Transactional Memory

Liangzhen Lai

Outline

- Transactional Memory vs. Lock
- Data Versioning
- Conflict Detection
- Hardware TM vs. Software TM

Memory Interleaving

```
Thread I
A = counter //read
(…)
A++
counter = A //write
```

```
Thread II
A = counter //read
(…)
A++
counter = A //write
```

R1->W1->R2->W2: counter +=2
R1->R2->W2->W1: counter +=1
R1->R2->W1->W2: counter +=1

Transaction vs. Lock

```
Transaction
Atomic{
  A = counter //read
  (…)
  A++
  counter = A //write
}
```

```
Lock
Lock(counter)
A = counter //read
(…)
A++
counter = A //write
Unlock(counter)
```

- Transaction guarantees atomicity
- Programmers worry about program atomicity and transaction boundary
- System designers worry about implementation
- Transaction abort makes exception handler easier

- Lock guarantees variable ownership
- Programmers worry about lock locations to guarantee correctness
- System designers are happy~
- Lock blocks other thread to read the variable
- Read/Write lock is even tougher to use

Transactional Memory

- Execute each transaction atomically
  - All or nothing
  - No interference from other threads
- Data versioning
  - Store both old and new version
- Conflict detection
  - Detect memory interleavings that violate the atomicity

Data Versioning

- Eager Versioning
  - Update inplace for each memory write
  - Store old values somewhere
  - Proceed upon commit or restore upon abort
- Lazy Versioning
  - Maintain a write buffer to memory write
  - Write the values into memory upon commit or clear upon abort

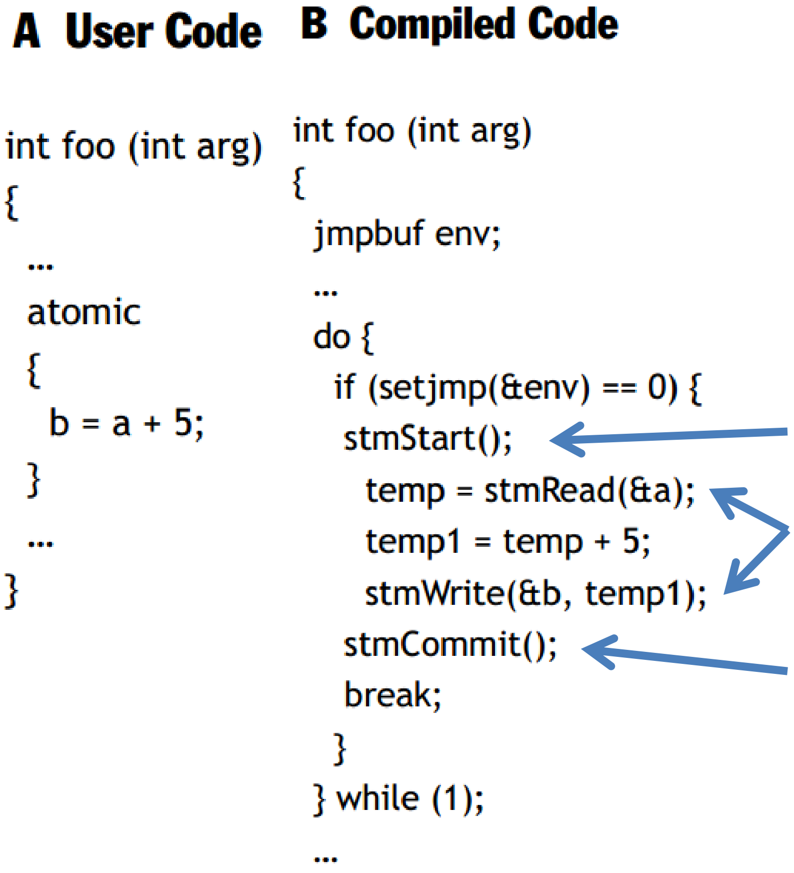
Conflict Detection

- Condition: write-set of one thread overlaps with either read-set or write-set of another thread
- Stall (Eager Detection)
  - – Avoid giving up already finished work
  - – Can result in deadlock
- Abort (Lazy Detection)
  - – Can result in livelock

Thread I	Thread II
Write A	Write B
(…)	(…)
Read B	Read A

Software Transactional Memory

- Implemented entirely on software



A User Code

```
int foo (int arg)
{
  …
  atomic
  {
    b = a + 5;
  }
  …
}
```

B Compiled Code

```
int foo (int arg)
{
  jmpbuf env;
  …
  do {
    if (setjmp(&env) == 0) {
     stmStart();
       temp = stmRead(&a);
       temp1 = temp + 5;
       stmWrite(&b, temp1);
     stmCommit();
     break;
    }
  } while (1);
  …
```

Heavily relied on compiler optimization of the instrumentation

Hard to guarantee isolation of transactional and nontransactional code

Data versioning

Data access barrier

Transaction completes and results are visible to other threads

Hardware Transactional Memory

- Data Versioning
  - Use cache hierarchy
  - Hardware write-buffer/Software thread log
  - Be aware of cache overflow!
- Conflict Detection
  - Use cache coherence protocol
  - Associate W/R bit for each cache line
  - Be aware of cache overflow!
- Contention Management
  - Random back-off (avoid live lock)
  - Priority-forced abort (avoid dead lock)