15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Software Transactional Memory **Date:** April 21,e2009
Scribe: Harsha Simhadri

We want to efficiently support concurrent statements of the form {`Atomic {S1; S2; ...; Sn}`} such that they are serializable. For instance, we may want to support atomic FetchAdd ({`a=*a+v;`}). We Assume that all memory systems are sequentially consistent. Serializability means that some sequential ordering of atomic blocks is consistent with respect to memory operations

26.0.1 Approach 1: Single global lock

Acquiring a single global lock for each transaction, while serializable is not efficient. It ruins parallelism in the code. Another simple approach is to check the conflict graph of the transactions, and determine the maximum independent set of the graph. This set can be run safely simultaneously. However, this does not work too well when the conflict graph has too many dependencies.

26.0.2 Approach 2: Two Phase Locking

Two phase locking is a technique developed for databases as early as the 1960s to aid atomic transactions of databases.

Goal: Avoid serializing if no conflict.

Approach: separate locks for each location.

Idea:

1. Acquire lock in all transactions reading or writing (Acquire phase)
   - Run atomic code
2. Release all locks (Release phase)

Two natural questions that arise about this approach are:

- Can it guarantee serializability?
- Can it cause deadlocks?

The answer to the second question depends on the order in which locks are acquired. If two processors acquire locks in the following order : {`P1: acq(x); acq(y)`}, {`P2: acq(y); acq(x)`}, then there is a cycle in the lock acquisition order. A solution to this problem is to impose an ordering on these locks so that the processors can acquire locks only in this order. This avoids cyclic dependencies between locks and avoids deadlocks. For this, we need to know what locations need to be locked ahead of time.

An example of unsafe locking scheme for the computation `z:=x+y` is:
`acq(z); acq(x); rel(x); acq(y); rel(y); rel(z);`

Some computation could execute between the lock release of x and lock acquire of y. If that computation had written to both x and y, then two incompatible version would be used to compute z leading to error.

Serializability can be proved similar to linearizability. Each atomic region should look like it happened at the same point somewhere between the start and the end of the computation. Here is an argument to show serializability: Consider the point where all locks are acquired. This atomic region looks like it happened at the point where all locks have been acquired. When all the locks on my memory accesses are acquired, we are guaranteed that no other processor can even look at these memory locations. Therefore other operations on these memory location can not be interleaved to make it look as though they are interleaved with the instructions during the locking period.

Problems with this approach are:

- If all transactions share a variable, even just for reading, all of them are sequentialized (will be addressed with approach 3).
- Locks can be held for a long time. One long code can block out all others (will approach 4).

26.0.3 Approach 3: write locks

The only reason we need to acquire locks for reads is to avoid read-write conflicts. But since READ-READs are OK, we are being overly protective in 2-phase locking. We can take advantage of the fact that we don't need to lock read-only location.

1. Don't acquire lock on read-only location (such as read only Java variables or functional variables). Whether an arbitrary variables in the program is read-only is difficult infer.
2. (reader-write room sync: readers have one room and writers have another) Read-lock allows any number of readers but only one write. At any given time, the lock is in the read state or the write state.

26.0.4 Approach 4: Optimistic concurrency, delayed writing

If a certain transaction can take too long, it might be disadvantageous to have other processors wait on resources held by the long transaction. Instead, we can be optimistic and compute the atomic region (writing changes locally), before attempting to acquire locks for writing to shared locations. This allows for more flexibility. The details for this are as follows:

keep the following log as computation progresses:

- Read: keep addresses of reads and their values.
- Write: keep addresses of writes and keep local values of writes.

After the computation is over, in the commit phase, local values are written to shared variables conditioned on whether the inputs of computations have changed in the duration of the computation.

- acquire locks on read (can use read locks) and write locations.

- Check if the read locations have same values. If read locations have changes, abort and try again
- write values to memory
- release locks

This however has a minor bug: the ABA problem. The read could have been written to twice with the initial value restored. Despite the fact that the location has been over written, the current transaction goes ahead and commits, which may lead to non-serializability. This can be solved by storing the value and version number for each variable. Whenever a variables is written, we increment the version number. When checking whether a relocation has changed, also check the version number.

Zombie transactions: Some transactions can read values that are parts of some incomplete transactions. These values may not be consistent with the specification of the input parameters of the transaction, and may cause the transaction's computation part to behave unexpectedly. Though this transaction has no chance of succeeding because of version checks before writing, it can nevertheless go in to infinite loops are do unexpected things that would not have happened otherwise in a sequential execution. One costly way to avoid this is to have the processor check the current versions of each of it's previous reads whenever it is reading a new location.

26.0.5 Approach 5

- Give a transaction a time stamp based on when it started (implement with FA)(unique and increasing).
- Serialize based on time stamp.

Assume that all locations have both Write timestamps (time stamp of transaction of last write) and Read timestamps. The compute phase for a transaction with time stamp `i`:

- If when attempting to read a location `x` with write time stamp greater than the time stamp `i` obtained by the current transaction, then abort and retry with a new time stamp. If not, the read just returns the current value and the transaction progresses.
- Writes are buffered.

The commit phase works as follows:

- acquire write locks.
- If for any writes `x`, if read time stamp on `x` is greater than `i` then abort; else write values and release lock. (some one with a greater time stamp (whose effects are supposed to take place after the current transaction) could have read this value before the current transaction's write and perhaps may have even committed the transaction (with future time stamp) based on current value at `x`. Therefore, the current transaction writing at `t` would be like back in time. This would break serializability based on timestamps)