Mutex Locking versus Hardware Transactional Memory: An Experimental Evaluation

Thesis Defense—Master of Science

Sean Moore

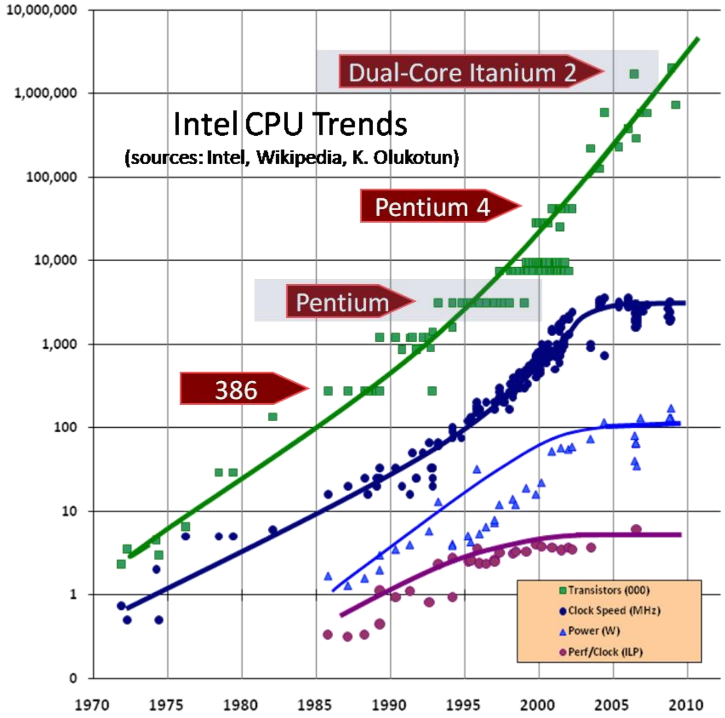
Advisor: Binoy Ravindran

Systems Software Research Group

Virginia Tech

Multiprocessing - the Future is Now

- Processors with multiple cores are widely available.
- CPU improvements aiding serial performance has largely ceased.



Systems Software Research Group

VirginiaTech Invent the Future

Motivation

- PARSEC's fluidanimate
  - Smoothed Particle Hydrodynamics for animation
- Fine-grained Futex: complex, fast
- Global Futex: simple, slow (~6.62x slower)
- Global Fallback HTM: simple, quick (~1.16x slower)

Configuration (at 8 threads)	Region-of-Interest Duration (s)
Fine-grained Futex	69.1243
Global Futex	457.904
Fine-grained HTM	76.3357
Global HTM	79.861

Contributions

- Global locking glibc
  - Available under open source
- Global lock fallback HTM is competitive with fine-grained futex
  - 23 applications
  - No source code modification necessary
- Describe *lock cascade failure*

Background: Mutex Locks

- A*cquire* and *release* semantics
  - Critical sections
  - Blocks thread process on contention
  - Pessimistic, mutually exclusive access
- Does not directly protect data
  - *Protect data, not code*
- Constrains race conditions which may cause inconsistent state

Background: Race Conditions

- Two threads increment a variable
  - No synchronization: lost increments
  - Synchronization: no lost increments
- What if *b* were a dereference?
  - How does *b* need to be protected?
  - Does locking *b*'s mutex violate a lock ordering scheme?

Potential Data Race

```
c = b
c = c+1
b = c
```

Removed Data Race

```
lock(a)
c = b
c = c+1
b = c
unlock(a)
```

Background: Livelock and Deadlock

- Deadlock
  - N≥1 threads eventually depend on themselves progressing to progress
  - Lock ordering scheme (DAG)
    - May require acquisition in an inefficient order
- Livelock
  - N≥1 threads perform work but cannot ultimately progress
  - Lock ordering schema circumvented with trylock+rollback
  - Complex analysis (*see thesis for extended example*)
- Efficient to program? Efficient to maintain?

Background: Transactional Memory

- *Begin* and *commit* semantics
  - Atomic sections
  - Does not necessarily block thread progress on contention
  - Optimistic, allows mutually shared access
- Directly Protects Data
  - Read-sets and write-sets
- Redo work when race conditions are detected

Background: Fallback Locks

- STM and (best-effort-only) HTM
  - Intel's Restricted Transactional Memory (RTM)
- Best-effort-only cannot guarantee completion
  - Various abort causes plus true conflicts
- HTM fallback onto futex locks
- Elision-Fallback Path Coherence
  - Eager subscription
  - Lazy subscription

Related Work: C++ Draft TM in GCC

- Proposal to add TM to C++ language
  - Implements syntactic atomic sections
- Acts as if guarded by a global lock
- Requires source code modifications
- Neither STM nor HTM-specific
  - Duplicated functions for instrumentation

Related Work: TM memcached

- Ruan et al. converted memcached for C++ TM
  - Convert critical sections to atomic sections
  - Modify condition synchronization
  - Replace atomic and volatile variables
- Concluded that incremental transactionalization is not generally likely
- Logically simple C library functions incur irrevocable serialization
  - String length

Related Work: glibc RTM

- GNU C Library (glibc) implements elision locking
  - Intel RTM with fine-grained futex fallbacks
- Attempts outermost transaction 3 times
  - Except for trylocks, only tries once
- No anti-lemming effect code
- Transaction backoff with a no-retry abort
  - Acquire lock at least 3 times before eliding again

glibc Library: Global Lock

- Added support for a library-private global lock
- Transparently substitutes global lock in-library
- Recursive locking
  - Acquire lock *a* then *b*, must be recursive when reduced
  - Recursion counter is allocated thread-local
- Full function called only when recursion counter is 0
  - Acquire succeeds immediately when non-0

glibc Library: Statistics Gathering

- Statistics structures initialized/updated efficiently
  - Done on thread's first interaction with a lock
  - Statistics tracked per-thread combined near program exit
  - Initialized wait-free
- Tracks:
  - Flat xbegin and xend
  - Time spent on aborted and successful transactions
  - Occurrences of abort codes (including trylock aborts)

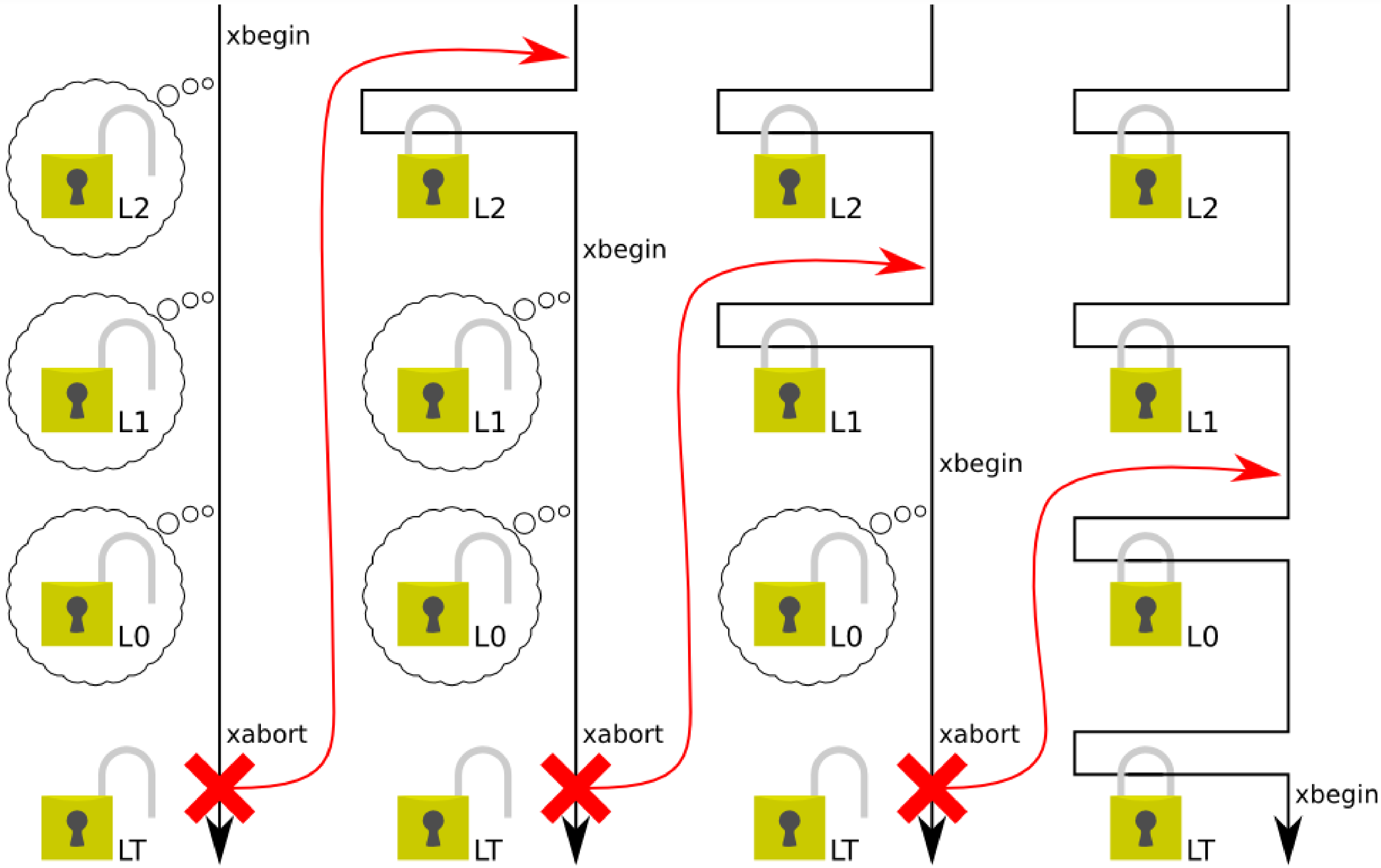
glibc Library: Semantic Differences

- Deadlock introduction and hiding
  - Fine-grained deadlocks may disappear with a global lock
- Communicating critical sections
  - Explicit synchronization may deadlock without locks
- Empty critical sections
  - May impede progress via *global lock semantics*
- Time spent in synchronized sections
  - May be higher for elision than mutexes

Lock Cascade Failure

- glibc associates tries with the lock only
  - Tries are not associated with the thread
  - Elision backoff does not carry between mutexes
- Quadratic amount of work for a linear task
  - Occurs under a reliable abort and multiple transactions
  - Outermost atomic section repeatedly peeled off
- Bounded by:
  - MAX_RTM_NEST_COUNT=7 (*see thesis for detection*)
  - Periodic aborts

Lock Cascade Failure



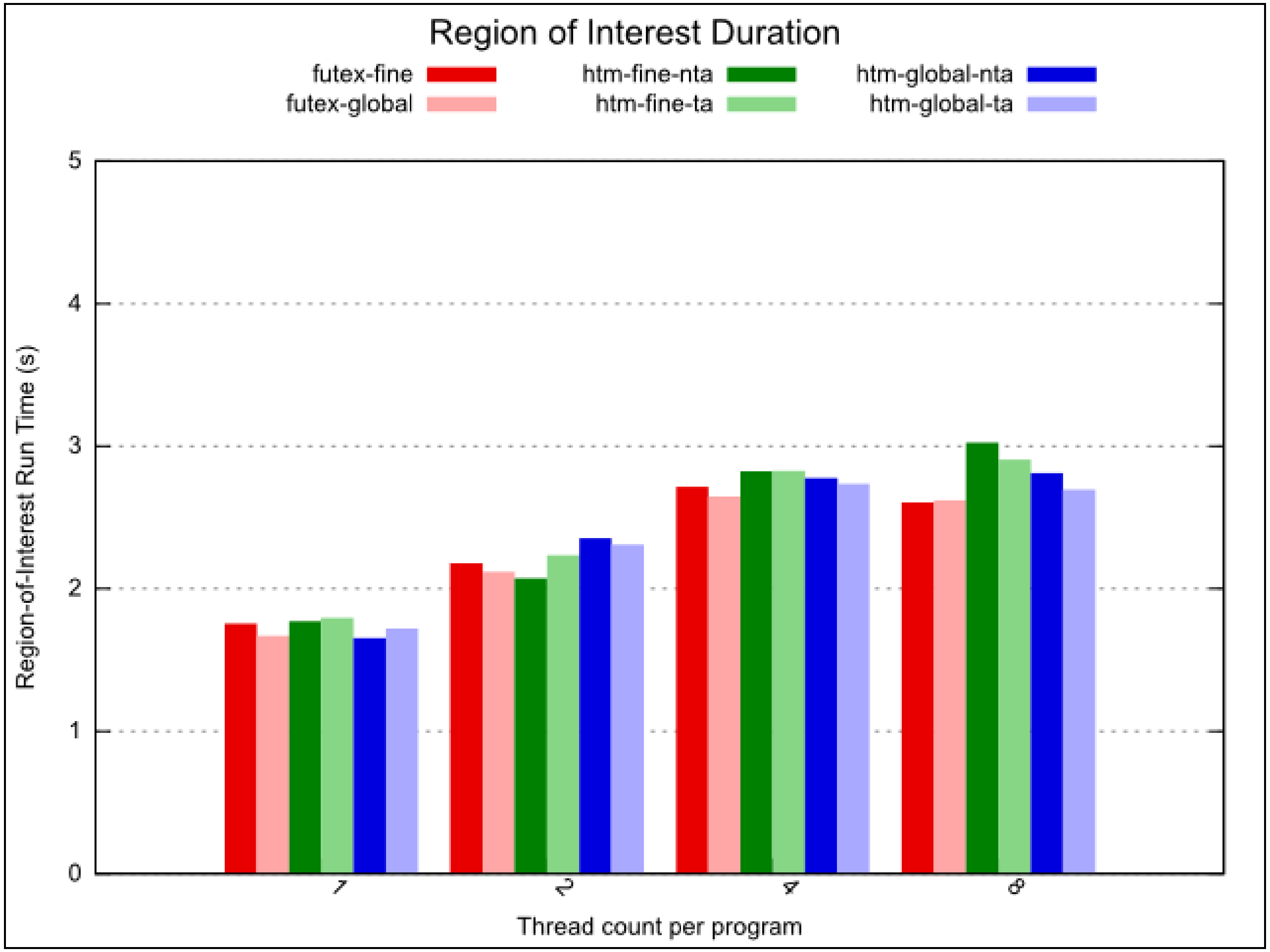
Results: Experimental Setup

- Hardware
  - Haswell 64-bit x86 i7-4770, 3.40GHz
  - **8 Hyper-thread CPUs, 4 cores**, 1 socket, 1 NUMA zone
  - 16GiB memory
  - 32KB L1d, 256KB L2, 8192KB L3 cache
  - **MAX_RTM_NEST_COUNT=7**
- Software
  - glibc version 2.19, compiled with -O2
  - g++ version 4.9.2
  - Ubuntu 14.04 LTS, Linux 3.13.0-63-generic

Results: memcached

- In-memory object cache
  - Capable of distributed caching
  - Meant to relieve processing done by web databases
- Setup
  - memcached version 1.4.24
  - memslap from libmemcached-1.0.18
- Notable synchronization methods
  - Nested trylocks
  - Condition variables
  - Hanging atomic sections

Results: memcached Region-of-Interest

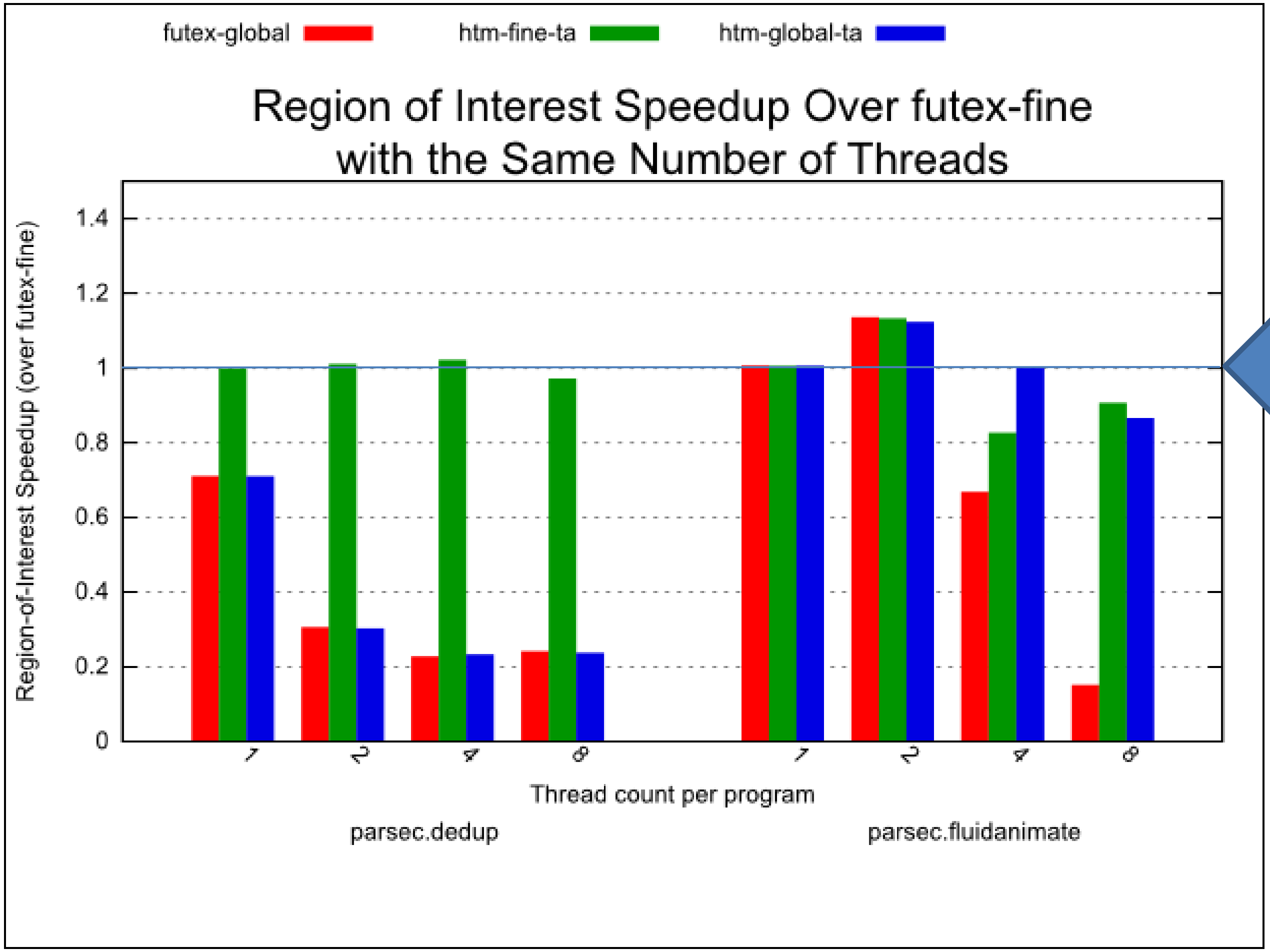


Lower is better

Results: PARSEC and SPLASH-2x

- Suites of parallel programs (22 programs used)
  - PARSEC 3.0: general programs
  - SPLASH-2x : high-performance computing
- According to SPLASH-2x's authors: PARSEC and SPLASH-2 complement each other
  - Diverse cache miss rate
  - Working set size
  - Instruction distribution

Results: PARSEC and SPLASH-2x Region-of-Interest



futex-fine baseline

Higher is better

Results: dedup, fluidanimate and Other Trends

- PARSEC: dedup
  - Slowdown for global futex and global fallback HTM
  - Despite ~½ transactions committing
- PARSEC: fluidanimate
  - Slowdown for global futex, less so for global fallback HTM
  - Significant time spent in committed transactions
- General Trends
  - Very few programs spend significant time in transactions
  - Generally very little change in performance

Conclusion

- Global lock fallback HTM competes with fine-grained locking in a large majority of cases.
- Global locking is largely simplified over fine-grained locking
  - HTM makes it more competitive
- Introduced lock cascade failure
- Provide a method to easily experiment with HTM and global locking in real word applications

Question and Answer

Questions?

Thank You