Software Transaction Memory

Divyansh Singhvi Megha Agarwal

I. Introduction

Parallel programs execute in a nondeterministic way, so they are hard to test, and bugs can be almost impossible to reproduce. Current lock-based abstractions are difficult to use and make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals. Software Transactional Memory (STM) is a promising new approach to programming shared-memory parallel processors that seems to support modular programs in a way that current technology does not.

II. Problems with Locks

Fundamental shortcoming of lock-based programming is that locks and condition variables do not support good modular programming ie the process of building large programs by gluing together smaller programs. Here are some standard difficulties:

- Taking too few locks
  It is easy to forget to take a lock and thereby end up with two threads that modify the same variable simultaneously
- Taking too many locks
  It is easy to take too many locks and thereby inhibit concurrency (at best) or cause deadlock (at worst).
- Taking the wrong locks
  it is all too easy to take or hold the wrong locks
- Taking locks in the wrong order
  In lock-based programming, one must be careful to take locks in the right order. Avoiding the deadlock that can otherwise occur is always tiresome and error-prone, and sometimes extremely difficult
- Error recovery
  Can be very hard, because the programmer must guarantee that no error can leave the system in a state that is inconsistent, or in which locks are held indefinitely.
- Lost wakeups and erroneous retries
  It is easy to forget to signal a condition variable on which a thread is waiting; or to re-test a condition after a wake-up.

III. Problem Statement

If two key transactional semantics atomicity and isolation is guaranteed in execution of every block of concurrent codes then writing concurrent programs becomes quite simple. The synchronization of multi-threaded programs can be ensured by using such transactional codes. A programmer is relieved from being unduly bothered about the complexity of writing correct lock based concurrent programs. Design, develop and implement a set of APIs for multi-threaded computation where shared data is synchronized without locks. Threads synchronize through memory transactions [short-lived computations which either commit (take effect) or abort (take no effect)]. The STM should have the following properties :

A. Atomicity

B. Composability

C. Choice

IV. Atomicity

Atomicity is one of the ACID (Atomicity, Consistency, Isolation, Durability) transaction properties. An atomic transaction is an indivisible and irreducible series of database operations such that either all occur, or nothing occurs. A guarantee of atomicity prevents updates to the database occurring only partially, which can cause greater problems than rejecting the whole series outright. As a consequence, the transaction cannot be observed to be in progress by another database client. At one moment in time, it has not yet happened, and at the next it has already occurred in whole (or nothing happened if the transaction was cancelled in progress). Any code segment wrapped by key word atomic, such as,

```
do {
        condition = false;
        Transaction tx = new Transaction();
        int tem = (int) tx.read(a);
        if (tem == -1) {
            continue;
        }
        int temp = tem + 1;
        TVar<Integer> z= new TVar<>();
        z.value=temp;
        tx.write(a,z);
        condition = tx.commit();
    }while (!condition);
```

should excute all or nothing and in isolation. Each read-/write is logged in thread's local transactional memory. Writes go to the log only not the memory. At the end commit to the memory is tried, if commit fails then transaction should re-run from beginning.

V. Composability

Operations that are individually correct (insert, delete) cannot be composed into larger correct operations. Two individually-correct abstractions, p1 and p2, cannot be composed into a larger one; instead, they must be ripped apart

and awkwardly merged, in direct conflict with the goals of abstraction. Rather than fixing locks, a more promising and radical alternative is to base concurrency control on atomic memory transactions, also known as transactional memory.

A. Transactional Memory

The key idea is that a block of code, including nested calls, can be enclosed by an atomic block, with the guarantee that it runs atomically with respect to every other atomic block. Transactional memory can be implemented using optimistic synchronisation. Instead of taking locks, an atomic block runs without locking, accumulating a thread local transaction log that records every memory read and write it makes. When the block completes, it first validates its log, to check that it has seen a consistent view of memory, and then commits its changes to memory. If validation fails, because memory read by the method was altered by another thread during the blocks execution, then the block is re-executed from scratch. For eg. :

atomic {

$$x = Queue1.getItem();$$

$$Queue2.getItem(x);$$

}

If either getItem or putItem retries then whole transaction retries. That is the transaction waits until Queue1 is not empty and Queue2 is not full.

VI. Choice

The transaction s1 orElse s2 first runs s1; if it retries, then s1 is abandoned with no effect, and s2 is run. If s2 retries as well, the entire call retries but it waits on the variables read by either of the two nested transactions. Again, the programmer need know nothing about the enabling condition of s1 and s2. For eg:

```
boolean condition1 = false;
boolean condition2 = false;
do {
    Transaction tx = new Transaction();
//statement 1
    tx.read(x);
    condition1 = tx.commit();
    //statement 2
    if (!condition1) {
        Transaction t2 = new Transaction();
        TVar<Integer> temp = new TVar<>();
        temp.value = 4;
        tx.write(x, temp);
        condition2 = t2.commit();
        if(condition2)
            break;
    }else
        break;
} while (true);
```

Or Else tries two alternative path, if the first retries then it run the second. If both retries then whole orElse retries. So transaction waits until Queue1 is nonempty and either Queue2 or Queue3 is not full.

VII. Implementation

We implemented Transactional Locking II

A. *TL2 Algorithm*

- Global integer clock giving timestamps(version) is used.
  - It generates a readstamp for each transaction
  - A writestamp for each transaction that tries to commit.
- It's an optimistic approach assuming every transaction proceeds. Hence all modification done by a transaction is kept local and updated to their corresponding values when committed
- Transaction T contains
  - ReadStamp
  - WriteStamp
  - ReadSet : Contains items which are read.
  - WriteSet : Contains tentative new versions.
- When A Transaction Starts record the current clock as it's readstamp
- TVar x (Object which is modified or read) contains :
  - Stamp : Writestamp of the last transaction to write to x
  - Version
  - lock (exclusive).
- When Transaction T tries to read Tvar x:
  - If x is in T.writeset, then read X's tentative new version
  - Validation: If x is locked or x.stamp $>$ T.readstamp, then T aborts.
  - Otherwise, read x.version and add x to T.readset.
- When Transaction T tries to write Tvar x:
  - If x is in T.writeset, then overwrite Xs tentative new version.
  - Otherwise, add x to T.writeset with the tentative new version.
- When Transaction T tries to commit:
  - T locks all the objects in its write set
  - Reads and increments the global clock (fetchincrement), stores new value as Ts writestamp.
  - For every x in T.readset check if x is locked by another transaction or x.stamp $>$ T.readstamp then T aborts
  - Installs the new versions in all the objects, with T.writestamp
  - Releases all the locks.

B. *Two-phase locking*

- Each TVar2 has the following fields:
  - old-version, value of object x before any changes by the current lock-holder

  - new-version, value of object x including the current lock-holders updates
  - lock
- When Transaction T reads TVar2 x:
  - if x.lock == (exclusive,T.TransactionId), then T.read returns x.new-version
  - if x.lock == (shared,T.Transactionid) , then T.read old-version
  - if x.lock == (exclusive,not T.TransactionId) then T Aborts. Releases locks and discards it's new-versions
  - if x.lock != (shared, T.TransactionId) —— x.lock == (unlocked) then x.lock.insert(shared,T.TransactionId)
- When transaction T tries to write TVar2 x
  - If x.lock == (exclusive,T.TransactionId) then new-version = updatedValue
  - if x.lock = (shared, T.TransactionId) size(x.lock) == 1 then x.lock.upgrade(T)
  - if x.lock == unlock then x.lock.upgrade(T)
  - if x.lock != (shared, T.TransactionId) || x.lock != (exclusive, T.TransactionId) then x.new-version = updatedValue

VIII. Data Structures

- Thread Program increments a variable to verify correctness
- Array Program to check the Array Implementation of the code.
- HashMap with get and add methods.
- SkipList with get and add methods.

IX. Pointers to Understand code

A. Layout

- /src is the source directory.
  - TVar is the Transaction Variable for TL2 algorithm
  - TransactionId is the class to get global transaction id and increase count atomically
  - Clock is the class to get global transaction id and increase count atomically
  - Transaction is the class which contains the actual algorithm for TL2. Contains : ReadTVar, WriteTVar, commit functions,retry functions
  - STMArray is the implementation of Arrays for STM library
- /test is the test directory
  - FirstSimpleTransaction file works on single variable and updates it atomically
  - TestArray test the STMArray implementation by using get and set.

X. Correctness

First, let us discuss the point of **serializability**.

- To define serializability of commits in our algorithm, we must consider the point when the committed Transaction T determines T.writestamp.
- Hence transactions are serializable in the order of their writestamps.

A. Explaination

- Whenever T commits, the following conditions hold.
- Every version T read must have been written by some other transaction with writestamp $<=$ T.readstamp ie. Every object read by T(Objects in readSet) must have been written by some other transaction before the read-stamp of T otherwise our commit function returns a false.
- For each object x, T must read the version of x with the largest writestamp $<=$ T.readstamp, since they are already written when T starts.
- Final validation check (If x is locked by another transaction or x.stamp $>$ T.readstamp then T aborts) ensures that no transaction A with T.readstamp $<$ A.writestamp $<$ T.writestamp wrote on any object present in T.readset.
- Hence we could see that no transaction affects another transactions even if operating on same values and we came with the above invariant that no transaction A with T.readstamp $<$ A.writestamp $<$ T.writestamp wrote on any object present in T.readset

XI. Future Work

For future work we would like to improve the implementation by providing support for exceptions. Techniques for handling overflows of globalClock and transactionId can be implemented. Making a wrapper over the current implementation could be improved to have a more native feeling of language.

References

[1] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions, ACM Conference on Principles and Practice of Parallel Programming 2005 (PPoPP'05).

[2] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. Lock -Free Data Structures using STMs in Haskell, Eighth International Symposium on Functional and Logic Programming, April 2006 (FLOPS'06).

[3] Tim Harris and Simon Peyton Jones. Transactional memory with data invariants, First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06), 11 June 2006, Ottawa, Canada.

[4] B. Smith, An approach to graphs of linear forms (Unpublished work style), unpublished.

[5] Beautiful Code by Greg Wilson; Andy Oram Published by O'Reilly Media, Inc., 2007

[6] https://queue.acm.org/detail.cfm?id=1454466

[7] https://link.springer.com/content/pdf/10.1007/978-3-642-22045-6_3.pdf