

# Concepts of Concurrent Computation

## Seminar Talk

Karolos Antoniadis

April 17, 2013

# Applying Transactional Memory to Concurrency Bugs

Haris Volos, Andres Jaan Tack,  
Michael M. Swift, Shan Lu

ASPLOS' 12

# What is Transactional Memory (TM)?

Paradigm for concurrent programming that eliminates the usual problems when programming with locks.

A **transaction** is a sequence of steps executed by a single processor.

Transactions are:

- ▶ **Atomic:** each transaction either commits (it takes effect) or aborts (its effects are discarded)
- ▶ **Isolated:** intermediate changes of one transaction are not visible to other transactions

There are hardware, software and hybrids TMs.

# Motivation

Check utility of TM as a tool for fixing concurrency bugs.

Find mechanisms that could be useful if added in TM.

# Bug-Fixing Methods

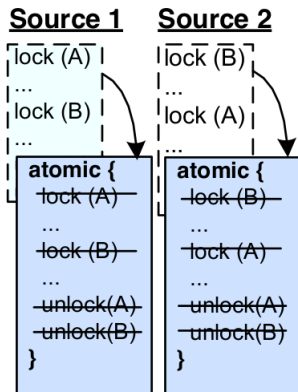
- ▶ **Ingredients:** underlying mechanisms provided by TM.
- ▶ **Recipes:** methods on how to combine the mechanisms to fix bugs.

# Ingredients

- ▶ **Atomic regions:** ensure that transactions execute atomically and in isolation.
- ▶ **Explicit rollback (retry):** transactions can rollback partially executed transactions.  
Can be used to wait until some condition is true, similar to condition variables.
- ▶ **Preemptible resources:** can safely be acquired inside a transaction and automatically released if the transaction aborts.
- ▶ **Atomic/lock serialization:** synchronizes accesses protected under a transaction with accesses protected under a lock.

# Replacement of Deadlock-Prone Locks

Remove all locks contributing to a deadlock and insert transactions in their places.



If deadlock is about to occur, it is prevented by:

- ▶ serializing the threads involved
- ▶ automatically aborting one or more transactions

# (Asymmetric) Deadlock Preemption

Make at least one thread involved in the deadlock preemptible.

## Source 1

```
lock (A)  
...  
lock (B)  
...  
atomic {  
  lock (A)  
  ...  
  lock (B)  
}  
...  
unlock(A)  
unlock(B)
```

## Source 2

```
lock (B)  
...  
lock (A)  
...  
unlock(A)  
unlock(B)
```

Removes non-preemption requirement for a deadlock.

Code wrapped in a transaction better if used infrequently.

Requires:

- ▶ preemptible resources
- ▶ deadlock detector



# Wrap All

Wrap all conflicting code regions into transactions.

## Source 1

```
read x  
write x
```

```
atomic {  
  read x  
  write x  
}
```

## Source 2

```
lock (A)  
read x  
write x  
unlock(A)
```

```
atomic {  
  lock (A)  
  read x  
  write x  
  unlock (A)  
}
```

Ensures that memory accesses are protected under the same synchronization mechanism.

# Wrap Unprotected

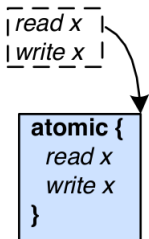
Wrap the code region intended to be executed atomically in a transaction.

## Source 1

```
| read x |  
| write x |
```

## Source 2

```
lock (A)  
read x  
write x  
unlock(A)
```



```
atomic {  
  read x  
  write x  
}
```

“Wrap All” recipe duplicates any existing effort put into using locks.

Requires:

- ▶ atomic/lock serialization

# Recipes Comparison

	Atomic Regions	Explicit Rollback	Preemptible Resources	Atomic/Lock Serialization
Replacement of Deadlock-Prone Locks	YES	Not needed by a specific recipe	NO	NO
Deadlock Preemption			YES + deadlock detector	NO
Wrap All			YES(?)	NO
Wrap Unprotected			YES	YES

“Deadlock Preemption” and “Wrap Unprotected” easy to apply since existing locks are not removed.

Less code changes (only one transaction is added).

Better performance, less overhead by the TM.

Although they need more ingredients.

# Apache: Deadlock Bug

(solved using deadlock preemption)

```
1 worker_thread(...)
2 {
3   ...
4   LOCK (timeout);
5   ...
6   UNLOCK (timeout);
7   ...
8   LOCK (idlers)
9   ...
10  SIGNAL (wait_for_idlers)
11  ...
12  UNLOCK (idlers)
13 }
```

(a) Buggy code

```
1 listener_thread (...)
2 {
3   ...
4   LOCK (timeout);
5   ...
6   LOCK (idlers);
7   ...
8   COND_WAIT (wait_for_idler,
9             idlers)
10  UNLOCK (idlers)
11  ...
12  UNLOCK (timeout)
13 }
```

```
1 listener_thread (...)
2 {
3   ...
4   atomic {
5     LOCK (timeout);
6     ...
7     LOCK (idlers);
8     ...
9     if (!COND_TRY_WAIT(...))
10      retry;
11    UNLOCK (idlers)
12  }
13  ...
14  UNLOCK (timeout)
15 }
```

(b) Fixed with TM

# Apache: Deadlock Bug

**Developers fix:** Unlock the timeout lock before going through the conditional wait. Took them 3 failed attempts.

TM fix is 22% slower than the developers, but is preferable since:

- ▶ simple, no reasoning
- ▶ performance under stress test workload
- ▶ retry was done with spinning

# Apache: Atomicity Violation Bug

(solved using wrap unprotected)

```
1 void ap_buffered_log_writer (...)  
2 {  
3   ...  
4   s = &buffer[buf->outputCount];  
5   memcpy (s, str, len);  
6   temp = buf->outputCount + len;  
7   buf->outputCount = temp;  
8   apr_file_write(buf->handle);  
9   ...  
10 }
```

(a) Buggy code

```
1 void ap_buffered_log_writer (...)  
2 {  
3   ...  
4   atomic {  
5     s = &buffer[buf->outputCount];  
6     memcpy (s, str, len);  
7     temp = buf->outputCount + len;  
8     buf->outputCount = temp;  
9     apr_file_write(buf->handle);  
10  }  
11  ...  
12 }
```

(b) Fixed with TM

I/O is done using xCalls: library-based implementation of transactional semantics for common system calls.

# Apache: Atomicity Violation Bug

**Developers fix:** assign a lock to each log device, more scalable than using a single big lock.

TM fix is 4% slower and is preferable because:

- ▶ it is local (only changes inside one function)
- ▶ has comparable performance with the developers fix

# Evaluation

Tried to use TM to fix 60 bugs in already found and fixed bugs in Mozilla, Apache HTTP Server and MySQL.

- ▶ Fixed 43 out of the 60 bugs.
- ▶ Out of the 43 TM fixes, 34 of them are preferable to the developers fixes.



# Why can't TM fix every concurrent bug?

- ▶ Locks contained in multiple modules.
- ▶ Code that needs to be changed is contained in third-party plugins.
- ▶ Design errors.
- ▶ Long-running operations.
- ▶ Two-way communication.

# Conclusion

- ▶ Straightforward use of TM has the expressive power to fix many concurrency bugs.
- ▶ But still work to be done concerning TM performance.
- ▶ TM research could be directed to tackle:
  - ▶ two-way communication
  - ▶ long-running operations

# Questions?