

# Supporting Intel Transactional Synchronization Extensions in QEMU

Sebastien Dabdoub      Stephen Tu  
{sdabdoub,stephent}@mit.edu  
MIT CSAIL

## 1 INTRODUCTION

Concurrent programming in shared memory multiprocessor (SMP) systems is challenging. One of the difficulties is that programmers are accustomed to reasoning about single-threaded execution. Locking primitives help to make concurrent execution easier to reason about by providing mutual exclusion of critical sections, but often suffer from poor scalability [3]. This forces programmers to explicitly reason about finer grain locking strategies, which is often error-prone.

A different approach is to use transactional memory (TM), which combines the same ease-of-use as coarse grain locking with the performance benefits of finer grain locking. There has been a lot of work in the academic literature about both hardware and software based TM systems [8][5][6], but only recently has a mainstream implementation been proposed by Intel. Transactional Synchronization Extensions (TSX) is a new specification recently introduced by Intel for the upcoming Haswell multicore processor [1].

The specification introduces a new software interface for executing instructions transactionally. This includes three new main instructions: `XBEGIN`, `XEND`, and `XABORT`. `XBEGIN` informs the processor to start executing in transactional mode. All subsequent changes to the architectural state (including memory) will not be exposed to other processors until commit, and will be exposed atomically. `XEND` signals the processor to atomically commit writes which have accumulated since the corresponding `XBEGIN`. `XABORT` signals the processor to revert the current processor's architectural state to before the corresponding `XBEGIN` was issued.

Because this is a new hardware specification, currently no hardware implementation is available to researchers outside of Intel. Since this is a promising specification, we wanted to start experimenting with it. Specifically, we want to understand how useful the new primitives are to both OS kernel developers and userspace application developers. This served as the primary motivation for this paper: to implement a fully compliant emulation of TSX in QEMU [2], the popular open source x86 emulator.

## 2 DESIGN AND IMPLEMENTATION

To achieve a working implementation of TSX, there are two main sets of design decisions to be made. The first set of design decisions deals with trying to faithfully emulate the semantics of TSX, so that code running on top of our emulation would run similarly to code running on real processors. The second set deals with providing a reasonably efficient implementation in QEMU, so that developing code using our emulation is practical. It is important to note that both sets of decisions are necessary for our work to be useful.

### 2.1 Conceptual design

We settled on a design very similar to how database systems implement transactions. Our system divides target physical memory into cache-line size units (a cache-line on x86 processors is typically 64 bytes wide), and assigns a read/write lock per cache-line.

**Processors in transactional mode.** We first focus on how processors running in transactional mode interact with the shared memory. When a processor in transactional mode wants to access a

cache-line, it must first acquire the lock in the appropriate mode. Our system allows multiple processors to own read locks on a cache-line, but only a single processor can own a write lock. We use strict two-phase-locking (2PL), so a processor does not release its accumulated read/write locks until the end of a transaction (abort or commit).

Each processor maintains a per-CPU store buffer (this is not to be confused with the store-buffer optimization used by modern multiprocessors). When we do a transactional store, we do not write the changes directly to shared memory. Instead, we buffer the changes in the store buffer (in cache-line sized units). Transactional reads work by first checking the store buffer for the cache-line, and then falling back to shared memory. A transaction commit iterates through its per-CPU store buffer and writes each cache-line back into shared memory. A transaction abort simply discards its store buffer. We checkpoint all CPU registers on a `XBEGIN`, so that an abort can restore CPU register state.

If a processor cannot successfully acquire the necessary lock, it aborts its current transaction. While we could explore alternate aborting strategies which abort based on priorities, for instance, or try to wait to see if the lock becomes available (as is the case in database systems), we stick with this simple design since we have no reason to believe the Intel processors will behave differently. This is because any more complicated scheme will require more book-keeping, and is thus less feasible for a hardware implementation.

Other actions, besides read/write conflicts, that cause a processor to abort its current transaction include any instruction which generates an interrupt. This means that, for instance, a processor running a transaction cannot make a syscall or trigger a page fault without aborting. The specification suggests that this will be standard behavior in at least some, if not all, implementations.

**Processors in non transactional mode.** We now focus on how non transactional processors interact with the shared memory. Unfortunately, the Intel specification is vague on this point. One possible implementation would be to do nothing special and treat this behavior as undefined. In other words, we could make transactions atomic only with respect to other transactions. However, we decided to implement a stronger consistency model, where we make transactions atomic with respect to all other processors, regardless of mode. Therefore, a non transactional processor interacts with the shared memory by first checking to see if the cache-line it wants to write to is currently locked in an incompatible manner. If it is not, it proceeds with the load or store. If it is, then it first aborts all the transactions holding locks on the cache-line, and then proceeds with the load or store. Conceptually, this is equivalent to wrapping each load/store within a single transaction, except we guarantee that it *never* aborts. We provide this guarantee since non transactional code has no notion of aborting. An alternative implementation could not abort other transactions, but keep trying each non transactional load/store until it is able to succeed. This is equivalent to supplying an abort handler equal to the instruction itself.

**Correctness.** Because we use strict 2PL, our implementation guarantees serializability of transactions. Furthermore, our scheme cannot deadlock, because our locks do not wait for acquisition, they

simply abort themselves. For example, suppose processors  $P1$  and  $P2$  are executing transactionally. Now suppose there are two variables  $x$  and  $y$  on separate cache-lines, and  $P1$  holds a read-lock on  $x$  while  $P2$  holds a read-lock on  $y$ . Now say  $P2$  tries to acquire a write-lock on  $x$  at the same time  $P1$  tries to acquire a write-lock on  $y$ . In our design, the first processor which tries to acquire the write-lock will abort itself, allowing the other processor to successfully acquire the other write-lock and proceed.

## 2.2 QEMU implementation

The design outlined above is conceptually simple. However, realizing such a design in QEMU turned out to be surprisingly non-trivial. We originally started our implementation using QEMU's user-space mode, which allows for emulation of a single user process instead of full system emulation. However, because QEMU's SMP support in user-space mode is broken, we found our implementation to be not of very much practical use, since we could not test it with any concurrency. We therefore re-implemented TSX support in full system emulation mode. For the remainder of the paper, we will only discuss the full system emulation implementation. In this section, we outline a few of the challenges we faced.

**Trapping all load/stores.** Probably the biggest challenge faced was trapping all loads and stores in QEMU. QEMU implements a software-based MMU, which in turn emulates a software TLB. To implement a load/store, QEMU generates native code to lookup a virtual address in the TLB during guest binary translation. In the case of a TLB hit, the generated code directly does the load/store without calling into a helper function, for performance reasons. Only in the case of a TLB miss does the generated code trap into a helper function. We explored a variety of different solutions here, but ultimately decided it was simpler to disable the TLB hit case and have all loads/stores trap into a shared helper. Once we did this, it was reasonably straightforward to place a few hooks after virtual to physical address translation but before actually performing the load/store, which allowed us to control whether or not to perform the load from shared memory or a per-CPU store buffer, and whether or not to perform the store to shared memory or a per-CPU store buffer.

**Handling hardware interrupts.** Software interrupts (such as those triggered by an INT instruction) are straightforward to handle in transactions, since we just need to hook into the routines which raise CPU level interrupts. However, asynchronous interrupts, such as those delivered by the hardware timer, are tricky because they occur while a CPU is current executing a transaction. Because we found that most of the hardware interrupts being delivered during a CPU transaction were hardware timer interrupts, we worked around the problem by having XBEGIN mask interrupts, and then having XEND restore the previous interrupt mask. While this is not the most ideal solution, it does allow transactions to make more progress (otherwise, for instance, a timer tick could abort a transaction).

**Aborting transactions on other processors.** It is reasonably straightforward in QEMU to abort a transaction running on the current processor, but what about aborting transactions running concurrently on other processors? The key insight is that an abort does not need to happen until the very end of the transaction (it is ok to allow transactions which we know will need to abort to keep running as long as we *eventually* abort it). We give each processor an `aborted` flag. This flag is set by other processors to signal an abort. XEND simply checks this flag to see if the processor should be allowed to commit. As an optimization, we also have loads and stores check this flag, so we do not need to wait until the end of the transaction to abort.

## 3 EVALUATION

In order to evaluate our implementation we have two main considerations: testing the TSX implementation and implementing interesting code samples. The first consideration provides evidence that our implementation exhibits the desired behavior of the Intel extensions. The second consideration provides evidence that the extensions simplify concurrent programs as compared with lock-based synchronization. Specifically, we implemented a concurrent hash table using transactional memory and course-grained locking.

We decided to evaluate our implementation by running user processes in the xv6 operating system [4]. We exposed the hardware extensions to user-level programs in xv6 as a C library with the following API:

```
unsigned int _xbegin(void);
void _xend(void);
void _xabort(int);
```

Every transaction begins by calling `_xbegin()` and ends by calling `_xend()`. A transaction can abort by calling `_xabort()`. `_xbegin()` returns 0 if a transaction was successfully started. When a transaction aborts, `_xbegin()` returns 1 (this is similar in spirit to the `setjmp/longjmp` API). The intended usage pattern is:

```
if (_xbegin() == 0) {
    /** critical section */
    _xend();
} else {
    /** abort handler */
}
```

We currently use the simple usage pattern:

```
retry:
if (_xbegin() == 0) {
    /** critical section */
    _xend();
} else {
    goto retry;
}
```

We have not investigated more complex abort handlers- we leave this to future work.

**Testing the implementation.** In order to test our implementation, we came up with seven test cases that cover the basic behavior of transactions. Four of those test cases only involve a single CPU. These cases include testing that transactional writes get committed on a successful transaction, that aborting a transaction restores the system memory, that aborting a nested transaction aborts the top level transaction, and that using a syscall in a transaction aborts the transaction.

The other three tests check the expected behavior of concurrent transactions. We implemented a shared memory fork syscall in xv6 in order to have shared pages between processes, since fork provides for separate address spaces. We test for two concurrent transactions that both successfully complete if they are writing to different shared variables (on different cache-lines), that both successfully complete if both are only performing reads to the same shared variables, and that one aborts if they are both using the same shared variable and one of the performs a write.

**Code simplification.** To demonstrate the code simplification, we implemented a concurrent hash table and a locking mechanism.

The hash table resembles the standard implementation with coarse-grained locking, but instead using transactional memory. We protect the hash table from data races using our programming pattern for transactions. Fig 1 shows what a simplified set call of a concurrent hash table looks like using transactional memory as compared to coarse-grained locking. While the two implementations have comparable code complexity, the transactional memory implementation is more concurrent; transactions will not abort from writing concurrently to different variables. Therefore, if we make many concurrent writes to distinct entries in the table, the transactional memory implementation will allow them to proceed concurrently, whereas the coarse-grained locking implementation will serialize these writes. In order to achieve similar concurrency, we would require fine-grained locking and its additional complexities. While fine-grained locking for a hash tables is straightforward, we use this example to demonstrate how transactional memory achieves the concurrency of fine-grained locking with the complexity of coarse-grained locking.

```

TM
retry:
if (_xbegin() == 0) {
    unsafe_set(&table, key, val);
    _xend();
} else {
    goto retry;
}

Coarse Grained Lock
pthread_mutex_lock(&mutex);
unsafe_set(&table, key, val);
pthread_mutex_unlock(&mutex);

```

**Figure 1:** HTM vs Coarse grained lock protecting hash table set call.

## 4 CONCLUSION

This paper introduces our implementation of Intel’s upcoming support for hardware transactional memory in the QEMU x86 emulator. We tried to be faithful to the specification, but hardware specifications are notoriously vague [7]. Where the specification was vague, we tried to make reasonable behavioral choices based on what a programmer would want and what we believe is feasible to implement in hardware.

This paper also demonstrates a simple exploration of concurrent programming using transactional memory. Unfortunately, there is one issue that our emulation of TSX cannot answer; we will have to wait for real hardware before we can start thinking about performance.

## REFERENCES

- [1] Intel architecture instruction set extensions programming reference. Intel Corp., February 2012.
- [2] F. Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.
- [3] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.
- [4] R. Cox, C. Frey, X. Yu, N. Zeldovich, and A. Clements. Xv6—a simple unix-like teaching operating system.
- [5] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.
- [6] B. Saha, A.R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 185–196. IEEE, 2006.
- [7] P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, and M.O. Myreen. x86-tso: a rigorous and usable programmer’s model for x86 multiprocessors. *Communications of the ACM*, 53(7): 89–97, 2010.
- [8] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM, 1995.