Transactional Memory Simulator Project Proposal

Joseph Koshakow (jkoshako@andrew.cmu.edu)

Summary

I am going to implement a transactional memory simulator that can accept multiple concurrent requests

Background

An increasingly popular tool in parallel programming is Transactional Memory. Not only does it allow you to read and write to and from locations in memory, it also allows you to perform transactions with multiple memory operations. Transactional memory abstracts away transactional logic from the application logic and into the memory, allowing programmers to focus on program logic and not memory consistency. Additionally, the underlying transactional implementation can optimize the memory operations for specific hardware.

In 15-213 we implemented our own version of `malloc` which allowed users to dynamically allocate and deallocate memory on the heap. I plan on using this as inspiration to create my own transaction memory in software. I plan on implementing both eager and lazy versioning and comparing the performance under various workloads, and implementing both optimistic and pessimistic conflict detection and comparing the performance under various workloads.

The simulator will be able to handle multiple concurrent transactions all running in parallel.

I hope to implement the following API

```
/**
   * 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 Challenge

Transaction processing and transaction consistency is very difficult to do correctly. Databases accomplish somthing very similar with operations on database records instead of operations on memory. They have an entire subset of the field dedicated to properly handling transactions.

ACI~~D~~

Transactional memory operations must adhere to all of the ACID properties of transactions, minus the D. One of the challenging aspects of transactional memory is how can we guarantee all of these properties when multiple transactions are running in parallel.

Atomic

When committing a transaction, either all of the operations must take effect at once or none of the operations take effect. It would be incorrect to end in a scenario where some of the operations happened successfully while others did not. Imagine a bank transaction where money was taken out of one account but then never placed in another.

Consistent

The transactions must appear to have executed in some serial order even though they are actually all being run in parallel. It doesn't matter what the serial order is as long as there is some serial order. This means that the end state of memory must be equal to the end state of memory had you run all transactions serially.

Isolation

No transactions can view the result of a different transaction before it commits.

~~Durability~~

Memory is not durable, so we do not actually care about memory transactions being durable. This would be similar to an in-memory database with no Write Ahead Log.

Conflicts

Another challenge is what do we do when one of these properties is violated? Specifically, how do we deal with conflicts between two or more transactions?

Read-Write Conflicts

When one transaction reads from a location in memory and another transaction writes to this location in memory we have the potential to violate the consistent and isolation properties because we are either reading a stale value or an uncommitted value. How do we solve this issue? We can stall one of the transactions, we can abort one of the transactions, or we can restart one of the transactions.

Write-Write Conflicts

When two transactions try and write to the same location in memory we are violating the consistent and isolation properties because we are overwriting committed data. How do we solve this issue? We can abort one of the transactions, or we can restart one of the transactions.

Read-Read ~~Conflicts~~?

When two transactions read the same location in memory this is not a conflict.

Resources

I plan on implementing this using Ubuntu 20.04 on my own personal computer. However, since this is just a simulator it should be able to run on any OS or machine type. I am starting this project from scratch. I will be using lecture notes from this class and my database class, my malloc lab implementation, and online resources as references.

Goals and Deliverables

- MVP of a simulator that can load and store memory, but does not detect conflicts, keep data versions, or attempt to abort transactions.
- Eager data versioning implementation
- Lazy data versioning implementation
- Transaction rollback implementation
- Pessimistic conflict detection implementation
- Optimistic conflict detection implementation
- Workload generators
- Workload performance benchmarking
- Extra: implement different Contention strategies on conflict detection (aggressive stall vs abort) The presentation will involve showing charts and graphs comparing the performance of different implementations under different workloads, as well as a demo of running one of the workloads.

Platform Choice

I plan on implementing this in C++ for Ubunto 20.04. I'm choosing C++ because it has the low-level capabilities to manipulate memory directly, but also the high-level data structures to organize code more effectively. I'm choosing Ubuntu 20.04 because that's the OS on my computer and it's convenient.

Schedule

- April 12th: MVP
- April 19: Data versioning
- April 26: Conflict Detection

- May 3: Workload and Benchmarks