Chapter 6

Conclusion

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

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

6.1 Ideas for Future Work

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

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

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

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

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

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

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

6. Because Libxac maintains multiple versions of a given page, it is not immediately clear whether the cache-oblivious property of a CO B-tree is maintained when insertions are performed as memory-mapped transactions, or when insertions happen in parallel. It would be interesting to investigate whether cache-oblivious algorithms can retain their optimal caching behavior in a memory-mapped transaction system.

6.2 Libxac and Transactional Memory

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

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

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

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

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

UTM supports unbounded transactions by spilling cache lines from transactions into main memory and handling that transaction in software. The mechanisms for doing this are fairly complicated

however. With access to a system such as LIBXAC, however, it may be possible to support unbounded transactions with only the simpler hardware scheme of [24]. In what follows, I sketch one possible proposal for integrating the various transaction systems.

A Generic Transactional Memory Hierarchy

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

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

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

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

The tricky part is determining how the two levels can communicate with each other. I sketch a simple scheme that tries to keep the two levels as independent of each other as possible. I propose that at every level of the hierarchy, a block can conceptually be in one of the following states: N, R, W, L, or U.

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

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

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

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

- **U**: An upper level TM system is handling this block. For example, if block q_1 is in state U at level 1, then every small block in the larger block $f_2(q_1)$ that is in the level 1 cache must also

be in state U. In this example, a level 1 transaction is not allowed to access any memory in block $f_2(q_1)$, but level 2 transactions can.

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

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

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

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

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

A Specific Example

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

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

- In HTM mode, a process can have multiple threads, each possibly accessing the shared-memory segment. The restriction is, however, that no level-2 transactions can be executed on this process while in HTM mode.
- Similarly, in LIBXAC mode, the process must execute serially, and no level-1 transactions can be executed.

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

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

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

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

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

Unresolved Questions

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

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

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

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

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

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

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

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

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

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