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]

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 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.

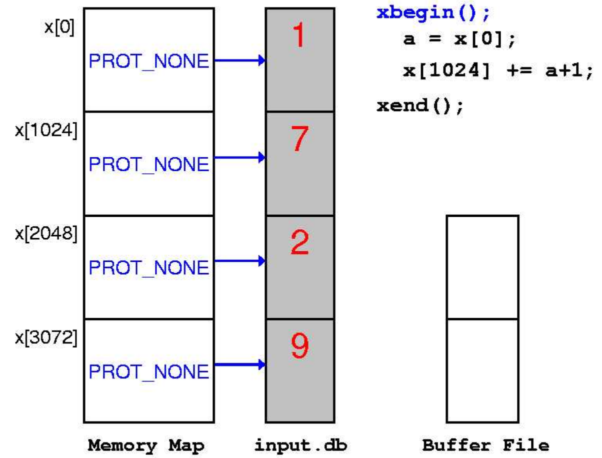
1. The `xMmap` call initially maps the entire shared-memory segment (`x` in this example) for the current process with no-access memory protection (`PROT_NONE`). The `xbegin` call starts the current transaction with a status of `PENDING`.

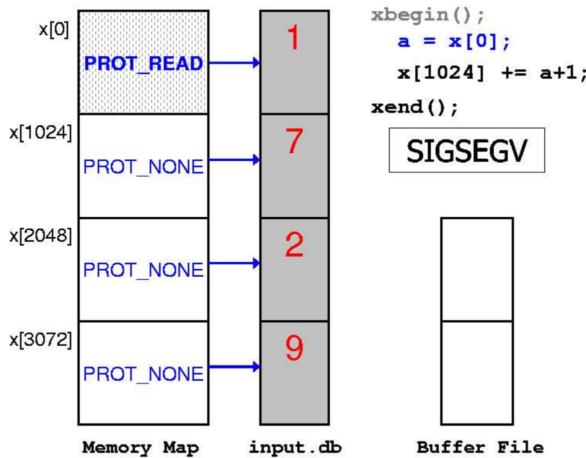
2. Line 6 causes a segmentation fault when it attempts to read from the first page of mapped file. Inside the `SIGSEGV` handler, LIBXAC checks the global transaction state to determine whether one or more transactions need to be `FAILED` because this memory access causes a conflict. As described in Section 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`.

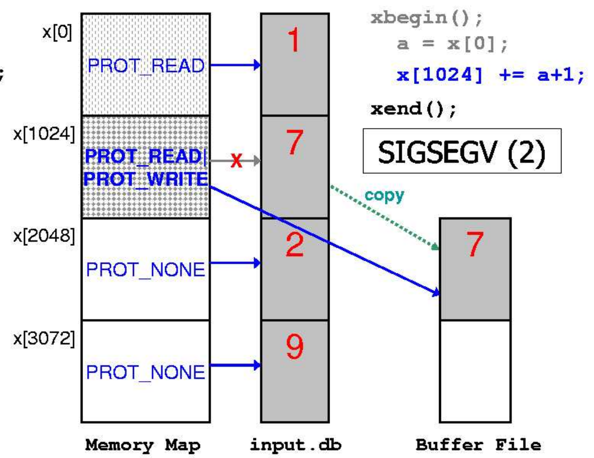
[1]The `man` page states that when `fsync` returns, the data may not actually be written to disk if the harddrive's write-cache is enabled.

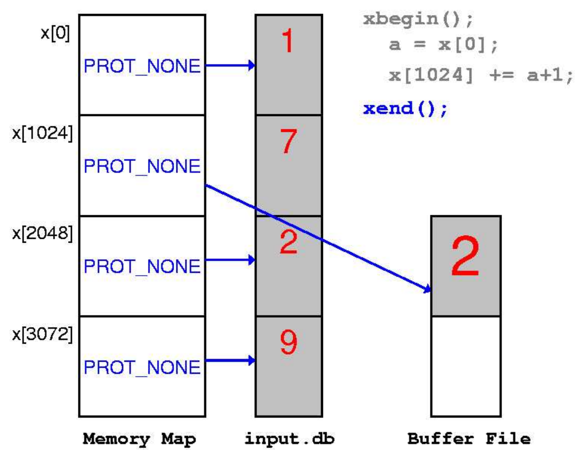
[2]The readset and writeset are defined to be disjoint sets.

```
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  }
```









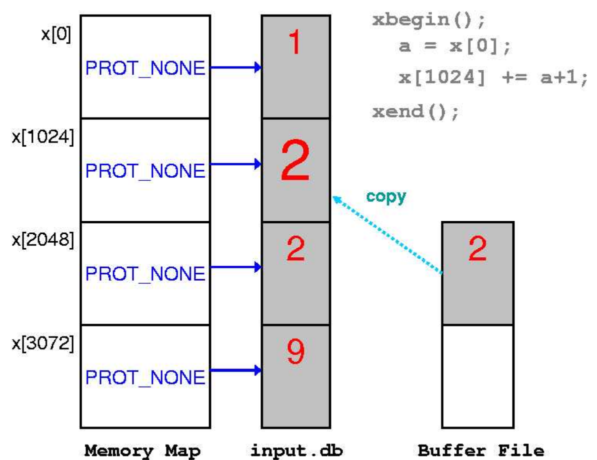


Figure 3-1: Changes to the memory map for a simple transaction.

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, during 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.

[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.

- 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 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 protections.

[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.

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 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

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

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 that a parent-child relationship between two committed transactions corresponds to a valid serial ordering of the two

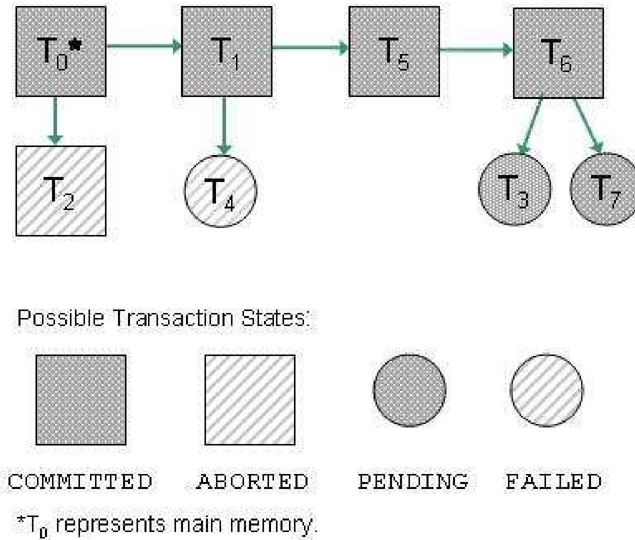


Figure 3-2: An example of a consistency tree.

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 serializable 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.

  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.

[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.

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. Transaction 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.