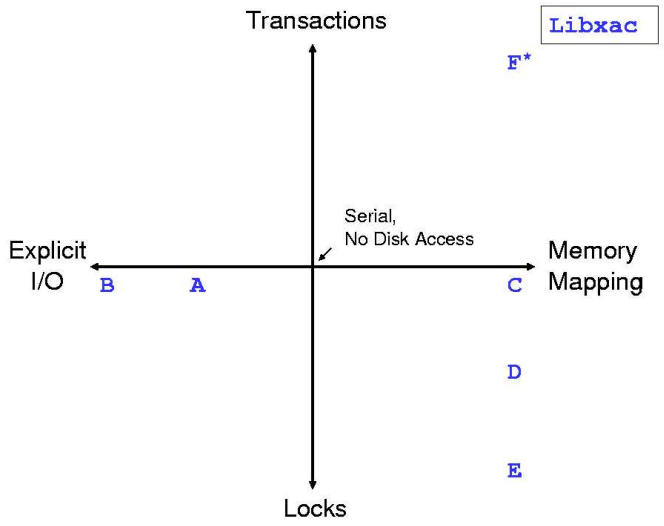

Figure 1-7: An illustration of the possible space of programs that can concurrently access data on disk.

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 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 concurrency 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.