Language-level Transactions for Modular Reliable Systems

C. Scott Ananian    Martin Rinard
cananian@csail.mit.edu    rinard@csail.mit.edu

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

HPEC 2004
Outline

- Problems with traditional software development
  - lock ordering
  - proper atomicity
  - fault-tolerance
  - priority inversion
- Language-level Transactions
- How?
  - Software implementation
  - Hardware implementation
  - Both!
- Conclusions
Programming Reliable Systems (is hard)
Conventional Locking: Ordering

- When more than one object is involved in a critical region, **deadlocks may occur!**
  - Thread 1 grabs A then tries to grab B
  - Thread 2 grabs B then tries to grab A
  - No progress possible!
- **Solution:** all locks ordered
  - A before B
  - Thread 1 grabs A then B
  - Thread 2 grabs A then B
  - No deadlock
**Conventional Locking: Ordering**

- Maintaining lock order is a lot of work!
- Programmer must choose, document, and rigorously adhere to a global locking protocol for each object type
  - development overhead!
- All symmetric locked objects must include lock order field, which must be assigned uniquely
  - space overhead!
- Every multi-object lock operation must include proper conditionals
  - which lock do I take first? which do I take next?
  - execution-time overhead!
- *No exceptions!*
Multi-object atomic update

- Programmer's mental model of locks can be faulty
- **Monitor synchronization**: associates locks with objects
- Promises modularity: locking code stays with encapsulated object implementation
- Often breaks down for multiple-object scenarios
- **End result**: unreliable software, broken modularity
public final class StringBuffer ... {
    private char value[];
    private int count;
    ...
    public synchronized StringBuffer append(StringBuffer sb) {
        ...
        A: int len = sb.length();
        int newcount = count + len;
        if (newcount > value.length)
            expandCapacity(newcount);
        // next statement may use state len
        B: sb.getChars(0, len, value, count);
        count = newcount;
        return this;
    }

    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }
}
**Fault-tolerance**

- Locks are **irreversible**
- When a thread fails holding a lock, the system will crash
  - it's only a matter of time before someone else attempts to grab that lock
- What are the proper semantics for exceptions thrown within a critical region?
  - data structure consistency not guaranteed
- Asynchronous exceptions?
Priority Inversion

- Well-known problem with locks
- Described by Lampson/Redell in 1980 (Mesa)
- Mars Pathfinder in 1997, etc, etc, etc
- Low-priority task takes a lock needed by a high-priority task -> the **high priority task must wait**!
- Clumsy solution: the low priority task must become high priority
- What if the low priority task takes a long time?
Outline

• Problems with traditional software development
  – lock ordering
  – proper atomicity
  – fault-tolerance
  – priority inversion
• Language-level Transactions
• How?
  – Software implementation
  – Hardware implementation
  – Both!
• Conclusions
Programming Reliable Systems (is easy?)
Locks are the wrong model for expressing synchronization!

**Atomicity** is a more natural (and modular) way to specifying the system

Let's use transactions to implement atomic regions

What sort of transactions do we want?
Transactions (definition)

- A transaction is a sequence of loads and stores that either **commits** or **aborts**
- If a transaction commits, all the loads and stores 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
- Simplified version of traditional ACID database transactions (no durability, for example)
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 trans. suicide
Making StringBuffer atomic

```java
public final class StringBuffer ... {
    private char value[];
    private int count;
    ...

    public synchronized StringBuffer append(StringBuffer sb) {
        ...
        A: int len = sb.length();
        int newcount = count + len;
        if (newcount > value.length)
            expandCapacity(newcount);
        // next statement may use state len
        B: sb.getChars(0, len, value, count);
        count = newcount;
        return this;
    }

    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }
}
```
public final class StringBuffer ... {
    private char value[];
    private int count;
    ...
    public atomic StringBuffer append(StringBuffer sb) {
        ...
        A: int len = sb.length();
        int newcount = count + len;
        if (newcount > value.length)
            expandCapacity(newcount);
        // next statement may use state len
        B: sb.getChars(0, len, value, count);
        count = newcount;
        return this;
    }
    public atomic int length() { return count; }
    public atomic void getChars(...) { ... }
}
Solving the lock ordering problem

```c
void pushFlow(Vertex v1, Vertex v2, double flow) {
    v1.excess -= flow; /* Move excess flow from v1 */
    v2.excess += flow; /* ...to v2 */
}
```

- Simple network flow algorithm
- "Flow" moved from node to node in the graph
- Updates to two different objects
- Serial version above requires a complicated parallel version when using locks
Solving the lock ordering problem

```java
void pushFlow(Vertex v1, Vertex v2, double flow) {
    v1.excess -= flow; /* Move excess flow from v1 */
    v2.excess += flow; /* ...to v2 */
}

void pushFlow(Vertex v1, Vertex v2, double flow) {
    Object lock1, lock2;
    if (v1.id < v2.id) { /* avoid deadlock */
        lock1 = v1; lock2 = v2;
    } else {
        lock1 = v2; lock2 = v1;
    }
    synchronized (lock1) {
        synchronized (lock2) {
            v1.excess -= flow; /* Move excess flow from v1 */
            v2.excess += flow; /* ...to v2 */
        }
    }
}
```
Solving the lock ordering problem

```c
void pushFlow(Vertex v1, Vertex v2, double flow) {
    v1.excess -= flow; /* Move excess flow from v1 */
    v2.excess += flow; /* ...to v2 */
}
```

```c
void pushFlow(Vertex v1, Vertex v2, double flow) {
    atomic {
        v1.excess -= flow; /* Move excess flow from v1 */
        v2.excess += flow; /* ...to v2 */
    }
}
```

- Specifying desired atomicity property directly is much simpler for the programmer!
Addressing reliability, fault tolerance, and priority inversion

- A proper implementation of the transaction mechanism allows constant-time abort
  - Allows us to solve priority inversion by aborting the low-priority thread!
- Atomicity properties are modular – no global lock ordering required
- A reasonable semantics for exceptions: critical region aborted/undone. No dangling locks.
- Failure of one thread will not cause the system to fail!
Programming Reliable Systems (is hard)

- Problems with traditional software development
  - lock ordering
  - proper atomicity
  - fault-tolerance
  - priority inversion
- Language-level Transactions
- How?
  - Software implementation
  - Hardware implementation
  - Both!
- Conclusions
Software Transaction Implementation

● **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 FLAGged field is: “value of the first version owned by a committed transaction on the version list”
  - Values which are “really” FLAG are handled with an escape mechanism
Transactions using version lists

Object #1

- MyClass
  - type
  - versions
  - {OID68}
    - readers
  - FLAG
    - field1
  - 3.14159
    - field2
  - ...

Object #2

- OtherClass
  - type
  - versions
  - {OID25}
    - readers
  - 2.71828
    - field1
  - FLAG
    - field2
  - ...

Transaction ID #68

- Version
  - owner
  - next
  - 23
    - field1
  - FLAG
    - field2
  - ...

Transaction ID #56

- Version
  - owner
  - next
  - 55
    - field1
  - FLAG
    - field2
  - ...

Transaction ID #23

- Version
  - owner
  - next
  - 2.71828
    - field1
  - FLAG
    - field2
  - ...

Waiting status

COMMITTED status

COMMITTED status
Performance

- Non-transactional code only needs to check whether a memory operand is \texttt{FLAG} before continuing.
  - On superscalar processors, there are plenty of extra functional units to do the check
  - The branch is extremely predictable
  - This gives only a few \% slowdown

- Once \texttt{FLAGged}, transactional code operates directly on the object’s “version”

- Creating versions can be an issue for large arrays; use “functional array” techniques
Non-blocking algorithms are hard!

- In published work on Synthesis, a non-blocking operating system implementation, three separate races were found:
  - One **ABA problem** in LIFO stack
  - One **likely race** in MP-SC FIFO queue
  - One **interesting corner case** in quaject callback handling
- It's hard to get these right! Ad hoc reasoning doesn't cut it.
- Non-blocking algorithms are too hard for the programmer
- Let's get it right **once** (and verify this!)
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 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 – pretty useful in practice

• **Make systems reliable by concentrating complexity in a verifiable component**
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
Verified Software Transactions

- Modelled the software transaction implementation in Promela
- Low-level model – every memory operation represented
- Spin used 16G of memory to exhaustively verify the implementation within a 6-version 2-object scope.
**Hardware Implementation**

- Following earlier work by Knight '86, Herlihy and Moss '92, '93
- Cache is used to store uncommitted transactional state (marked with a T bit)
- Main memory contains 'backup state'
- Cache-coherence protocol extended to coordinate transactions
- Our recent work (Ananian, Asanović, Kuszmaul, Leiserson, Lie HPCA 2005) overcomes transaction-size limitations in earlier designs
- Near-zero performance overhead.
  - Piggy-backs on existing cache coherency traffic
Each cache line gets a “T” bit indicating that this line is involved in a transaction.

On abort, “T” lines are invalidated.

On commit, the T bits are cleared.

Overflow mechanism.
Register File Modifications

- Minor modifications to the processor rename table to support register restore after transaction abort.
Hardware/Software Implementation

- Hardware transaction implementation is very fast! But it is limited:
  - Slow once you exceed Cache capacity
  - Transaction lifetime limits (context switches)
  - Limited semantic flexibility (nesting, etc)
- Software transaction implementation is unlimited and very flexible!
  - But transactions may be slow
- **Solution:** failover from hardware to software
  - Simplest mechanism: after first hardware abort, execute transaction in software
  - Need to ensure that the two algorithms play nicely with each other (consistent views)
Overcoming HW size limitations

- Simple node-push benchmark
- As xaction size increases, we eventually run out of cache space in the HW transaction scheme
Overcoming HW size limitations

- Simple node-push benchmark
- Hybrid scheme best of both worlds!
Conclusions

- Language-level transactions provide a more-modular way to build reliable concurrent systems.
- Transactions can reduce software complexity and eliminate common programmer mistakes.
- We've implemented a transaction mechanism for Java programs using software, hardware, and (in progress) joint approaches using the FLEX compiler infrastructure.
- Transactions can be efficient and practical to use!