Language-level Non-blocking Software Transactions (in Java!)

C. Scott Ananian
cananian@csail.mit.edu

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Transactions (review)

- A transaction is a sequence of loads and stores that either commits or aborts.
- If a transaction commits, all the loads and store appear to have executed atomically.
- If a transaction aborts, none of its stores take effect.
- Transaction operations aren’t visible until they commit or abort.
Non-blocking synchronization

- Although transactions can be implemented with mutual exclusion (locks), we are interested only in non-blocking implementations.
- In a non-blocking implementation, the failure of one process cannot prevent other processes from making progress. This leads to:
  - Scalable parallelism
  - Fault-tolerance
  - Safety: freedom from some problems which require careful bookkeeping with locks, including priority inversion and deadlocks.
- Little known requirement: limits on transaction suicide.
Monitor Synchronization

public class Count {
    private int cntr = 0;
    void inc() {
        synchronized(this) {
            cntr = cntr + 1;
        }
    }
}

- Traditionally, monitors associated with each object provide mutual exclusion between concurrent accesses to the object.
Monitor Synchronization

```java
public class Count {
    private int cntr = 0;
    void inc() {
        synchronized (this) {
            cntr = cntr + 1;
        }
    }
}

Instead we provide an **atomic** block, and make linearizability guarantees without (necessarily) providing mutual exclusion.
```
Implementation Idea

Traditional

Transactional
A software transaction impl.

- **Goals:**
  - Non-transactional operations should be fast.
  - Reads should be faster than writes.
  - Minimal amount of object bloat.

- **Solution:**
  - Use special `FLAG` value to indicate “location involved in a transaction”.
  - Object points to a linked list of *versions*, containing values written by (in-progress, committed, or aborted) transactions.
  - Semantic value of a `FLAGged` field is: “value of the first version owned by a committed transaction on the version list.”
Transactions using version lists

Object #1

- MyClass
  - type
  - versions
  - (TID68)
    - readers
    - FLAG
    - field1
    - 3.14159
    - field2

Object #2

- OtherClass
  - type
  - versions
  - (TID25)
    - readers
    - 2.71828
    - FLAG
    - field1
    - field2

Version

- Transaction ID #68
  - WAITING
    - status
    - owner
    - next
  - Version
    - field1
    - 23
    - field2

Version

- Transaction ID #56
  - COMMITTED
    - status
    - owner
    - next
  - Version
    - field1
    - 55
    - field2

Version

- Transaction ID #23
  - COMMITTED
    - status
    - owner
    - next
  - Version
    - field1
    - 'A'
    - field2

Version

- Version
  - field1
  - 'B'
  - field2

Transaction ID #68 Transaction ID #56 Transaction ID #23

... ... ...

Ananian, CRS retreat – p. 7
Races, races, everywhere!

- Lots of possible races:
  - What if two threads try to $\text{FLAG}$ a field at the same time?
  - What if two threads try to copy-back a $\text{FLAGged}$ field at the same time?
  - What if two transactions perform conflicting updates?
  - Do transactions commit atomically?
- Formulated model in Promela and used Spin to verify correctness (for bounded scope, etc).
Bugs found with model-checking

- Memory management (object recycling, reference counting)
- Read caching (check copies to local variables)
- “Real” bug: missing abort of readers during non-transactional write
Bugs found with model-checking

- Memory management (object recycling, reference counting)
- Read caching (check copies to local variables)
- “Real” bug: missing abort of readers during non-transactional write

Too much time spent minimizing/coalescing state. =(  

Ananian, CRS retreat – p. 9
More Fun

- Large objects
- Interaction with I/O
- Interaction with native methods
- Nested transactions
- Exposing abort/retry mechanism
- Supporting wait/notify
Cooperating HW/SW transactions

- Using “node-push” micro-benchmark with a hardware transaction mechanism (submitted ASPLOS-XI)
- Hardware starts to perform poorly for large or long-lived transactions.

![Graph showing the performance of HTM and STM transactions over transaction size](image)

*HTM Transactions stop fitting after this point*
Cooperating HW/SW transactions

- Using “node-push” micro-benchmark with a hardware transaction mechanism (submitted ASPLOS-XI)
- Hardware starts to perform poorly for large or long-lived transactions.
Optimistic parallelism

for (...) optimistically {
    ...do an iteration ...
}

conquer(A[n], n) {
    ...
    optimistic spawn
    conquer(A, n/2);
    optimistic spawn
    conquer(A+n/2, n-n/2);
}

Programmer notes that the iterations or spawns are expected to be independent. If there are dynamic dependencies, the computations are serialized.
The Spin Model Checker

- Spin is a model checker for communicating concurrent processes. It checks:
  - Safety/termination properties.
  - Liveness/deadlock properties.
  - Path assertions (requirements/never claims).
- It works on finite models, written in the Promela language, which describe infinite executions.
- Explores the entire state space of the model, including all possible concurrent executions, verifying that Bad Things don’t happen.
- Not an absolute proof — but pretty useful in practice.
Dekker’s mutex algorithm (C)

```c
int turn;
int wants[2];

// i is the current thread, j=1-i is the other thread
while(1) { // trying
    wants[i] = TRUE;
    while (wants[j]) {
        if (turn==j) {
            wants[i] = FALSE;
            while (turn==j) ; // empty loop
            wants[i] = TRUE;
        }
    }
    critical_section();
    turn=j; // release
    wants[i] = FALSE;
    noncrit();
}
```
Dekker’s “railroad”

Railroad visualization of Dekker’s algorithm for mutual exclusion. The threads “move” in the direction shown by the arrows.
Dekker’s mutex algorithm (Promela)

bool turn, flag[2]; byte cnt;
active [2] proctype mutex() /* Dekker’s 1965 algorithm */
{
    pid i, j;
    i = _pid;
    j = 1 - _pid;
again:
    flag[i] = true;
    do /* can be ‘if’ – says Doran&Thomas */
        :: flag[j] ->
            if
                :: turn == j ->
                    flag[i] = false;
                    !(turn == j);
                    flag[i] = true
                :: else
                    fi
            :: else -> break
        od;
    cnt++; assert(cnt == 1); cnt--; /* critical section */
    turn = j;
    flag[i] = false;
    goto again
}
Spin verification

$ spin -a mutex.pml
$ cc -DSAFETY -o pan pan.c
$ ./pan
(Spin Version 4.1.0 -- 6 December 2003)
    + Partial Order Reduction

Full statespace search for:

never claim    - (none specified)
assertion violations +
cycle checks    - (disabled by -DSAFETY)
invalid end states +

State-vector 20 byte, depth reached 65, errors: 0
    190 states, stored
    173 states, matched
    363 transitions (= stored+matched)
    0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
$
If an error is found, will give you execution trail producing the error.
Spin theory

• Generates a Büchi Automaton from the Promela specification.
  • Finite-state machine w/ special acceptance conditions.
  • Transitions correspond to executability of statements.
• Depth-first search of state space, with each state stored in a hashtable to detect cycles and prevent duplication of work.
  • If $x$ followed by $y$ leads to the same state as $y$ followed by $x$, will not re-traverse the succeeding steps.
• If memory is not sufficient to hold all states, may ignore hashtable collisions: requires one bit per entry. # collisions provides approximate coverage metric.
Modeling software transactions
Non-transactional Read

inline readNT(o, f, v) {
    do
        :: v = object[o].field[f];
        if
            :: (v!=FLAG) -> break /* done! */
        :: else
            fi;
    copyBackField(o, f, kill_writers, _st);
    if
        :: (_st==false_flag) ->
            v = FLAG;
            break
        :: else
            fi
    od
}
Non-transactional Write

```c
inline writeNT(o, f, nval) {
    if
    :: (nval != FLAG) ->
        do
            :: atomic {
                if /* this is a LL(readerList)/SC(field) */
                :: (object[o].readerList == NIL) ->
                    object[o].fieldLock[f] = _thread_id;
                    object[o].field[f] = nval;
                    break /* success! */
                :: else /* success! */
            fi
        }
    /* unsuccessful SC */
    copyBackField(o, f, kill_all, _st)
    od
    :: else -> /* create false flag */
        /* implement this as a short *transactional* write. */
        /* start a new transaction, write FLAG, commit the transaction, */
        /* repeat until successful. Implementation elided. */
        fi;
}
```
inline copyBackField(o, f, mode, st) {
    _nonceV=NIL; _ver = NIL; _r = NIL; st = success;
    /* try to abort each version. when abort fails, we’ve got a
    * committed version. */
    do
    :: _ver = object[o].version;
        if
        :: (_ver==NIL) ->
                  st = saw_race; break /* someone’s done the copyback for us */
        :: else
        fi;
    /* move owner to local var to avoid races (owner set to NIL behind
    * our back) */
    _tmp_tid=version[_ver].owner;
    tryToAbort(_tmp_tid);
        if
        :: (_tmp_tid==NIL || transid[_tmp_tid].status==committed) ->
                  break /* found a committed version */
        :: else
        fi;
    /* link out an aborted version */
    assert(transid[_tmp_tid].status==aborted);
    CAS_Version(object[o].version, _ver, version[_ver].next, _);
    od;
    continued.
/* okay, link in our nonce. this will prevent others from doing the copyback. */

if (st==success) ->
    assert (_ver!=NIL);
    allocVersion(_retval, _nonceV, aborted_tid, _ver);
    CAS_Version(object[o].version, _ver, _nonceV, _cas_stat);
if :: (!_cas_stat) ->
    st = saw_race_cleanup
:: else
fi
:: else
fi;

continued...
Copy-back Field, part III

/* check that no one's beaten us to the copy back */
if :: (st==success) ->
  if :: (object[0].field[f]==FLAG) ->
    _val = version[_ver].field[f];
    if :: (_val==FLAG) -> /* false flag... */
      st = false_flag /* ...no copy back needed */
    :: else -> /* not a false flag */
      d_step { /* LL/SC */
        if :: (object[0].version == _nonceV) ->
          object[0].fieldLock[f] = _thread_id;
          object[0].field[f] = _val;
          :: else /* hmm, fail. Must retry. */
            st = saw_race_cleanup /* need to clean up nonce */
        fi
      }
    fi
  :: else /* may arrive here because of readT, which doesn't set _val=FLAG*/
    st = saw_race_cleanup /* need to clean up nonce */
fi
:: else /* !success */
fi;

continued...
/* always kill readers, whether successful or not. This ensures that we
* make progress if called from writeNT after a readNT sets readerList
* non-null without changing FLAG to _val (see immediately above; st will
* equal saw_race_cleanup in this scenario). */

if
:: (mode == kill_all) ->
do /* kill all readers */
:: moveReaderList(_r, object[o].readerList);
  if
  :: (_r==NIL) -> break
  :: else
    fi;
  tryToAbort(readerlist[_r].transid);
  /* link out this reader */
  CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _);
  od;
:: else /* no more killing needed. */
  fi;
/* done */


Synchronization Failures

class A { // OK!
    int x; // shared variable
    synchronized int inc() {
        return x++;
    }
}

class B { // Race-free, but not OK.
    int x; // shared variable
    synchronized int get() { return x; }
    synchronized void set(int y) { x=y; }
    int inc() { // not monitored
        int t = get();
        t++;
        set(t);
        return t;
    }
}