Using Software Transactional Memory In Interrupt-Driven Systems

Michael J. Schultz

Department of Mathematics, Statistics, and Computer Science
Marquette University

Thesis Defense
Software transactional memory can be used in interrupt-driven device drivers as a method to automate and provide fine-grained synchronization between the upper and lower halves.
Operating Systems

- **operating system**—low-level software on a computer.
- **application programming interface**—set of functions a programmer can use.
- **device driver**—deals with high-level API and low-level hardware.
- **interrupt**—external signal to processor.
- **jitter**—variations in delay of interrupt handling.
Jitter Avoidance

Don’t disable interrupts!

```plaintext
global integer our_var = 1

function increment(void)
    our_var++

.data
    .globl our_var
    our_var: 0x0001

.text
increment:
    lw    s0, 0x4c(zero)
    addiu s0, s0, 1
    sw    s0, 0x4c(zero)
    j     ra
```
Don’t disable interrupts!

global integer our_var = 1

function increment(void)
    our_var++

.data
    .globl  our_var
    our_var: 0x0001

.text
increment:
    lw     s0, 0x4c(zero)
    addiu  s0, s0, 1
    sw     s0, 0x4c(zero)
    j      ra
Jitter Avoidance

Don’t disable interrupts!

global integer our_var = 1

function increment(void)
    our_var++
Related Work

Transactional Memory

- Transactions began in database community.
- Brought to systems as concurrency control for multiprocessors.
- Hardware TM
  - Suffer from physical memory boundaries.
- Software TM
  - Blocking vs. non-blocking implementations.
  - Blocking STM allows progress guarantees.
- Hybrid TM
  - HTM is expensive and restrictive.
  - STM needs more space and time to compute.
  - Hybrid TM takes advantages from both (speed and reliability).
    - Also takes disadvantages.
Lock-free kernels: Synthesis and Cache.
Use CAS and DCAS opcodes for shared data structures.
TxLinux uses HTM and \textit{cxspinlocks}. 


Rossbach et al. “TxLinux: using and managing hardware transactional memory in an operating system.” SOSP ’07.

**Major Contributions**

- Modernized port of the Xinu kernel for IA-32.
- Method for integrating STM library into kernel.
- Embedded Xinu augmented with STM:
  - “Transactional Xinu.”
- Evaluation of interrupt-driven device drivers in Transactional Xinu.
Outline

1. Design and Implementation
   - Critical Sections
   - Transactional System

2. Performance Results
   - Measurements
   - Testing Methodology

3. Summary and Future Work
   - Summary
   - Future Work
Outline

1. Design and Implementation
   - Critical Sections
   - Transactional System

2. Performance Results
   - Measurements
   - Testing Methodology

3. Summary and Future Work
   - Summary
   - Future Work
Device Driver Structure

Upper half with thread-level API

Shared data buffers and state

Interrupt-driven Lower half
A *critical section* is a piece of code that accesses shared data or resources in the system and must not be accessed by two or more processes simultaneously.
Critical Sections

Properties

- **atomically**—perform indivisible operations “instantaneously.”
- **consistent**—no illegal system state will exist.
- **isolation**—no thread will see intermediate state.
- **durable**—no reversion after completion.

*These are known as the ACID properties.*
*We are only interested in the A, C, and I properties.*
Critical Sections

Problems

- Deadlock

- Priority Inversion
Variations in the amount of time the system takes to respond to incoming interrupts.
• STM library assures that the ACI properties are followed.
• Compiler and library automatically handle rollbacks and commits.
Interrupts are implicitly given highest priority in system.

STM can be set up to give interrupt-driven transaction highest priority.

User-level thread will not prevent interrupt from entering.

I/O operations are difficult because they cannot be taken back.
Xinu was developed 25 years ago by Doug Comer.

Provide a easy-to-understand O/S for teaching and research.

Under 20,000 lines of code, but still a rich experimentation platform.

lightweight thread model, shared memory space, preemptive multitasking priority scheduler, synchronization primitives, IPC, and device drivers.
Transactional Xinu

- Built on top of Embedded Xinu.
- Provides needed components for transactions.
  - POSIX-thread library.
  - Thread-local storage.
  - Intel’s STM library.
POSIX-thread library

- Keep the Xinu model: lightweight.
- pthread_key_create
- pthread_setspecific
- pthread_create
- pthread_join
Interrupt-local Storage

- Thread-local storage gives private memory to every thread.
- Interrupts push state onto stack, does not update GS register.
- Interrupt-local storage switches GS context for interrupts.
Transactional Library

- Several modes of operation: optimistic, pessimistic, serial, and obstinate.
- Optimistic and pessimistic are “normal” modes.
- Serial mode works with legacy and irrevocable operations.
- Obstinate allows for “stubborn” transactions.
Transactional Library

Single Global Lock Atomicity

- Creates an equivalence between global lock code and atomic code.
- STM library actually uses SGLA for serialized transactions.

```c
__tm_atomic {
    Statements; → Statements;
}
wait(global_lock);
signal(global_lock);
```
Transactional Library
Obstinate Mode

- Library lets an obstinate transaction beat all conflicting transactions.
- Provides an interface allowing the programmer to declare an obstinate transaction.
- Transactional Xinu uses this in interrupt handlers.
Transactional Library

Versioning and Logging

- Read versioning tracks the version number of a variable.
- Writing will obtain a lock and increment the version number.
- Undo logging saves original values to private memory.
Transactional Library

Versioning Example

### Reader Code

```plaintext
global integer our_var
local integer my_var

atomic
if ( our_var = 1 )
    my_var = our_var + 1
else
    my_var = 1

{} → {our_var : 2}
```

### Writer Code

```plaintext
global integer our_var

atomic
    our_var = our_var + 1

{} → {our_var : 2} → {our_var : 3}
```
2PL for contention manager.
Maps every contended memory location to a unique lock.
Mapping performed at runtime, takes \( \sim 4 \) MB of memory.
Transactional Xinu minimizes size of atomic sections.
Transactional Device Drivers

- Wrap upper half critical sections in `tm_atomic`.
- *Wrap lower half shared data in `tm_atomic`.*
- Every function call and incidental function call must be re-instrumented.
### Code Size Differences

<table>
<thead>
<tr>
<th>Kernel image</th>
<th>Non-STM</th>
<th>STM</th>
<th>Increase</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw</td>
<td>365,628</td>
<td>595,251</td>
<td>229,623</td>
<td>62.80%</td>
</tr>
<tr>
<td>stripped</td>
<td>351,048</td>
<td>540,504</td>
<td>189,456</td>
<td>53.97%</td>
</tr>
<tr>
<td>excluding STM library (170,869 bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>raw</td>
<td>365,628</td>
<td>424,382</td>
<td>58,754</td>
<td>16.07%</td>
</tr>
<tr>
<td>stripped</td>
<td>351,048</td>
<td>369,635</td>
<td>18,587</td>
<td>5.29%</td>
</tr>
</tbody>
</table>
Transactional System

- Transactional Xinu was built to run on real hardware, available today.
- Built on mid-model Pentium 4, 3.0 GHz processor (“Northwood”).
- No BIOS calls or Linux code used behind-the-scenes.
Loading the Kernel

```c
#include <stdio.h>
#include <memory.h>
#include <mutex.h>

mutex global_lock;
int global_var;

int main(int n, char **argv) {
    int local_var = 0;
    wait(global_lock);
    global_var = global_var + 2;
    signal(global_lock);
    if (local_var >= 2) {
        local_var = 0;
        return local_var;
    }
    local_var = 0;
    return local_var;
}
```
Measurements

- On-chip: read timestamp counter (`readtsc`).
- In-system: timer counters (thread time, monotonic time).
- External: min, avg, max, mdev of round-trip time.
• Packet enters machine and raises interrupt (RX_TSC).
• Upper half call transfers incoming data (READ_TSC).
• Upper half call transfers outgoing data (WRITE_TSC).
• Final call in lower half of driver (TX_TSC).
Data gathering occurs at runtime, stores in memory until needed.

Timer interrupt fires every 1/10 of a millisecond, lightweight.
Ping Testing
Ping 1000 Millisecond Interval

- **Average (Non-STM)**
- **Average (STM)**

<table>
<thead>
<tr>
<th>measure</th>
<th>min</th>
<th>avg</th>
<th>max</th>
<th>mdev</th>
</tr>
</thead>
</table>
| milliseconds | 0.0 | 0.2 | 0.4 | 0.6 | 0.8 | 1.0 | 1.2 | 1.4 | Average (Non-STM) | Average (STM)

Michael J. Schultz
Using STM In Interrupt-Driven Systems
Ping Testing
Ping 500 Millisecond Interval

- **Average (Non-STM)**
- **Average (STM)**

<table>
<thead>
<tr>
<th>measure</th>
<th>min</th>
<th>avg</th>
<th>max</th>
<th>mdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>milliseconds</td>
<td>0.0</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Michael J. Schultz  Using STM In Interrupt-Driven Systems
Ping Testing
Ping Flood (0 Millisecond Interval)

- **measure**
  - min
  - avg
  - max
  - mdev

- **Average (Non–STM)**
- **Average (STM)**

- **milliseconds**
  - 0.0
  - 0.2
  - 0.4
  - 0.6
  - 0.8
  - 1.0
  - 1.2
  - 1.4

**Testing Methodology**

Michael J. Schultz

Using STM In Interrupt-Driven Systems
Machine specific

100,000 micro-operations takes about 33 microseconds.
Ping Testing
TSC 1000 Millisecond Interval

micro-op cycles

RX_TSC  READ_TSC  WRITE_TSC  TX_TSC

Average (Non-STM)  Average (STM)

Michael J. Schultz  Using STM In Interrupt-Driven Systems
Ping Testing
TSC 500 Millisecond Interval

RX_TSC
READ_TSC
WRITE_TSC
TX_TSC

critical region
micro-op cycles
0 20000 40000 60000 80000 100000

Average (Non-STM)
Average (STM)

Michael J. Schultz
Using STM In Interrupt-Driven Systems
Ping Testing
TSC Flood (0 Millisecond Interval)

<table>
<thead>
<tr>
<th></th>
<th>RX_TSC</th>
<th>READ_TSC</th>
<th>WRITE_TSC</th>
<th>TX_TSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average (Non-STM)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average (STM)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Michael J. Schultz  
Using STM In Interrupt-Driven Systems
Measuring jitter is difficult.

Lightweight timestamp counter at beginning of handler.

Packet generator, sending precisely timed packets.
### Timing Testing Results

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-STM</td>
<td>30552911.53</td>
<td>30525638</td>
<td>30582946</td>
<td>13847.91</td>
</tr>
<tr>
<td>STM</td>
<td>30553291.98</td>
<td>30534954</td>
<td>30576856</td>
<td>13223.17</td>
</tr>
<tr>
<td>Difference</td>
<td>380.45</td>
<td>316</td>
<td>-6090</td>
<td>-624.74</td>
</tr>
</tbody>
</table>
Summary

- HTM systems are hard to build, use STM instead.
- Initial results suggest that TM might be able to reduce jitter.
- Transactional Xinu is built to use Intel’s STM library and compiler.
  - Adds POSIX library and interrupt-local storage.
  - Specially instrumented interrupt handlers (tm_atomic and tm_callable)
- Experimentation shows software overhead is minimal in some scenarios.
  - Many other components still disable interrupts.
Future Work

- More tightly integrate STM library to Embedded kernel (scheduler, interrupts, etc.)
- Test scaling to multi-core systems (what TM was designed for).
- Test scaling with multiple network interfaces (ZigBee, WiFi, Bluetooth, GigE).
- Analyze real-time properties, if interrupts can always be received what happens (“interrupt overload”).

mschul@mscs.mu.edu
http://www.mscs.mu.edu/~mschul/