Deconstructing Transactions: The Subtleties of Atomicity

Colin Blundell, E. Christopher Lewis, Milo M. K. Martin
University of Pennsylvania
(blundell, lewis, milom}@cis.upenn.edu

Converting a Lock-Based Queue

enqueue(Q, v){
Node_t node = malloc(...);
node->val = v;
node->next = NULL;
begin_transaction();
if (Q->tail)
Q->tail->next = node;
else
Q->head = node;
Q->tail = node;
end_transaction();
}

decqueue(Q){
int ret = -1;
begin_transaction();
if (Q->Head)
Node_t temp = Q->Head;
ret = temp->val;
Q->Head = temp->next;
end_transaction();
return ret;
}

Translation enhances concurrency, retains safety

Another Example

flagA, flagB = false;
Thread1
acquire(m->lock);
flagA = true;
while (!flagB){}
//update m
release(m->lock);
}

Thread2
acquire(n->lock);
while (!flagA){}
flagB = true;
//update n
release(n->lock);

Key: m->lock ≠ n->lock
Program Execution

flagA, flagB = false;
Thread1
acquire(m->lock);
  while (!flagA) {}
  spin
while (!flagB) {}
  //update m
  release(m->lock);

flagA = true;
Thread2
acquire(n->lock);
  while (!flagA) {}
begin_transaction();
flagB = true;
Thread2
acquire(n->lock);
while (!flagB) {}
end_transaction();

//update m

Program Execution: Thread 2 First

flagA, flagB = false;
Thread1
begin_transaction();
flagA = true;
Thread1
while (!flagB) {}
//update m
end_transaction();

Thread1 must commit first

Program Execution: Thread 1 First

flagA, flagB = false;
Thread1
begin_transaction();
flagA = true;
Thread1
while (!flagB) {}
//update m
end_transaction();

Thread1 must commit first

Thread2
begin_transaction();
flagB = true;
Thread2
while (!flagA) {}
end_transaction();

Thread2 must commit first
Cycle ➔ deadlock

Converting this Program to Transactions

flagA, flagB = false;
Thread1
begin_transaction();
flagA = true;
Thread1
while (!flagB) {} //update m
end_transaction();

Thread2
begin_transaction();
flagB = true;
Thread2
while (!flagA) {} //update n
end_transaction();

Transactions must appear atomic to each other...
So one must commit first
**Summary**

flagA, flagB = false;
Thread1
acquire(m-lock);
flag = true;
while (!flag) {} //update m
release(m-lock);

flagA, flagB = false;
Thread2
acquire(n-lock);
flag = true;
while (!flag) {} //update n
release(n-lock);

Lock→transaction conversion not always safe
Promise: benign data races not important

**Agenda**

- Transactions Can Break Lock-Based Code
- Why Critical Sections Aren’t Transactions
- Transactions and Non-Transactional Code
- Takeaway Points

**Why Did This Example Go Wrong? (1)**

flagA, flagB = false;
Thread1
acquire(m-lock);
flag = true;
while (!flag) {} //update m
release(m-lock);

Critical sections: atomic wrt others guarded by same lock

**Why Did This Example Go Wrong? (2)**

flagA, flagB = false;
Thread1
begin_transaction();
flag = true;
while (!flag) {} //update m
end_transaction();

Critical sections: atomic wrt others guarded by same lock
Transactions: atomic wrt all other transactions
Strengthening atomicity can disallow necessary interleavings (1)
A More Realistic Example

Thread1
queueA->enqueue(val1);
while (queueB->empty()){}
//access queueB

Thread2
queueB->enqueue(val2);
while (queueA->empty()){}
//access queueA

Thread1 and Thread2 mutual producer/consumers

A More Realistic Example

Queue* queueA = new Queue();
Queue* queueB = new Queue();

Thread1
acquire(m->lock);
begin_transaction();
if (x != 0)
z = y/x;
end_transaction();
x = 0;
release(m->lock);

Thread2
acquire(n->lock);
begin_transaction();
if (x != 0)
z = y/x;
end_transaction();
release(n->lock);

Convert to transactions: program breaks
As promised, no data races

Transactions ⊇ Critical Sections

Assumption
Reality

Implications
- Lock→transaction conversion not always safe
- Intuition about locks may be invalid for transactions

Impact on previous proposals
- No proposed conversion safe
- Exception: TLR OK in current form

Transactions and Non-Transactional Code

Thread1
begin_transaction();
x = 0;
if (x != 0)
z = y/x;
end_transaction();
Transactions and Non-Transactional Code

```
Thread1               Thread2
begin_transaction();  
if (x != 0)           
  z = y/x;           
end_transaction();   

x = 0;               
```

Q: Is this interleaving legal?
A: “We leave undefined the issue of interaction between 
transactions and non-transactional code.” [Herlihy & Moss, ICSA '93]

Previous Proposals’ Choices

Hardware proposals: mostly strong atomicity
Software proposals: mostly weak atomicity
Reason: varying cost of strong atomicity
  • HW: cheap if leveraging cache coherence protocol
  • SW: potentially expensive
What about semantic implications?
  • Does strong atomicity subsume weak?

Defining Terms

- **Weak atomicity**: transactions are not atomic with respect to non-transactional accesses
- **Strong atomicity**: transactions are atomic with respect to non-transactional accesses

Our Final Example

```
begin_transaction();
flagA = true;
while (!flagB){}
  ...
end_transaction();
flagA, flagB = false;
Thread1
Thread2
begin_transaction();
flagA = true;
while (!flagB){}
  ...
end_transaction();
```
Execution with Weak Atomicity

Program executes correctly

Execution with Strong Atomicity

Thread1: loops until Thread2 commits
Thread2: can't commit until Thread1 breaks out of its loop
Cycle → deadlock

Strong Atomicity ⊄ Weak Atomicity

Assumption

Reality

Strong and weak atomicity are incomparable
• System's interface should specify policy
• Differing policies harm portability
• Strengthening atomicity can disallow necessary interleavings (2)

Conclusions

1. Surprise: Transactions do not subsume critical sections
   • Locks: direct conversion to transactions not safe
   • Transactions: lock-based intuition may be invalid

2. Need discussion on policies re: transactions & N-T code
   • Define terms: strong & weak atomicity (or others...)

3. Surprise: Strong atomicity does not subsume weak
   • Affects portability of transactional programs

Transparently strengthening atomicity can break correct programs