



Transaction Memory for Existing Programs

Michael M. Swift

Haris Volos, Andres Tack, Shan Lu, Adam Welc *

University of Wisconsin-Madison, *Intel

Where do TM programs come from?

- Parallel benchmarks replacing all locks
 - Splash 2, Parsec, ...
- Write new programs from scratch
 - Stamp
 - STMBench 7
- What about existing multithreaded programs?

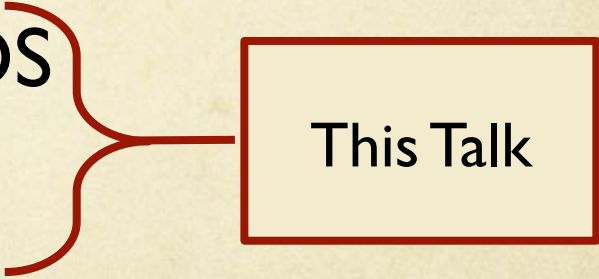
Considerations of Real Programs

- They run on virtual hardware
 - They context switch
 - They run in hypervisors
- They use the operating system
 - They use file, network I/O
- They are not 100% transactional
 - They use locks and condition variables
 - They have too much code to rewrite
 - The benefit from *targeted use* of TM

Our Work

- LogTM-SE/VSE/TokenTM: virtualizing HW transactional memory
- TxLocks and TxCondVars: interaction with locks and condition variables
- xCalls: interacting with the OS
- TM as a concurrency bug fix

Our Work

- LogTM-SW/VSE/TokenTM: Virtualizing HW transactional memory
 - TxLocks and TxCondVars: interaction with locks and condition variables
 - xCalls: interacting with the OS
 - TM as a concurrency bug fix
- 
- This Talk

Outline

- Introduction
- xCalls: transactional access to OS services
 - Design
 - Results
- TM as a concurrency bug fix
- Conclusions

A challenging world...

- Real world programs frequently take actions outside of their own memory
 - Firefox: ~1% critical sections call OS [Baugh TRANSACT '07]
- A single lock may normally protect memory, but *sometimes* protect OS state
- Most TM systems apply only to user-level memory

State of the art

○ Defer [TxOS]

○ Undo [LogTM]

○ Global Lock [IrrevTM, OneTM]

Ignore failures

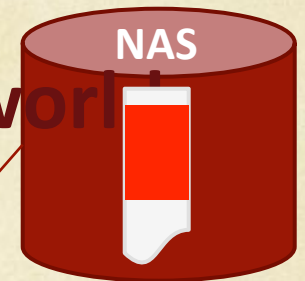
Stop the world

```
→ atomic {  
  item = procltem(queue);  
  write (file, principal);  
  write (file, item->header);  
  send (socket, item->body);  
}
```

COMMIT
Perform send

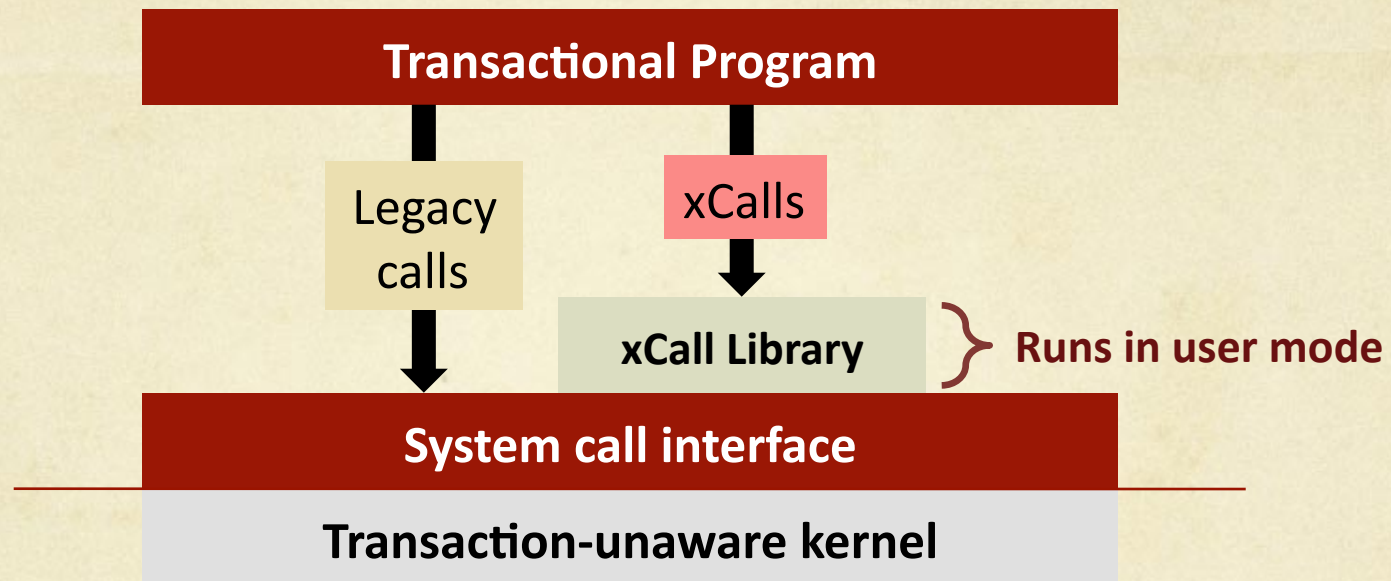
Defer

Deferred
send fails



Internet

Contribution



- xCall programming interface
 - Exposes transactional semantics to programmer
 - Enables I/O within transactions w/o stopping the world
 - Exposes all failures to the program

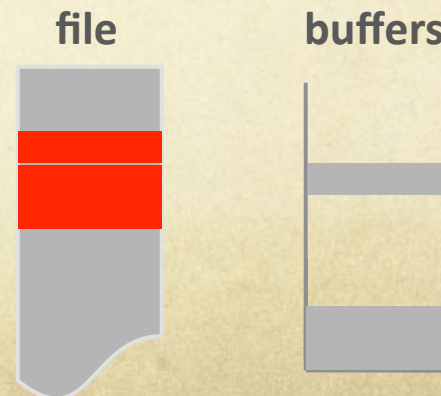
Atomic execution

- Provide abort semantics for kernel data and I/O
- Expose to programmer when action is performed

Can reverse action?	Need result?	Execution	Example
Yes	-	In-place	<code>x_write()</code>
No	No	Defer	<code>x_write_pipe()</code>
No	Yes	Global lock	<code>ioctl()</code>

```
➔ atomic {  
    item = procltem(queue);  
    x_write (file, principal);  
    x_write (file, item->header);  
}
```

Abort



Isolation

- Prevent conflicting changes to kernel data made within a transaction

➤ Sentinels

- Revocable user-level locks
- Lock logical kernel state visible through system calls

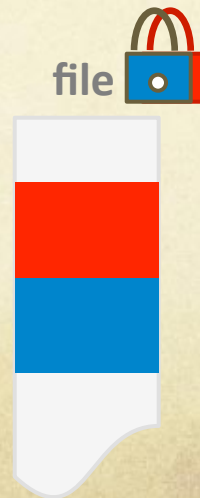
Thread 1

```
➔ atomic {  
  item = procltem(queue);  
  x_write (file, principal);  
  x_write (file, item->header);  
}
```

memory



file



Thread 2

```
➔ atomic {  
  item = procltem(queue);  
  x_write (file, principal);  
  x_write (file, item->header);  
}
```

Conflict

Error handling

- Some errors might not happen until transaction commits or aborts

➤ Inform programmer when failures happen

- Handle errors after transaction completes

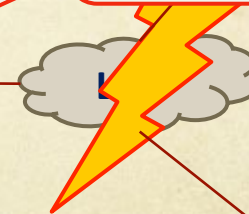
```
➔ atomic {  
    item = procItem(queue);  
    x_write (file, principal, &err1);  
    x_write (file, item->header, &err2);  
    x_send (socket, item->body, &err3);  
}  
if (err1 || err2 || err3) {  
    /* CL  
}
```

COMMIT
Perform send

Handle error here

Defer

Deferred send: FAILED
err3 = error



Summary

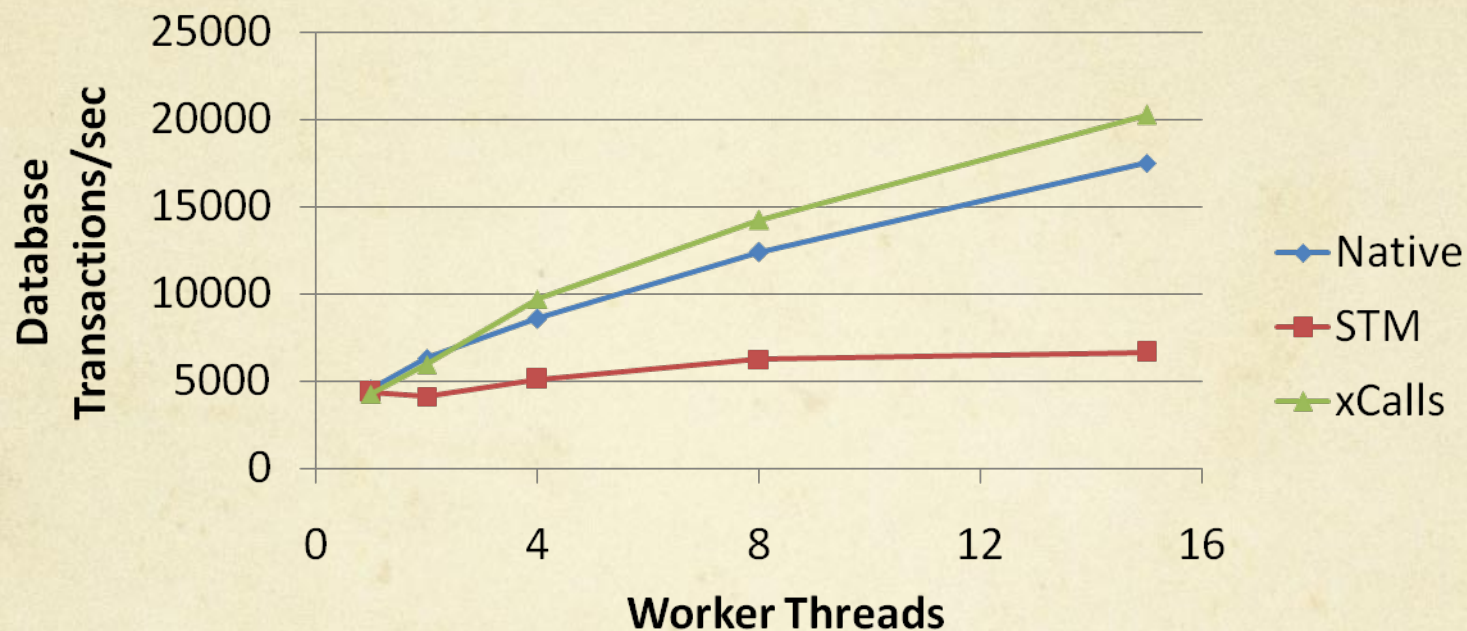
- xCall API exposes transactional semantics
 - Atomicity
 - Isolation
 - Error handling
- Prototype implementation
 - Executes as user-mode library
 - Relies on Intel STM for transactional memory
 - Provides 27 xCalls including file handling, communication, threading

Evaluation platform

- Transactionalized three large multithreaded apps
 - Berkeley DB : locking + logging subsystems (31 tx)
 - BIND: logging + memory subsystems (87 tx)
 - XMMS
- Configurations
 - **Native**: locks + system calls
 - **STM** : transactions + system calls + global lock
 - **xCalls**: transactions + xCalls
- Run on 16 core (4 x quad) 2 GHz AMD Barcelona

Performance: Berkeley DB

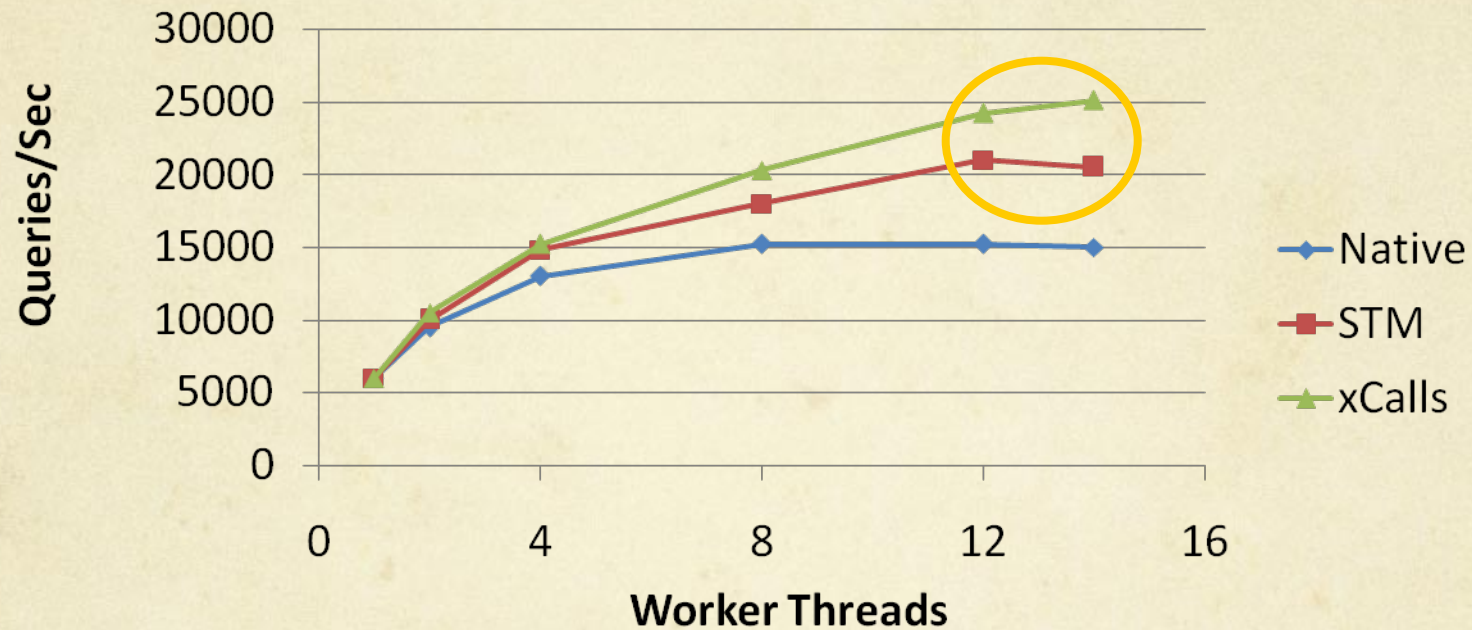
- Workload: Lockscale



- Global lock kills optimistic concurrency
- xCalls improve concurrency over native coarse-grained lock

Performance: BIND

- Workload: QueryPerf



- Transactions scale better than coarse grain locks
- xCalls enable additional concurrency

xCalls Summary

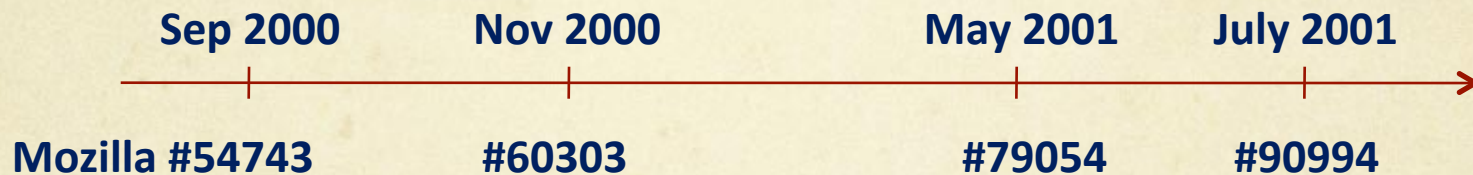
- Targeted use of TM can benefit legacy programs
- Transactional I/O is possible without kernel modifications

Outline

- Introduction
- xCalls: transactional access to OS services
- **TM as a concurrency bug fix**
 - Deadlock
 - Atomicity violations
- Conclusions

Concurrency Bugs

- Fixing concurrency bugs is challenging
 - Often adhoc [Lu ASPLOS'08]
 - Hard to get it right
 - In Mozilla, fixing one deadlock bug introduced another



- Existing code may benefit from TM as a **concurrency bug fix**

Example: Deadlock in Mozilla

Worker Thread

nsSocketReadRequest::OnRead
nsSocketTransport.cpp

~~Socket
Lock~~

PrepareAndDispatch
xptcstubs.cpp

UI
Lock

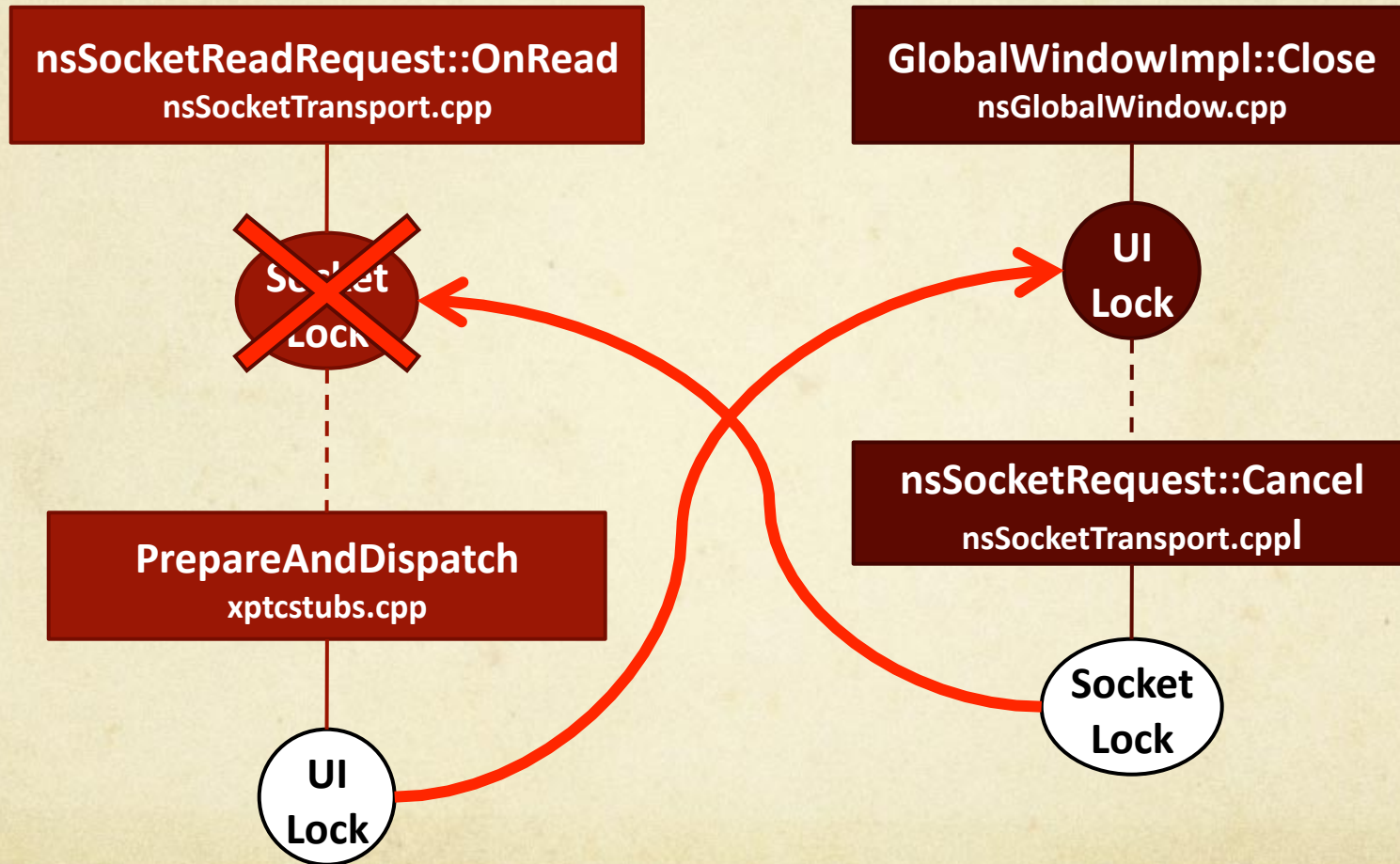
UI Thread

GlobalWindowImpl::Close
nsGlobalWindow.cpp

UI
Lock

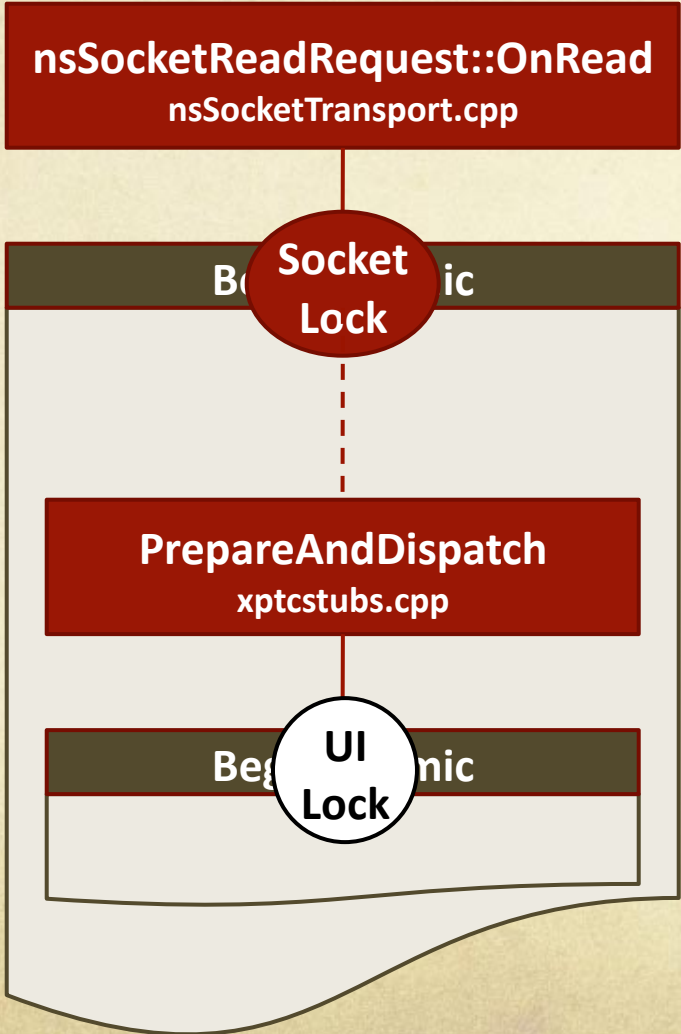
nsSocketRequest::Cancel
nsSocketTransport.cppl

Socket
Lock

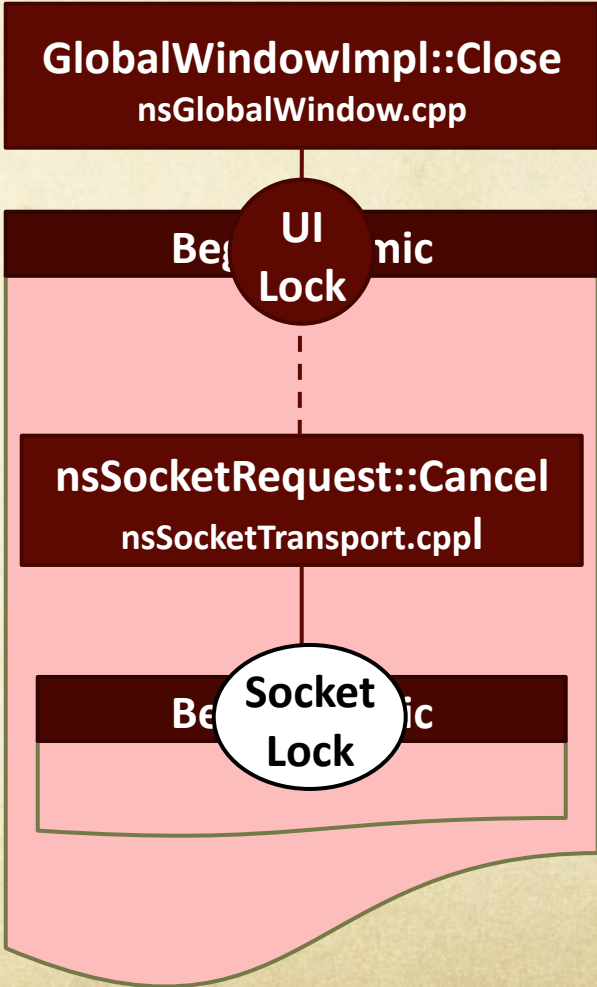


Fixing Mozilla Deadlock with TM

Worker Thread



UI Thread



Applying TM: Methodology

- Studied **78** *previously found and fixed* concurrency bugs in Mozilla, MySQL, Apache
- Classified bugs in **3** categories
 - Deadlock
 - Atomicity violation
 - Ordering violation
- Asked **3** questions
 - Can TM fix the bug?
 - Can TM simply fix the bug?
 - Can TM efficiently fix the bug?

} **Applicability**

Outline

- Introduction
- xCalls: transactional access to OS services
- TM as a concurrency bug fix
 - **Deadlock**
 - Atomicity violations
- **Conclusions**

Naïve Deadlock Fix

- Solution:
 - Replace all locks with transactions
- Benefits:
 - Easy-to-read code
 - Optimistic concurrency

Naïve Deadlock Fix Drawbacks

- Widely distributed code changes
 - All uses of a lock must be replaced
 - Requires system call support
- Performance overhead
 - 76% decrease for some bugs with STM

Can We Do Better?

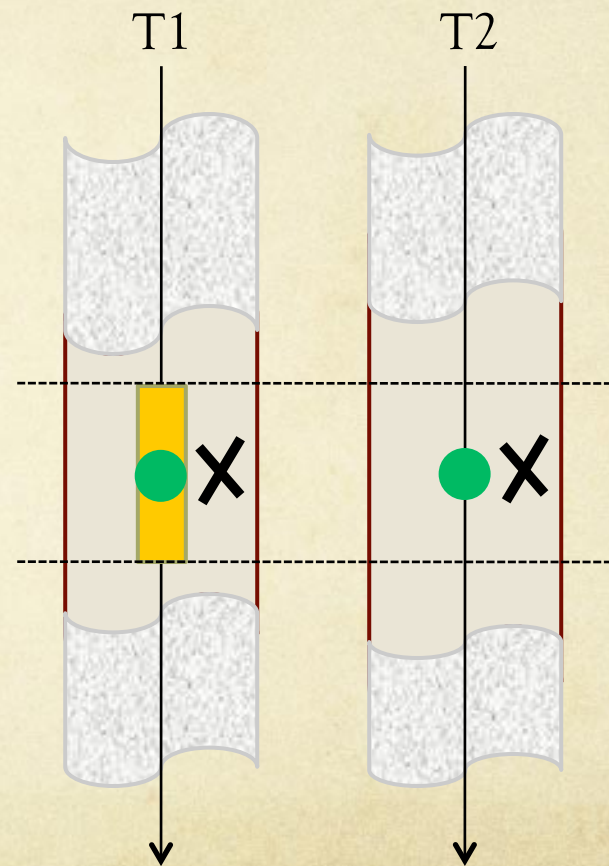
- Observation:
 - Only one thread needs to preempt to break deadlock
- Asymmetric deadlock preemption
 - Only one thread of a deadlock uses transactions + tx-safe locks
 - Remaining threads use only tx-safe locks

Fixing Deadlock Bugs Summary

- TM helps fixing **13 of 21** bugs
 - Preemption fixes **8** bugs
 - Converting locks to transactions fixes **5** bugs
 - Remainder require async I/O, involved condition variables, or modified unrelated state
- TM-fix for **10 of 13** bugs **simpler** than developers'
 - Fewer lines of code changed
 - Changes were more localized

Missing Synchronization Bugs

- Missing synchronization:
 - access to variable *never* uses a lock
- Partial synchronization:
 - Some access don't use a lock
- TM benefits:
 - Localized changes
 - Localized reasoning



Atomicity Violation Example

- Apache (httpd-2.0.45: mod_log_config.c)

```
void ap_buffered_log_writer (...)
{
    atomic_t buf[buf->outputCount];
    memcpy(&buf[buf->outputCount];
    memcpy(buf->outputCount + len;
    buf->outputCount + len;
    ap_buf->write(buf->handle);
}    apr_file_write(buf->handle); ← I/O
}
}
```

- Performs within 3% of developers' fix
- Changes only one function

Atomicity Violation Bugs Summary

- TM fixes **30 of 38** atomicity violations
 - Remainder do async I/O, cross modules, or are long
- TM-fix for **21 of 30** bugs is **simpler** than developers'
 - Localized reasoning about atomicity
 - Localized code changes
 - Less code involved

Summary

- TM can help fixing 58% of the bugs studied
 - Hard problems are still hard.
- TM fix is simpler than developer fix in 73% of these
- Sophisticated use of TM reduces change complexity and improves performance

Conclusions

- Transactional memory can benefit existing programs
- Targeted use reduces change complexity, performance problems
 - Highly contested critical sections
 - Deadlocks, atomicity violations
- System access important for legacy code

Questions?

Related Work

- Operating system access:
 - TxOs [SOSP'09]
 - QuickSilver [SOSP'91]
 - Tx diet libc [TRANSACTION'10]
- Concurrency bug repair
 - Deadlock Immunity [OSDI'08]
 - Atom-Aid [ISCA'08],
 - ISOLATOR [ASPLOS'09]