Noninvasive concurrency with Java STM

(Guy Korland, Nir Shavit, and Pascal Felber, 2010)

Patrik Persson, Dec. 5, 2013

Previously on *Software Transactional Memory*

- STM is about *opportunistic* concurrency control:
  try to commit, detect conflicts, retry transaction if needed

- Harris & Fraser, 2003: proposed *atomic* construct for Java

- 2013: STM is no longer science fiction

  - STM support for C++ in gcc 4.7
    (seems primitive though)

  - Several approaches for Java being investigated, aiming for modified VM, compiler, or dedicated frameworks

Korland/Shavit/Felber: Java with annotations

- Annotations checked at load time
- On-the-fly modification of class files when loaded
- Create instrumented (transaction-aware) versions of classes
- Per-method only

```
@Atomic(retries=64)
public boolean contains(int v) {
  Node node = head;
  for (int i = level; i >= 0; i--) {
    Node next = node.forward[i];
    while (next.value < v) {
      node = next;
      next = node.forward[i];
    }
  }
  node = node.forward[0];
  return (node.value == v);
 }
 // ...
}
```

On-the-fly modifications to loaded classes

- Getters & setters introduced
- Duplicate methods (transaction-aware, when called from @Atomic methods)

```
public class SkipList {
  private int level;
  // ...

  // Synthetic getter
  public int level__Getter$(Context c) {
    c.beforeReadAccess(this, level__ADDRESS__);
    return c.onReadAccess(this, level, level__ADD
  }
  // Synthetic setter
  public void level__Setter$(int v, Context c) {
    c.onWriteAccess(this, v, level__ADDRESS__);
  }
  // ...
}
```

On-the-fly modifications, cont'd

```
1  public class SkipList {
2    // ...
3
4    // Original method instrumented
5    public boolean contains(int v) {
6      Throwable throwable = null;
7      Context context =
8        ContextDelegator.getInstance();
9      boolean commit = true;
10     boolean result;
11
12     for (int i = 64; i > 0; --i) {
13       context.init();
14       try {
15         result = contains(v, context);
16       } catch(TransactionException ex) {
17         // Must rollback
18         commit = false;
19       } catch(Throwable ex) {
20         throwable = ex;
21       }
22       // Continued in next column...
```

```
23       // Try to commit
24       if (commit) {
25         if (context.commit()) {
26           if (throwable == null)
27             return result;
28           // Rethrow application exception
29           throw (IOException)throwable;
30         }
31       } else {
32         context.rollback();
33         commit = true;
34       }
35     } // Retry loop
36     throw new TransactionException();
37   }
38
39   // Synthetic duplicate method
40   public boolean contains(int v, Context c) {
41     Node node = head__Getter$(c);
42     // ...
43   }
44 }
```

Adding support for atomic blocks

```
public int transferAll(Account[] src, Account dst) {
  int total = 0;
  for (Account acc : src) {
    atomic {
      int amount = acc.balance();
      acc.withdraw(amount);
      dst.deposit(amount);
      total += amount;
    }
  }
  return total;
}
```

In summary

- A (somewhat) realistic system for STM in Java
  - Implementation flaky?
- Based on annotations & on-the-fly instrumentation of classes during loading
- Annotations are per-method;
  atomic blocks supported using separate, JastAdd-based source-to-source translation tool
- You *could* actually use this for concurrent programs…

@Atomic exercise

Example: AtomicAccount
One set of threads deposits, another set withdraws (the same amounts)

If everything works, final balance is 0

Your task

1. Run it a few times without synchronization. It hopefully doesn't work.

2. Make it work using *synchronized*.

3. Make it work using *@Atomic* instead – **not** *synchronized*.

4. Measure performance of *synchronized* vs. *@Atomic*. Experiment with the number of threads. Be prepared to force-terminate your program.

```
public class AtomicAccount {
  private static long balance = 0;

  public static void deposit(long n) {
    balance += n;
  }

  public static long getBalance() {
    return balance;
  }

  …
}
```

The code, with instructions for building and running, is available at http://fileadmin.cs.lth.se/cs/Education/EDA015F/2013/AtomicAccount.java