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.

`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.

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 nondurable 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 transactions 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.

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

```
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.

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 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 each of 5 randomly selected pages in a file and stores their sum

[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.

[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.

int fileLength = 100000;
int main(void) {
  int i, j, sum = 0;
  int fd;
  int* x;
  fd = open("input.db",
            O_RDWR,
            0666);
  x = (int*)mmap(x,
                 4096*fileLength,
                 PROT_READ|PROT_WRITE,
                 MAP_SHARED,
                 fd,
                 0);


  for (j = 0; j < 5; j++) {
     i = rand() % fileLength;
     sum += x[1024*i];
  }

  i = rand() % fileLength;
  x[1024*i] = sum;



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

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

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

  xInit(".", NONDURABLE);


  x = (int*)xMmap("input.db",
                  4096*fileLength);




  while (1) {
    xbegin();
    for (j = 0; j < 5; j++) {
       i = rand() % fileLength;
       sum += x[1024*i];
    }

    i = rand() % fileLength;
    x[1024*i] = sum;
    if (xend()==COMMITTED) break;
  }

  xMunmap("input.db");
  xShutdown();
  return 0;
}
```

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

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 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

[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`.

```
int sum(tree* t) {
   int answer = 0;
   while (1) {
     xbegin();
      if (t == NULL) answer = 0;
      else {
        answer = t->value + sum(t->left) + sum(t->right);
      }
     if (xend() != ABORTED) break;
   }
   return answer;
}
```

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

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

[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.

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.

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, its changes to the shared-memory segment are discarded.

[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.

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

```
// Program 1

int main(void) {
  int y = 0, z = 0; a = 0, b = 0;
  int* x;
  xInit(".", NONDURABLE);
  x = xMmap("input.db", 4096);

  while (1) {
    a++;
    xbegin();
    b += x[0];
    y++;
    x[0]++;
    z += (x[0] - 1);
    if (xend()==COMMITTED) break;
  }

  munmap(x, 4096);
  xShutdown();
  return 0;
}
```

```
// Program 2

int main(void) {

  int* x;
  xInit(".", NONDURABLE);
  x = xMmap("input.db", 4096);

  while (1) {

    xbegin();


     x[0]--;

    if (xend()==COMMITTED) break;
  }

  munmap(x, 4096);
  xShutdown();
  return 0;
}
```

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

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 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:

[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.

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 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].

[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.

[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].

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 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 concurrency 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.