TxLinux: Using and Managing Hardware Transactional Memory in an Operating System

Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan,

Aditya Bhandari, and Emmett Witchel

Presented by Jonathan Walpole

Overview

- TxLinux
  - rewrite of Linux kernel to use hardware TM
  - based on MetaTM simulator
- cx_spinlocks
  - new hybrid TM and locking primitive
- Integration of scheduler and TM
  - helps avoid problems during contention
- Performance evaluation

Claimed advantages of TM

- Easier concurrent programming
  - fewer program states
  - performance of optimistic execution
  - no deadlock
  - composability
  - true concurrency
  - longer critical sections
  - isolation
  - efficiency, all done in hardware

HTM primitives in MetaTM

- xbegin - start transaction
- xend - commit transaction
- xrestart - abort and retry
- xgettxid - get current transaction identifier
- xpush - save transaction state on interrupt
- xpop - restore transaction state
- xtest - add variable to data set if value matches
- xcas - subject non-transactional threads to contention management

Hardware Transactional Memory

- Transactions *conflict* when:
  - the *write set* of one transaction intersects with the union of the *read set* and *write set* of another
- Read set
  - set of addresses read by the transaction
- Write set
  - set of addresses written by the transaction
- Granularity?

Hardware Transactional Memory

- Conflict manager
  - decides which conflicting transactions can proceed
  - losers discard all changes and restart
- Asymmetric conflict
  - conflict between transactional and non-transactional access to a set of addresses
- Complex conflict
  - involves more than two transactions
  - write to an address that has been read by many

Virtualizing Transactions

- Transaction size may exceed the size of the hardware's transactional cache
  - can't just fail because it would impact portability of software
  - fall back (fault) to software TM
  - transparent to application
    - except for performance
    - so long as the semantics of the HTM and STM are identical

Goal: A Linux Kernel Based on Transactions

- First attempt at straight substitution of transactions for locks failed
- Problems
  - I/O
  - memory that's read by hardware (page tables)
  - performance under contention
  - interactions with non-transactional threads

Goal: A Linux Kernel Based on Transactions

- Second attempt:
  - spinlocks -> cx_spinlocks
  - reader/writer spinlocks -> cx_spinlocks
  - atomic operations -> xcas
  - sequence locks -> transactions
  - RCU write side spinlocks -> cxspinlocks
  - semaphores, RCU readers, etc -> not converted

Why Not Just Use Locks in Transactions?

- Spinning causes conflict if the lock variable is included in the transactional data set
- Asymmetrical conflicts between transactional and non-transactional threads are not resolved fairly
  - transactional threads wait for non-transactional lock holders
  - non-transactional threads attempting to acquire cause transactional lock holders to abort
  - conflicts must be decided in favor of non-transactional threads because they can't roll back

Solution?

- cx_spinlocks
  - cooperative transactional spinlocks
  - a new synchronization primitive that dynamically and automatically chooses between locks and transactions as necessary

Cooperative transactional spinlocks (cxspinlock)

- Allow a critical section to “sometimes” be protected by a lock and other times by a transaction
- Allows the same data structure to be accessed from different critical regions that are protected by transactions or locks
- I/O handled automatically
- Provides a simple lock replacement in existing code

cx_spinlock Features

- multiple transactional threads can enter the critical section without conflicting
  - the lock variable is excluded from the data set
- non-transactional threads holding the lock exclude transactional and non-transactional threads
- transactional threads poll the spinlock without restarting
- non-transactional threads use xcas to get the spinlock, which is arbitrated by the contention manager

cx_spinlock Primitives

- cx_optimistic
  - execute critical section transactionally
- cx_exclusive
  - locks critical section
  - contention manager decides if transactional holders can be preempted
  - causes transactional thread to use pessimistic concurrency control

Example Use: Replacement for spin_lock_irq

```
void cx_optimistic (lock) {
    status = xbegin ;
    // Use mutual exclusion if required
    if ( status == NEED_EXCLUSIVE ) {
        xend ;
        // xrestart for closed nesting
        if ( gettxid ) xrestart ( NEED_EXCLUSIVE ) ;
        else cx_exclusive ( lock ) ;
        return ;
    }
    // Spin waiting for lock to be free (==1)
    while ( xtest ( lock, 1 ) = = 0 ) ; // spin
    disable_interrupts ( ) ;
}
```

Example Use: Replacement for spin_lock_irq

```
void cx_exclusive ( lock ) {
    // Only for non−transactional threads
    if ( xgettxid ) xrestart ( NEED_EXCLUSIVE ) ;
    while ( 1 ) { // Spin waiting for lock to be free
        while ( ∗ lock != 1 ) ; // spin
        disable_interrupts ( ) ;
        // Acquire lock by setting it to 0
        // Contention manager arbitrates lock
        if ( xcas ( lock , 1 , 0 ) ) break ;
        enable_interrupts ( ) ;
    }
}
```

Example Use: Replacement for spin_unlock_irq

```
void cx_end ( lock ) {
      if ( xgettxid ) {
            xend ;
      }
      else {
            * lock = 1 ;
      }
      enable_interrupts ( ) ;
}
```

Some Issues

- I/O
  - Processor traps attempts to do I/O and automatically restarts transactions with cx_exclusive
- Naked xbegins
  - can cause infinite loops if they do not check their transaction status on restart
- Virtualizing transactions
  - cx_exclusive could be called when a transaction overflows the transactional cache
  - no need to fall back to STM … but semantics are different

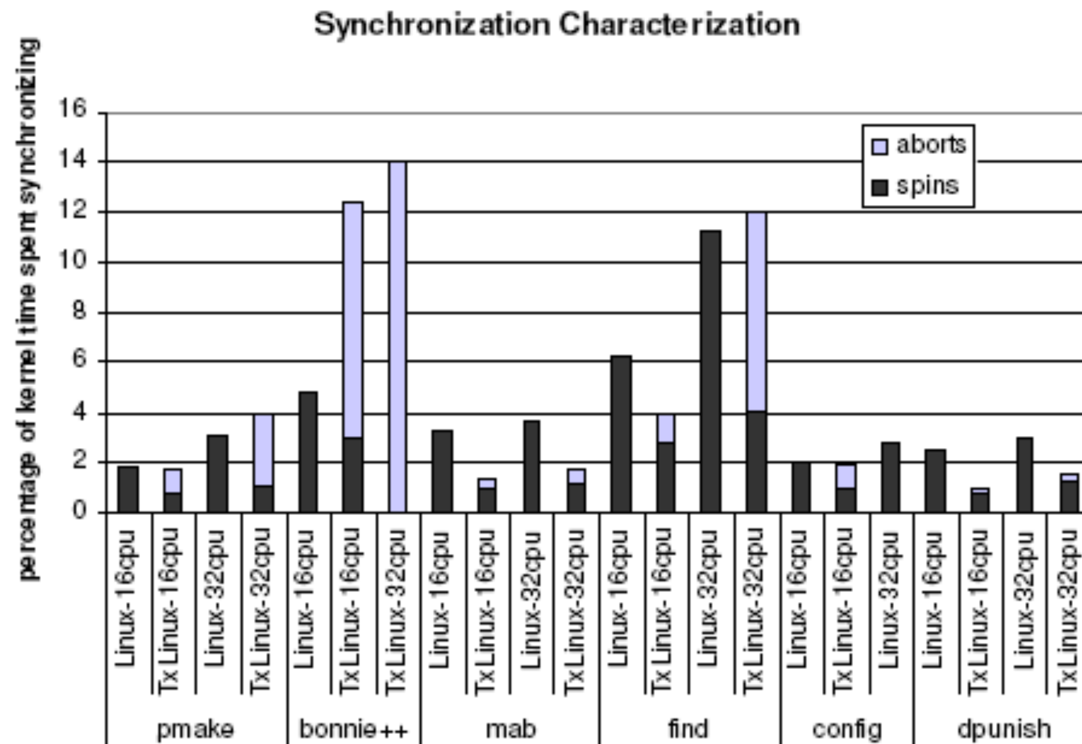
More Issues

- Deadlock is now possible again
  - cx_spinlocks and blocking, like locks
- Deadlock is possible due to interactions with contention management policies
  - its like the priority inversion problem of locking
- Programming complexity
  - cx_spinlocks appear to be harder to program with than either spinlocks or transactions alone
  - but transactions alone were insufficient

Scheduling in TxLinux

- Priority and Policy inversion still possible with transactions
- Contention management based on
  - conflict priority first, then size, then age
  - scheduler registers priority with hardware
- Transaction-aware scheduling
  - Dynamic Priority based on HTM state
  - Conflict-reactive descheduling

Performance (1)



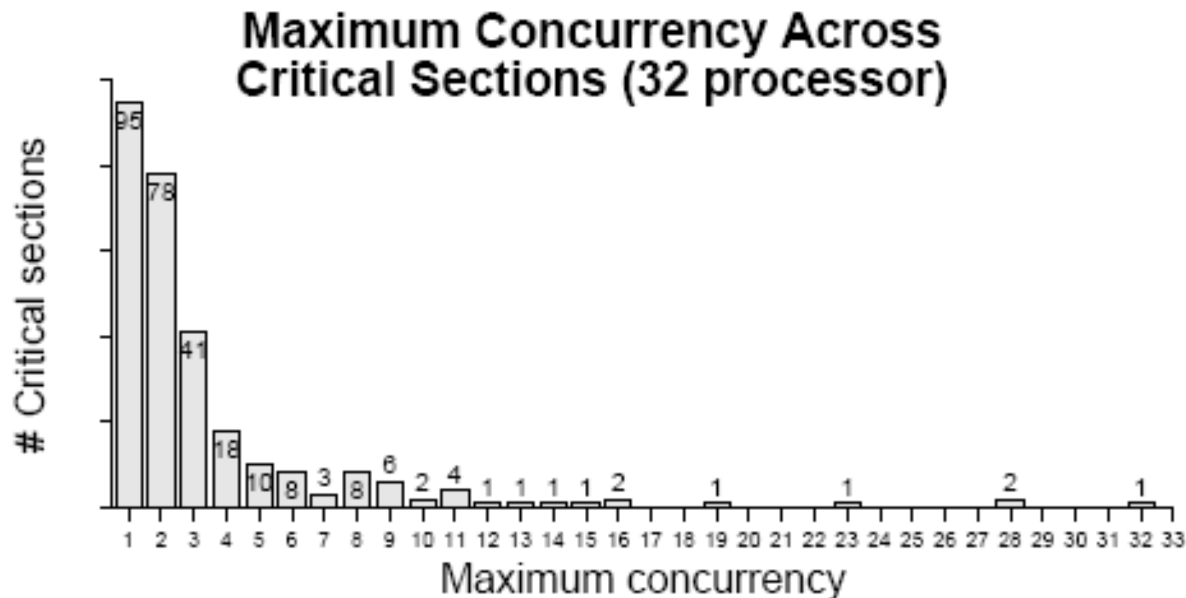
Time lost due to restarted transactions and acquiring spin locks in 16 & 32 CPU experiments
(TxLinux saves time on 16 cpus and loses time on 32 cpus)

Performance (2)

		Linux			TxLinux		
		Acq	TS	T	Acq	TS	T
bonnie++	16	12,478	132	340,523	28%	20%	68%
config	16	16,087	62	49,432	31%	56%	33%
dpunish	16	9,626	35	18,406	51%	66%	32%
	32	10,514	102	153,699	49%	39%	6%
find	16	2,912	72	34,553	39%	42%	14%
	32	2,758	183	111,629	40%	52%	21%
mab	16	15,451	101	45,167	51%	81%	55%
	32	15,871	146	96,370	50%	71%	39%
pmake	16	764	9	8,981	30%	38%	24%
	32	1,004	24	35,341	25%	48%	18%

Spinlock performance for an unmodified Linux vs.
the subsystem kernel TxLinux-SS
(TxLinux reduces the number of acquisitions, tests and test&sets)

Performance (3)



Distribution of maximum concurrency across TxLinux-CX critical sections for the benchmark programs on 32 processors
(There are critical sections that could benefit from concurrency)

Performance (4)

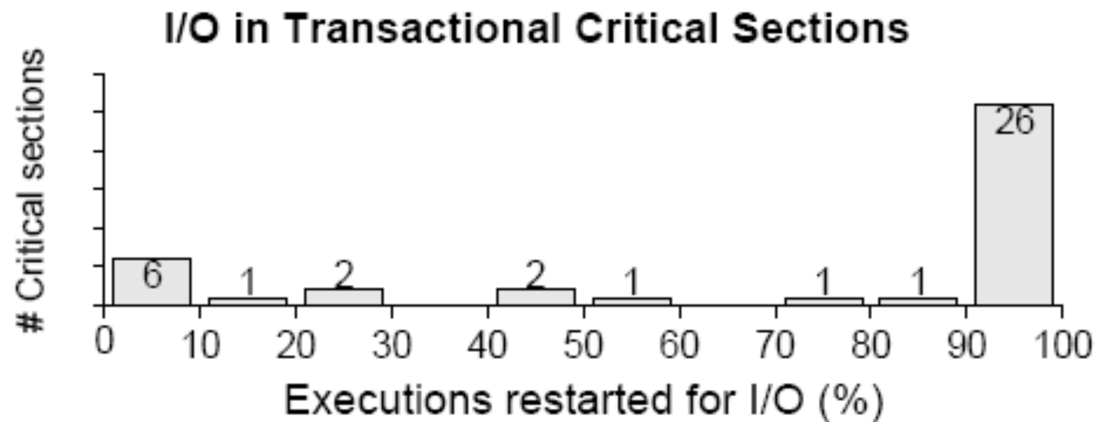


Figure 6: Distribution across TxLixux-CX critical sections of the percentage of executions that require restarts for I/O, measured with the config, find, mab and pmake benchmarks with 16 and 32 processors.

(There are some critical sections that only sometimes do I/O)

Performance (5)

		I/O		Origin (SS)		Origin (CX)	
		Nest	Waste	sys	intr	sys	intr
config	16	1.42	32.3%	46.3%	53.7%	49.6%	50.4%
	32	1.36	36.3%	45.9%	54.0%	49.8%	50.2%
find	16	1.51	0.3%	74.8%	25.2%	68.6%	31.4%
	32	1.39	2.8%	79.5%	20.5%	67.8%	32.2%
mab	16	1.36	13.7%	73.4%	26.6%	63.6%	36.4%
	32	1.30	31.2%	73.2%	26.8%	63.8%	36.2%
pmake	16	1.51	0.3%	51.5%	48.4%	21.3%	78.7%
	32	1.50	0.3%	48.6%	51.2%	15.1%	84.9%

Cxspinlock usage in TxLinux
(Nesting depth for I/O is low, but there is still substantial waste due to restarts)

Performance (6)

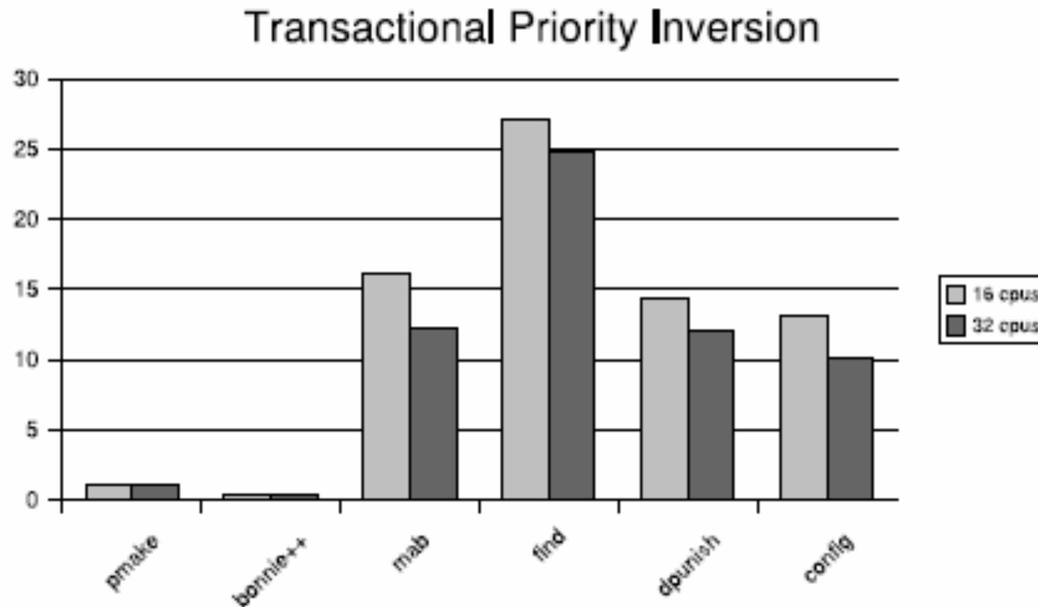


Figure 7: Percentage of transaction restarts decided in favor of a transaction started by the processor with lower process priority, resulting in "transactional" priority inversion. Results shown are for all benchmarks, for 16 and 32 processors, TxLinux-SS .

(There is substantial priority inversion due to contention management)

Performance (7)

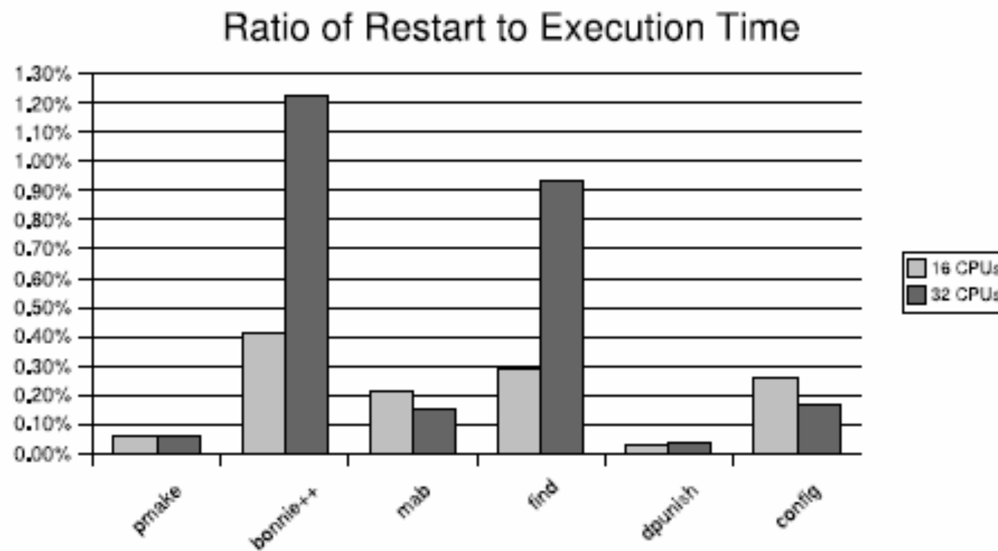


Figure 8: Restart cycles as a percentage of total execution time for TxLinux-default (SS) with 16 and 32 cpus. The percentage of restart cycles gives a theoretical upper bound on the performance benefit achievable by a scheduling policy that attempts to minimize restart waste.

(Restarts are low, but increase with CPU count)

Revisiting the Claims

- Easier concurrent programming?
  - fewer program states?
  - performance of optimistic execution?
  - no deadlock?
  - composability?
  - true concurrency?
  - longer critical sections?
  - isolation?
  - efficiency, all done in hardware?

Conclusion

- Performance comparable to locking
- Coding complexity potentially reduced
- New cx-spinlock primitive enables co-existence with locking
- But introduces new pathologies