The Linux Kernel:

A Challenging Workload for Transactional Memory

Hany E. Ramadan
Christopher J. Rossbach
Emmett Witchel

Operating Systems & Architecture Group
University of Texas at Austin

Talk overview

- Why OSes are interesting workloads (1)
- Interrupts (2)
  - Transaction stacking (3)
- Configurable contention management (1)
- Other issues considered in the paper (1)
- Preliminary results (2)

Why OSes are interesting workloads

- Large concurrent program with interacting subsystems
- Complex, will benefit from ease of programming and maintainability
- Lack of OS scalability will harm application performance
- Diverse primitives for managing concurrency
  - spinlocks, semaphores, per-CPU variables, RCU, seqlocks, completions, mutexes

Interrupts

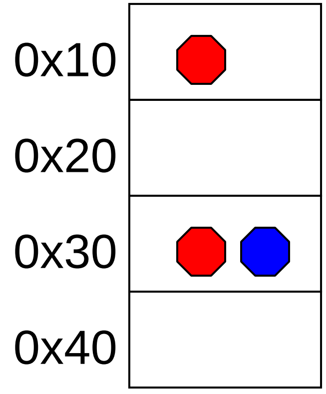
- Cause asynchronous transfer of control
- Do not cause a thread switch
- Are more frequent than thread switches
- May interrupt other interrupt handlers

- *Question*: How does a kernel which uses transactional memory handle interrupts?

Using transactions in interrupt handlers

system_call()	intr_handler()
{	{
XBEGIN	XBEGIN
modify 0x10	modify 0x30
XEND	XEND
}	}

interrupt



No tx in interrupts

TX #1 { 0x10 }

Interrupts abort active tx

TX #1 { 0x10 }

TX #2 { 0x30 }

Nest the transactions

TX #1 { 0x10, 0x30 }

Multiple active transactions

TX #1 { 0x10 }

TX #2 { 0x30 }

Benefits of multiple active Tx

- Most flexibility for programmer
  - Interrupt handlers free to use Tx as necessary
- Aborts only when necessary
  - Interrupts are frequent
- Interrupt handlers stay independent

- Implies..
  - Multiple transactions on a single thread !

Multiple transactions per thread

- Many transactions may be simultaneously active but at most one is running per thread
  - They can conflict with each other
  - Independent (no nesting relation)
- Stacked transactions
  - Transactions complete in LIFO order
  - Each thread has a logical stack of transactions
- Stacked transactions ideal for interrupts
  - Stack grows and shrinks as interrupts occur and complete

Multiple Tx Per Thread - Open questions

- What are the roles of HW and SW
  - ISA changes for managing multiple transactions
  - Efficient HW implementation
- Contention management must know about stacking
  - Stacked transactions can livelock
- Identifying other scenarios where this is useful
  - Non-interrupt cases?
  - Forms other than stacking?
- Program stack issues

Configurable Contention Management

- Contention can be heavy within OS
  - Transactions most effective when contention is rare
- OS contains programmer hints for contention management
  - RCU (read-change-update) favor readers
  - Seqlocks favor writers
- Hardware TM should accept programmer hints
  - XBEGIN takes contention mgmt parameter

Other issues considered in the paper

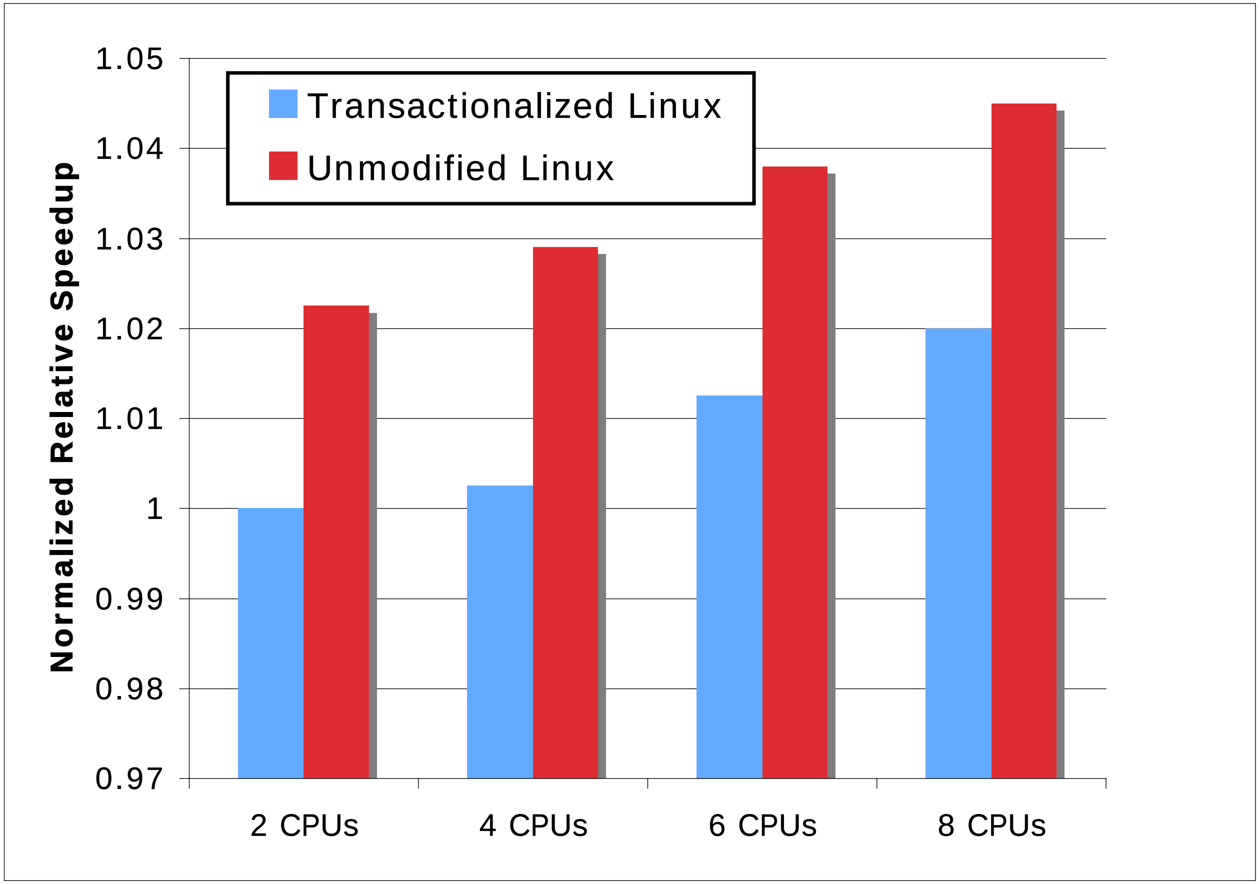
- Primitives for which transactional memory might not be suitable
  - Per-CPU data structures
  - Blocking operations
- I/O in transactions
  - Big issue for Linux
    - I/O is frequently performed while spinlock held
  - May be possible to just allow it
    - TLB shootdown

Implementation

- Implemented HTM as extensions to x86
  - With multiple active transactions
- Modified many spinlocks in Linux kernel (2.6.16.1) to use transactional memory
- Simulation environment
  - Simics 3.0.10 machine simulator
  - 16KB L1 ; 4MB L2 ; 256MB RAM
  - 1 cycle/instruction, 200 cycle/memory miss

Preliminary Results

- We are booting Linux
  - Transactions speed up boot by ~2%



Fin