Transactional Memory

Zichao Yang and Qing Zheng

Applying Transactional Memory to Concurrency Bugs

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

ASPLOS 12

Transaction

A block of code declared to be an atomic region, executed by a single processor, and isolated from other regions

Atomicity → execute to completion/not at all

Consistency → correctness (programmer’s job)

Isolation → side-effects remain invisible until completion

Durability → not needed

Transactional Memory

An underlying memory system capable of execution transactions

- H/W TM → speculative execution (with limitations)
- S/W TM → code instrumentation (performance issues)
- Hybrid TM → use H/W, fall back to S/W if necessary

Bug-Fix Techniques

Atomic regions → execute atomically with isolation
(speculative execution, locks)

Explicit rollback → abort a partially executed transaction
(opportunities for retries, mimic conditional variables)

Preemptive resources → transaction-friendly
(revertible locks, I/O, system calls)

Atomic/lock serialization → a bridge between atomic region and traditional locks

Concurrency Bugs

Writing correctly synchronized code can be challenge!

Deadlock (DL) → use locks with a wrong order

Atomicity Violation (AL) → fail to protect critical sections

Resolving Deadlock Bugs

Recipe 1: Replacement of Deadlock-prone Locks

(1) remove the use of multiple locks

(2) automatically aborting conflicting threads

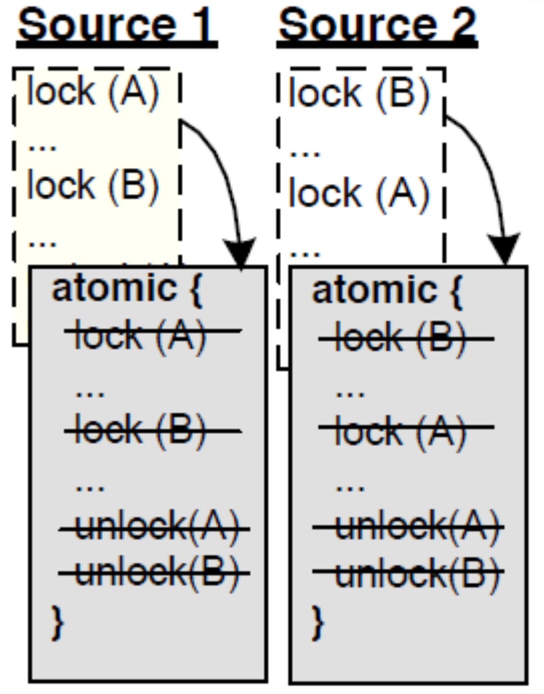
(3) preserves concurrency

Recipe 2: Asymmetric Deadlock Preemption

(1) retry lock acquisition if deadlock happens

(2) require preemptive resources and deadlock detector

(3) use TM only for atomicity, not isolation



Source 1

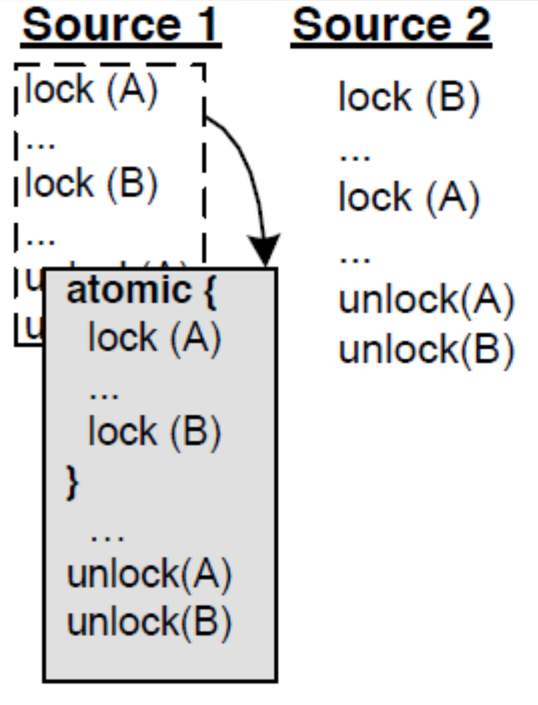
lock (A)
...
lock (B)
...

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

Source 2

lock (B)
...
lock (A)
...

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



Source 1

lock (A)
...
lock (B)
...
u
u

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

Source 2

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

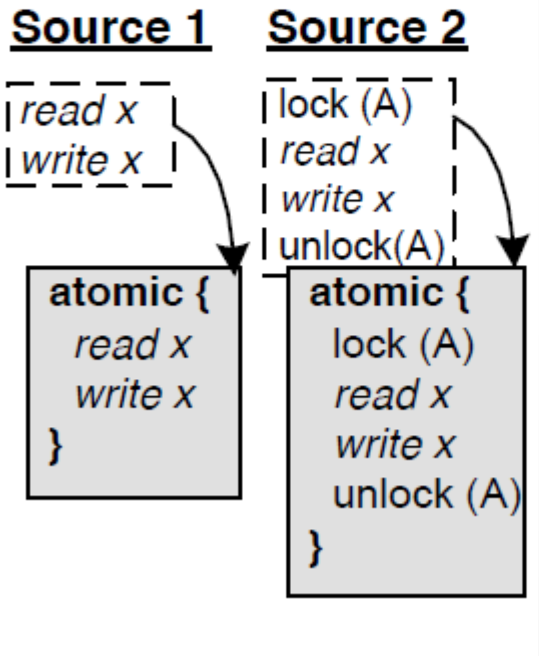
Addressing Atomic Violations

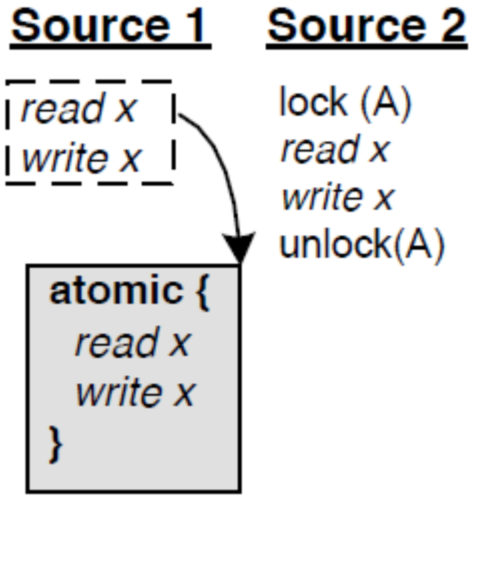
Recipe 1: Wrap All

(1) use TM to achieve isolation and mutual exclusion
(2) compatible with existing locks
(3) no need to introduce new locks

Recipe 2: Warp Unprotected

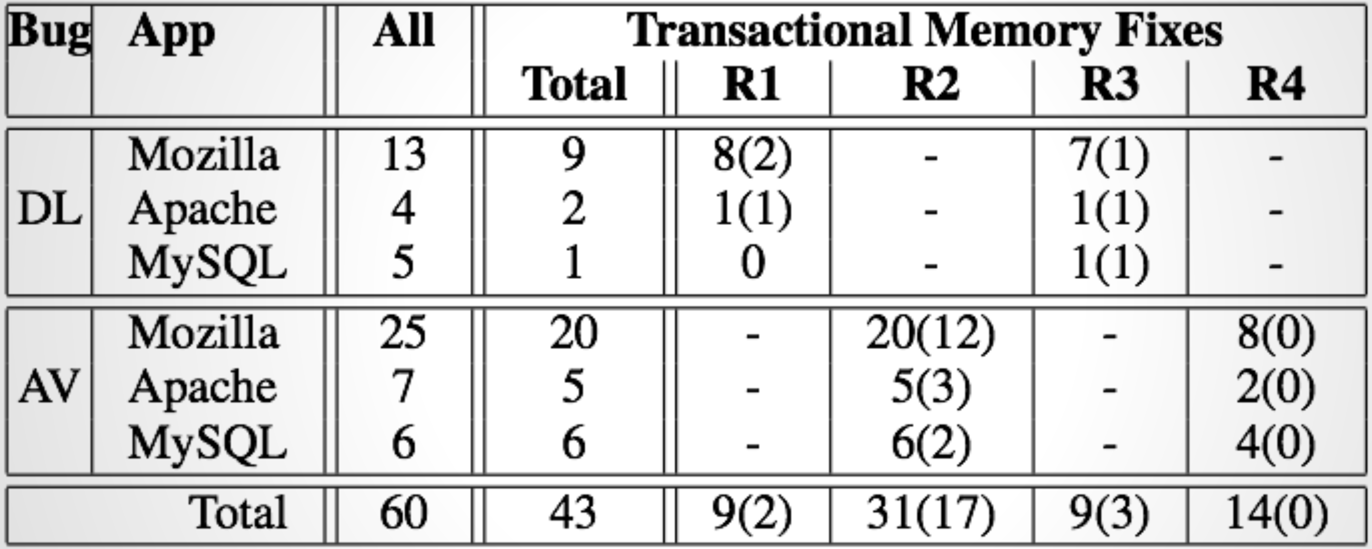
(1) reuse existing correct codes and keeps them unchanged
(2) require atomic/lock serialization
(3) may lead to performance issues





Evaluation & Case Study

Effectiveness

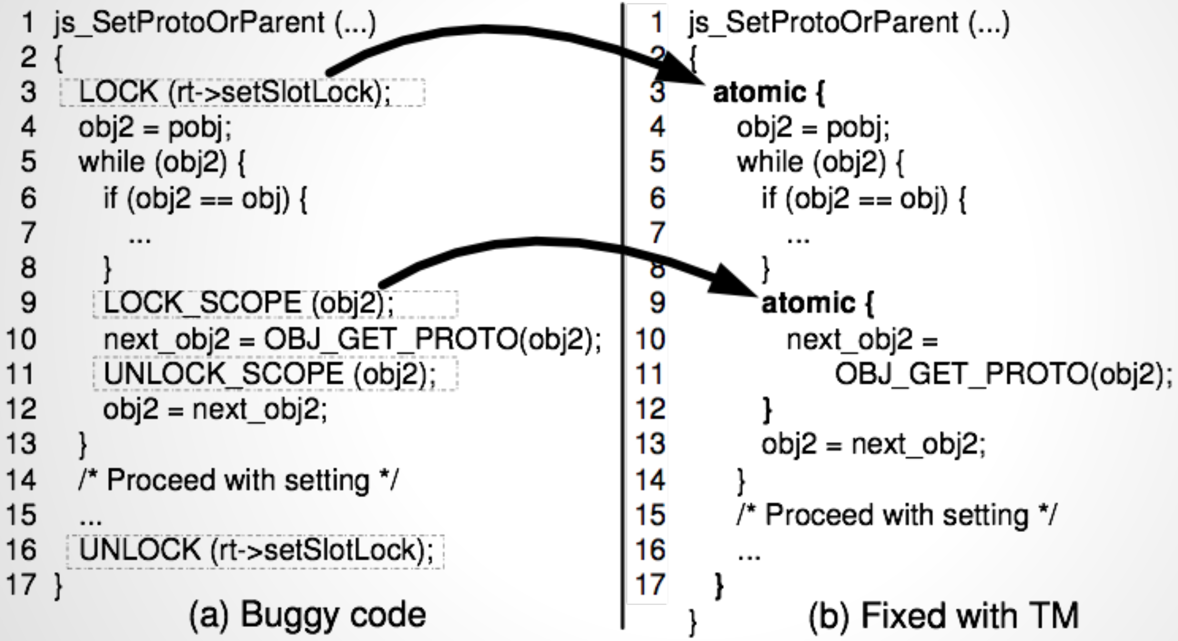


Bug	App	All	Transactional Memory Fixes				
			Total	R1	R2	R3	R4
DL	Mozilla	13	9	8(2)	-	7(1)	-
	Apache	4	2	1(1)	-	1(1)	-
	MySQL	5	1	0	-	1(1)	-
AV	Mozilla	25	20	-	20(12)	-	8(0)
	Apache	7	5	-	5(3)	-	2(0)
	MySQL	6	6	-	6(2)	-	4(0)
	Total	60	43	9(2)	31(17)	9(3)	14(0)

- fix 43 out of 60 bugs: DL 12/22 AV31/38
- R1 and R2 can fix 40 out of 43

Case study

- Mozilla-I: Deadlock



```
js_SetProtoOrParent (...)
{
  LOCK (rt->setSlotLock);
  obj2 = pobj;
  while (obj2) {
    if (obj2 == obj) {
      ...
    }
    LOCK_SCOPE (obj2);
    next_obj2 = OBJ_GET_PROTO(obj2);
    UNLOCK_SCOPE (obj2);
    obj2 = next_obj2;
  }
  /* Proceed with setting */
  ...
  UNLOCK (rt->setSlotLock);
}
```

(a) Buggy code

```
js_SetProtoOrParent (...)
{
 atomic {
   obj2 = pobj;
   while (obj2) {
     if (obj2 == obj) {
       ...
     }
     atomic {
       next_obj2 =
            OBJ_GET_PROTO(obj2);
     }
     obj2 = next_obj2;
   }
   /* Proceed with setting */
   ...
 }
}
```

(b) Fixed with TM

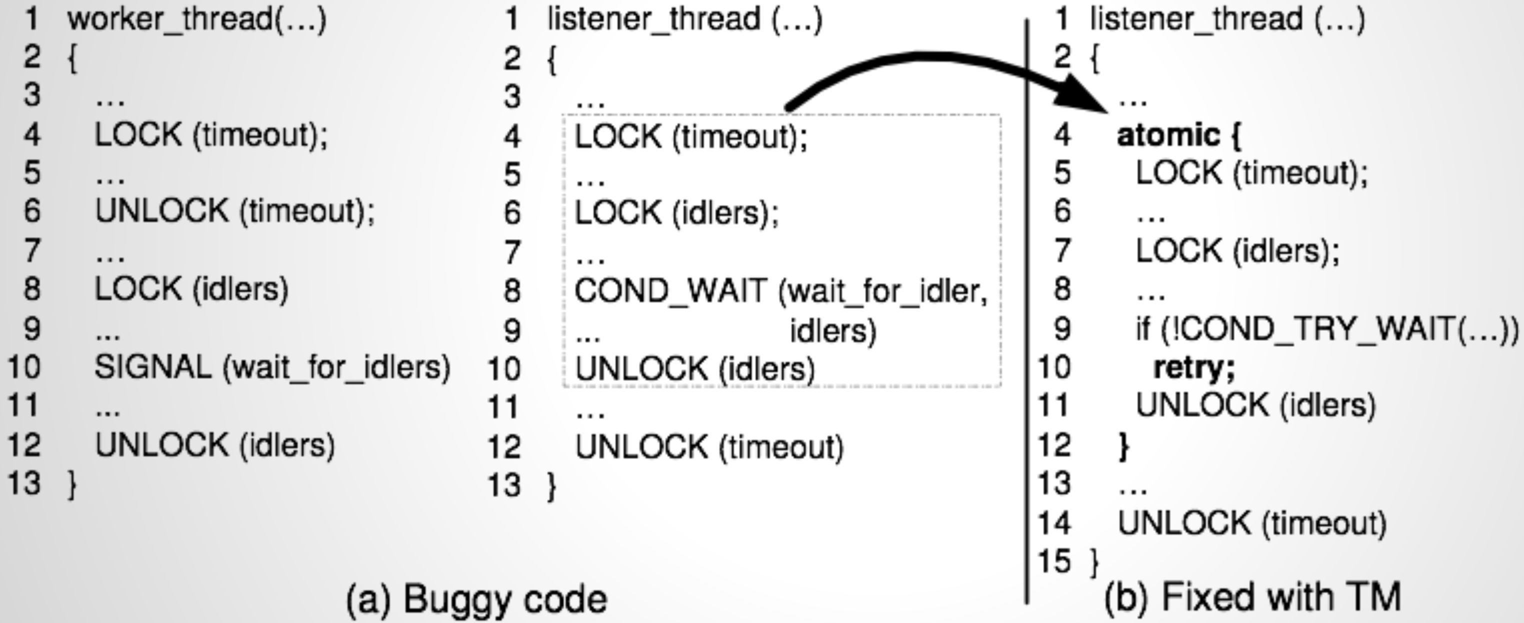
- Deadlock
    - two threads access the two locks in different order

Case study

- Mozilla-I: Deadlock
  - developer fix (hard)
    - force threads to drop ownership before blocking
    - new conditional variables
  - TM fixes
    - recipe 1: replace lock with atomic sections
    - recipe 3: revocable locks
  - Comparison
    - recipe1 solves fours other bugs (side effect)
    - performance: recipe 1: 79% worse

Case study

- Apache-I: deadlock



```
worker_thread(...)
{
   ...
   LOCK (timeout);
   ...
   UNLOCK (timeout);
   ...
   LOCK (idlers)
   ...
   SIGNAL (wait_for_idlers)
   ...
   UNLOCK (idlers)
}
```

```
listener_thread (...)
{
   ...
   LOCK (timeout);
   ...
   LOCK (idlers);
   ...
   COND_WAIT (wait_for_idler,
                        idlers)
   ...
   UNLOCK (idlers)
   ...
   UNLOCK (timeout)
}
```

(a) Buggy code

```
listener_thread (...)
{
   ...
   atomic {
     LOCK (timeout);
     ...
     LOCK (idlers);
     ...
     if (!COND_TRY_WAIT(...))
       retry;
     UNLOCK (idlers)
   }
   ...
   UNLOCK (timeout)
}
```

(b) Fixed with TM

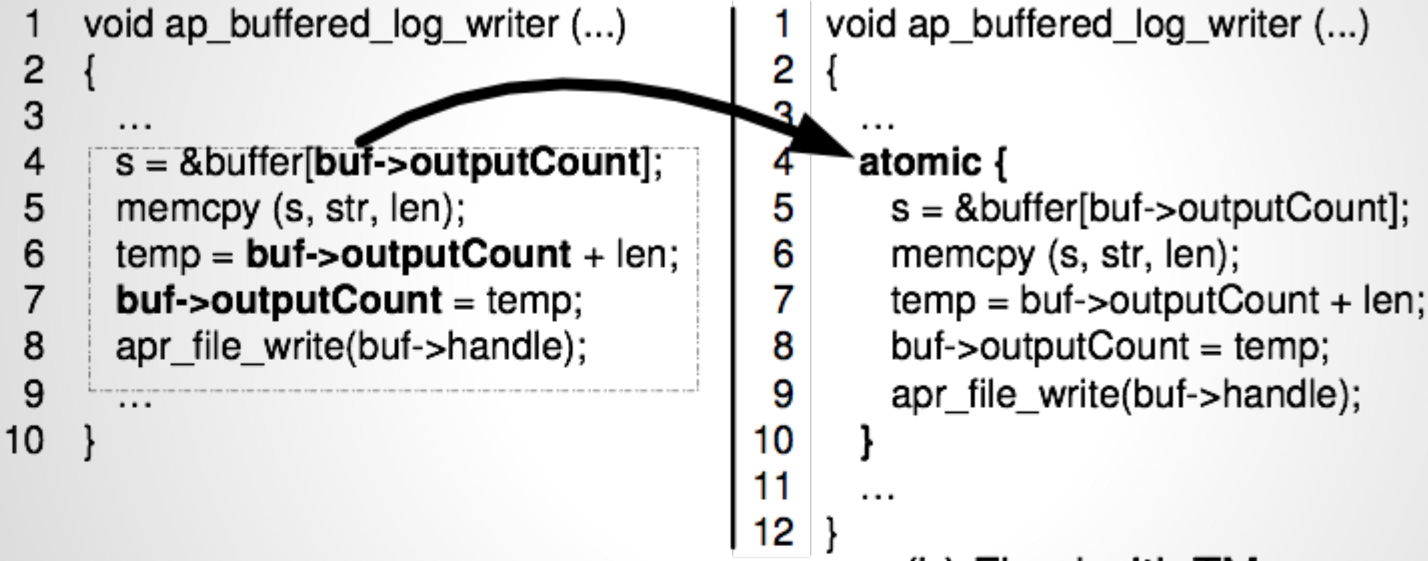
- Deadlock
  - listener first hold timeout then waiting for idle worker thread
  - worker thread first get timeout lock then signal

Case study

- Apache-I: deadlock
  - developer fix (hard)
    - release time out before wait
    - three failed attempts
  - TM fixes
    - recipe 3: abort transaction if wait
  - Comparison
    - simpler to fix
    - 28% worse in performance

Case study

- Apache-II: missing synchronization



```
void ap_buffered_log_writer (...)
{
  ...
  s = &buffer[buf->outputCount];
  memcpy (s, str, len);
  temp = buf->outputCount + len;
  buf->outputCount = temp;
  apr_file_write(buf->handle);
  ...
}
```

(a) Buggy code

```
void ap_buffered_log_writer (...)
{
  ...
  atomic {
    s = &buffer[buf->outputCount];
    memcpy (s, str, len);
    temp = buf->outputCount + len;
    buf->outputCount = temp;
    apr_file_write(buf->handle);
  }
  ...
}
```

(b) Fixed with TM

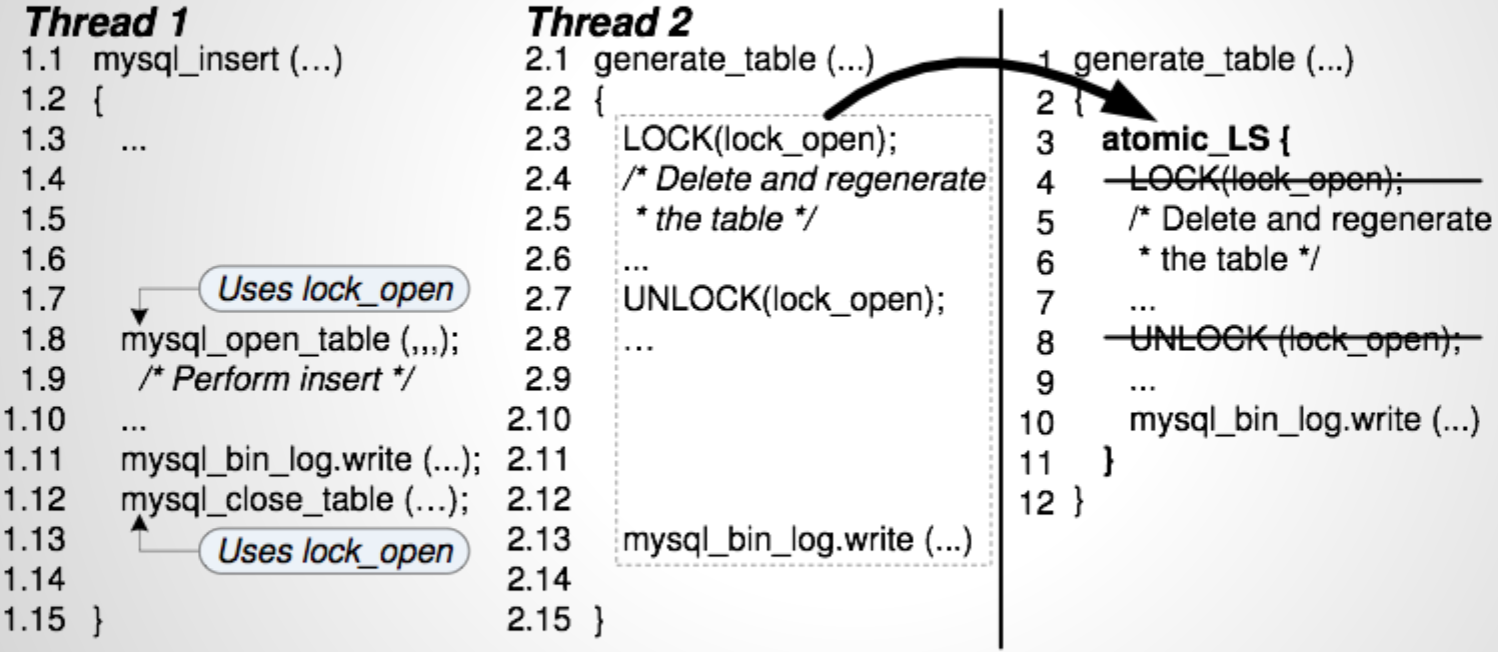
- race condition
    - two threads compete over the buffer

Case study

- Apache-II: missing synchronization
  - developer fix (medium difficulty)
    - use lock for each log device
  - TM fixes
    - insert single atomic block
    - easy
  - Comparison
    - TM simpler to fix
    - 4% slower

Case study

- MySQL-I: missing synchronization



```
Thread 1                                   Thread 2
1.1   mysql_insert (...)                   2.1   generate_table (...)
1.2   {                                    2.2   {
1.3     ...                                2.3     LOCK(lock_open);
1.4                                        2.4     /* Delete and regenerate
1.5                                        2.5      * the table */
1.6                                        2.6     ...
1.7                                        2.7     UNLOCK(lock_open);
1.8     mysql_open_table (,,,);            2.8     ...
1.9      /* Perform insert */              2.9
1.10    ...                                2.10
1.11    mysql_bin_log.write (...);         2.11
1.12    mysql_close_table (...);           2.12
1.13                                       2.13    mysql_bin_log.write (...)
1.14                                       2.14
1.15  }                                    2.15  }
```

```
1  generate_table (...)
2  {
3    atomic_LS {
4    ~~LOCK(lock_open);~~
5      /* Delete and regenerate
6       * the table */
7      ...
8    ~~UNLOCK (lock_open);~~
9      ...
10     mysql_bin_log.write (...)
11   }
12 }
```

(a) an incorrect interleaving

(b) Fixed with TM

- Deadlock
  - delete release lock too early

Case study

- Apache-II: missing synchronization
    - developer fix (hard)
        - extent lock_open to the end
        - performance implications
    - TM fixes
        - insert single atomic block
        - easy
    - Comparison
        - simple, expressive and non-invasive
        - same performance

Conclusion

- Very simple to use TM to fix bugs
- performance remains to be improved
- deal with cases can’t tackle now