Transactional Memory Simulator Project Checkpoint

Schedule and Goals

My original schedule had the following due by April 26th

- April 12th: Project MVP
- April 19th: Data versioning implementations
- April 26th: Conflict detection implementations

Of these I have completed the project MVP and the data versioning implementations. I have not fully completed the conflict detection implementations, but I have implemented transaction rollbacks. It is unlikely that I will be able to achieve my extra goals. Additionally I will likely have to remove the goal of "Workload generators" and replace it with a goal of manually creating workloads for testing.

Updated Goals

- [x] MVP of a simulator that can load and store memory, but does not detect conflicts, keep data versions, or attempt to abort transactions.
- [x] Eager data versioning implementation
- [x] Lazy data versioning implementation
- [x] Transaction rollback implementation
- [ ] Pessimistic conflict detection implementation
- [ ] Optimistic conflict detection implementation
- [ ] Manually generate workloads and benchmark those workloads

Update Schedule

- April 26th - 28th: Pessimistic conflict detection
- April 28th - May 1st: Optimistic conflict detection
- May 1st - May 5th: Benchmarking
- May 5th - May 10th: Report

Implementation Summary

MVP

The first task involved creating an MVP of a transactional memory simulator. To do this I created a class called `TransactionManager` that exposes the following API, which is slightly changed from the project proposal.

```
/**
* Begin memory transaction
* @return transaction id
*/
uint64_t xbegin();

/**
 * Commit memory transaction
 *
```

```
 * @param transaction_id id of transaction to commit
 *
 * @throws TransactionAbortException
 */
void xend(uint64_t transaction_id);

/**
* Store value at address for transaction
*
* @tparam T type of value
* @param address location to store value
* @param value value to store
* @param transaction_id id of transaction
*
* @throws TransactionAbortException
*/
template<typename T>
void store(T *address, T value, uint64_t transaction_id);

/**
* Loads value from address for transaction
*
* @tparam T type of value
* @param address location that value is stored
* @param transaction_id id of transaction
*
* @return value stored at address
*
* @throws TransactionAbortException
*/
template<typename T>
T load(T *address, uint64_t transaction_id)
```

The MVP provided no actual transactional semantics and therefore `xbegin` and `xend` were both NoOps. Load simply returned the value stored at the address and store saved the value at the given address. However callers could use this API to interact with memory.

Eager Data Versioning

Eager data versioning maintains an undo log for every transaction. For every write that a transaction makes, the undo log tracks what the value was at the address in memory before the transaction. That way if the transaction aborts then we can undo all the changes via the undo log.

We implement this with one global hash map that maps transaction ids to another nested hash map. The nested hash map maps memory addresses to values, where the value indicates what the value at that address was before the transaction. If a transaction writes to the same location multiple times, then we only care about the earliest write because that's the value that we will undo to in the case of an abort.

Lazy Data Versioning

Lazy data versioning maintains a write buffer for every transaction. Instead of applying the write to memory during the transaction, we track the writes that a transaction has made and apply them all at

once during commit time. That way if a transaction aborts, then we can just throw away it's write buffer.

We implement this very similarly to the undo log. We have a global hash map that maps transaction ids to another nested hash map. The nested hash map maps memory addresses to values, where the value indicates what the write to a location in memory will be. If a transaction writes to the same location multiple times, then we only care about the most recent write because that's the value that will actually get written during commit.

Presentation

At the end of the project I plan to present graphs and charts that display both the runtime and number of aborts for various workloads under different transactional settings. So for a single workload I will run it 3 times using the following settings:

1. Eager data versioning and pessimistic conflict detection
2. Lazy data versioning and optimistic conflict detection
3. Lazy data versioning and pessimistic conflict detection
4. ~~Eager data versioning and optimistic conflict detection~~ This is not *easily* possible
    1. If a conflict is detected during commit then somehow we'll need to identify which transaction wrote to the conflicting areas first. That way we know which value to undo to. To do this, it's not enough to know the timestamp of the transaction. We would need to know the timestamp of every write in the transaction. While this is possible, the overhead involved is not worth it.

Issues

The biggest issue is the global hash map. It serves as a single point of contention for all transactions. Starting a transaction and committing a transaction both require exclusive locks on the hash map to insert a new transaction and remove committed ones. Every write requires a shared lock on the hash map to update the specific transactions undo log/write buffer. One potential solution to this would to not have a global hash map and instead give every transaction their own hashmap. Instead of having a global transaction manager, each transaction would have it's own Transaction object. Callers of the transaction manager would get a Transaction object when they start a transaction, and they would call `load` and `store` on these object.