ECE459: Programming for Performance Winter 2011

Lecture Bonus — March 29, 2011

Patrick Lam *version 1*

This material is not on the final exam. I ad-libbed some material about software transactions today. Here are some notes to go with that.

Basic Idea. Developers use software transactions by writing `atomic` blocks. These blocks are just like `synchronized` blocks, but with different semantics.

```
atomic {
  this.x = this.z + 4;
}
```

You're meant to think of database transactions, which I expect you to know about. The `atomic` construct means that either the code in the atomic block executes completely, or aborts/rolls back in the event of a conflict with another transaction (which triggers a retry later on).

Benefit. The big win from transactional memory is the simple programming model. It is far easier to program with transactions than with locks. Just stick everything in an atomic block and hope the compiler does the right thing with respect to optimizing the code.

Drawbacks. As I understand it, three of the problems with transactions are as follows:

- I/O: Rollback is key. The problem with transactions and I/O is not really possible to rollback. (How do you rollback a write to the screen, or to the network?)
- Nested transactions: The concept of nesting transactions is easy to understand. The problem is: what do you do when you commit the inner transaction but abort the nested transaction? The clean transactional fa cade doesn't work anymore in the presence of nested transactions.
- Transaction size: Some transaction implementations (like all-hardware implementations) have size limits for their transactions.

Implementations. Transaction implementations are typically optimistic; they assume that the transaction is going to succeed, buffering the changes that they are carrying out, and rolling back the changes if necessary.

One way of implementing transactions is by using hardware support, especially the cache hardware. Briefly, you use the caches to store changes that haven't yet been committed. Hardware-only transaction implementations often have maximum-transaction-size limits, which are bad for programmability, and combining hardware and software approaches can help avoid that.

Re-entrant locking and abstractions. We had a question about abstraction and locking. Many lock implementations are re-entrant, so you can at least acquire the same lock more than once in the same thread. However, it turns out that locks are generally worse for abstraction than transactions: you need to know which locks your callees are going to acquire before you call them. Otherwise you risk deadlocks by obtaining locks in the wrong order. There are also performance implications of locks, so that tends to be problematic in terms of libraries, which are supposed to hide these kinds of details from you.

Transactions are generally better, but nested transactions are still a problem, as discussed above.