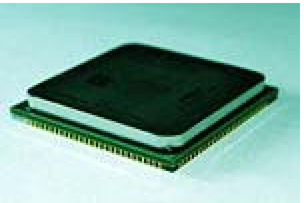
Maximum Benefit from a Minimal HTM

Owen Hofmann, Chris Rossbach, and Emmett Witchel

The University of Texas at Austin

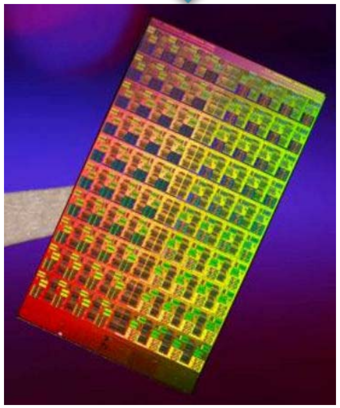
Concurrency is here

- Core count ever increasing
- Parallel programming is difficult
  - Synchronization perilous
  - Performance/complexity
- Many ideas for simpler parallelism
  - TM, Galois, MapReduce



2 core

16 core



80 core

Transactional memory

- Better performance from simple code
  - Change performance/complexity tradeoff
- Replace locking with *memory transactions*
  - Optimistically execute in parallel
  - Track *read/write sets,* roll back on *conflict*
  - $W_A \cap (R_B \cup W_B) \mathrel{!=} \emptyset$
  - *Commit* successful changes

TM ain't easy

- TM must be fast
  - Lose benefits of concurrency
- TM must be unbounded
  - Keeping within size not easy programming model
- TM must be realizable
  - Implementing TM an important first step

TM ain't easy

	Fast	Realizable	Unbounded
Best-effort HTM	✔		

- Version and detect conflicts with existing structures
    - Cache coherence, store buffer

TM ain't easy

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	

- Version and detect conflicts with existing structures
  - Cache coherence, store buffer
- Simple modifications to processor
  - Very realizable (stay tuned for Sun Rock)

TM ain't easy

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	✘

- Version and detect conflicts with existing structures
  - Cache coherence, store buffer
- Simple modifications to processor
  - Very realizable (stay tuned for Sun Rock)
- Resource-limited
  - Cache size/associativity, store buffer size

TM ain't easy

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	✖
STM	✖	✔	✔

- Software endlessly flexible
  - Transaction size limited only by virtual memory
- Slow
  - Instrument most memory references

TM ain't easy

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	✖
STM	✖	✔	✔
Unbounded HTM	✔	✖	✔

- Versioning unbounded data in hardware is difficult
  - Unlikely to be implemented

TM ain't easy

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	✖
STM	✖	✔	✔
Unbounded HTM	✔	✖	✔
Hybrid TM	~	~	✔

- Tight marriage of hardware and software
  - Disadvantages of both?

Back to basics

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	✖

- Cache-based HTM
  - Speculative updates in L1
  - Augment cache line with transactional state
  - Detect conflicts via cache coherence
- Operations outside transactions can conflict
  - *Asymmetric conflict*
  - Detected and handled in *strong isolation*

Back to basics

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	✖

- Transactions bounded by cache
  - *Overflow* because of size or associativity
  - Restart, return reason
- Not all operations supported
  - Transactions cannot perform I/O

Back to basics

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	✖

- Transactions bounded by cache
  - **Software finds another way**
- Not all operations supported
  - **Software finds another way**

Maximum benefit

	Fast	Realizable	Unbounded
Best-effort HTM	✔	✔	✔

- Creative software and ISA makes best-effort unbounded
- TxLinux
  - Better performance from simpler synchronization
- Transaction ordering
  - Make best-effort unbounded

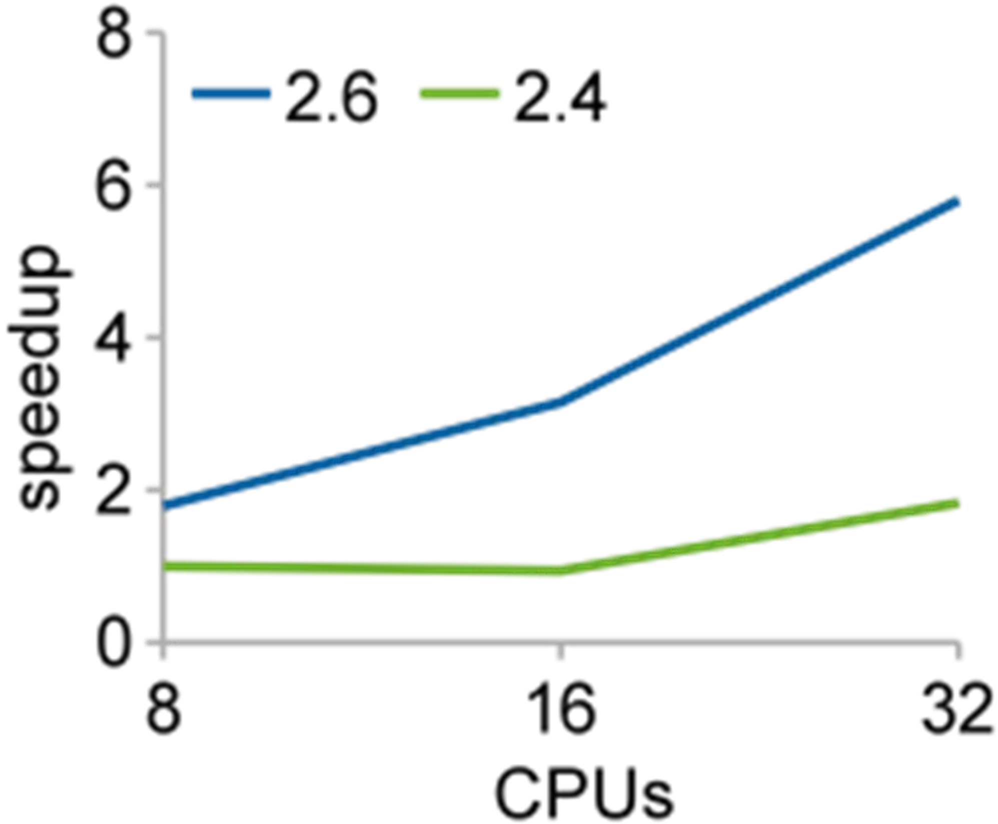
Linux: HTM proving ground

- Large, complex application(s)
  - With different synchronization
- Jan. 2001: Linux 2.4
  - 5 types of synchronization
  - ~8,000 dynamic spinlocks
  - Heavy use of Big Kernel Lock
- Dec. 2003: Linux 2.6
  - 8 types of synchronization
  - ~640,000 dynamic spinlocks
  - Restricted Big Kernel Lock use

Linux: HTM proving ground

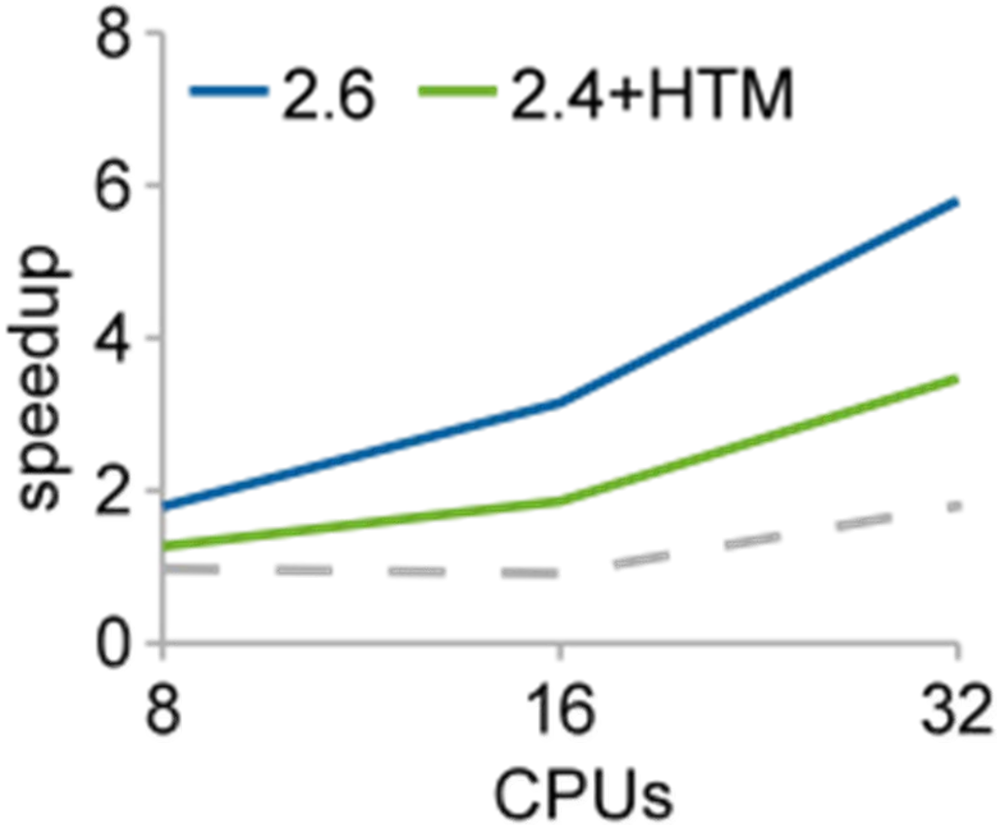
- Large, complex application
  - With evolutionary snapshots
- Linux 2.4
  - Simple, coarse synchronization
- Linux 2.6
  - Complex, fine-grained synchronization

Linux: HTM proving ground



Modified Andrew Benchmark

Linux: HTM proving ground



Modified Andrew Benchmark

HTM can help 2.4

- Software must back up hardware
  - Use locks
- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary

```
acquire_lock(lock)

release_lock(lock)
```

HTM can help 2.4

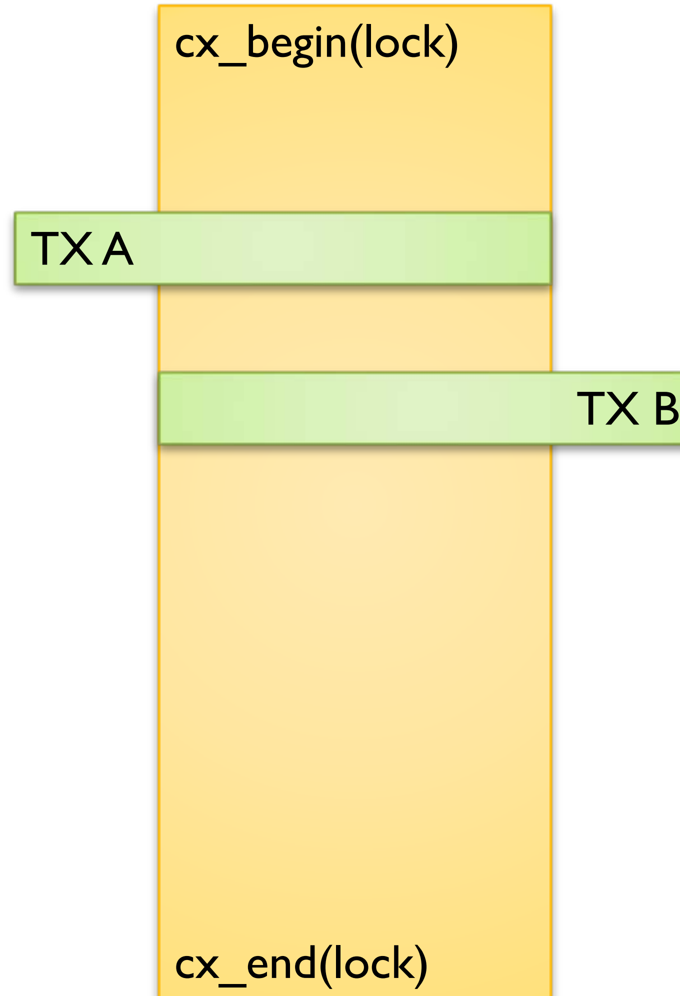
- Software must back up hardware
  - Use locks

- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary

```
cx_begin(lock)



cx_end(lock)
```

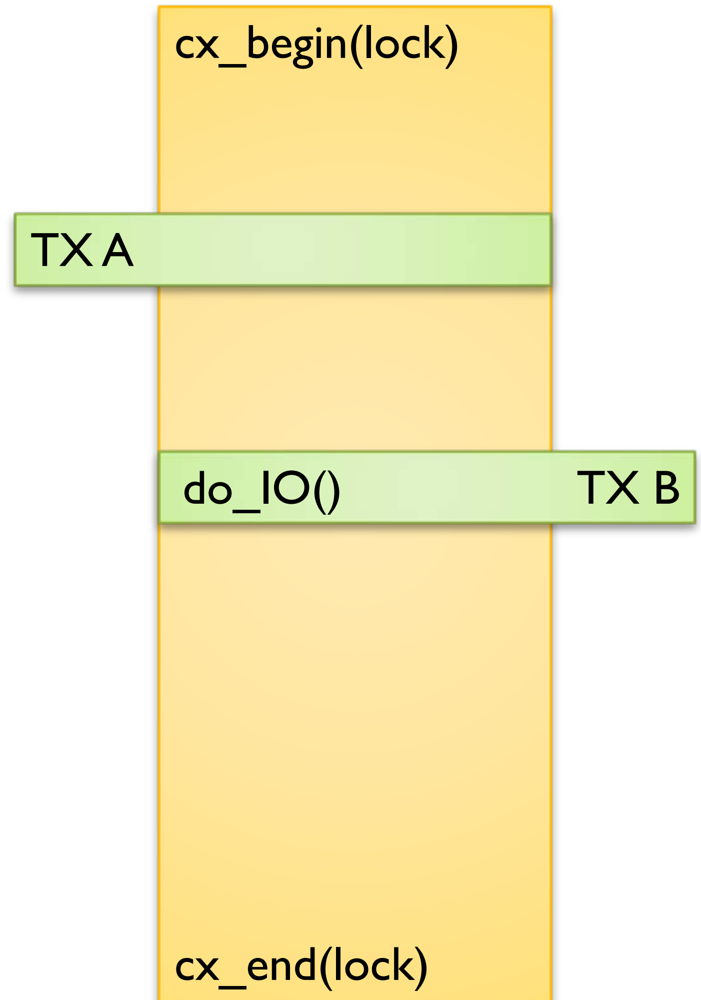
HTM can help 2.4

- Software must back up hardware
    - Use locks
- Cooperative transactional primitives
    - Replace locking function
    - Execute speculatively, concurrently in HTM
    - Tolerate overflow, I/O
    - Restart, (fairly) use locking if necessary



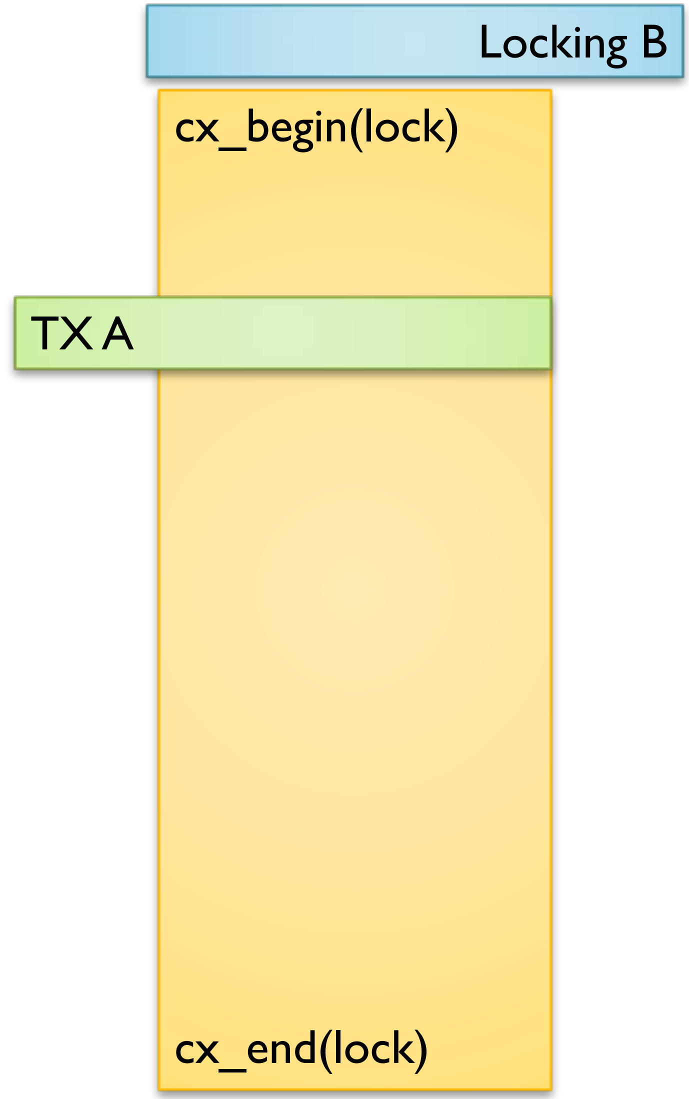
HTM can help 2.4

- Software must back up hardware
  - Use locks
- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary



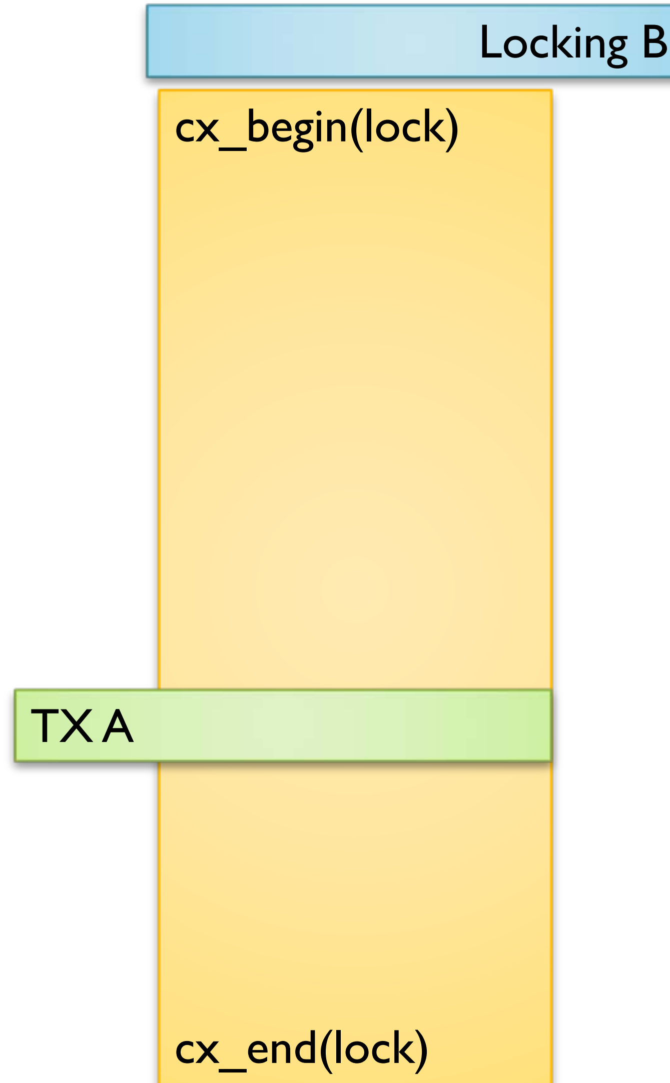
HTM can help 2.4

- Software must back up hardware
  - Use locks
- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary



HTM can help 2.4

- Software must back up hardware
  - Use locks
- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary

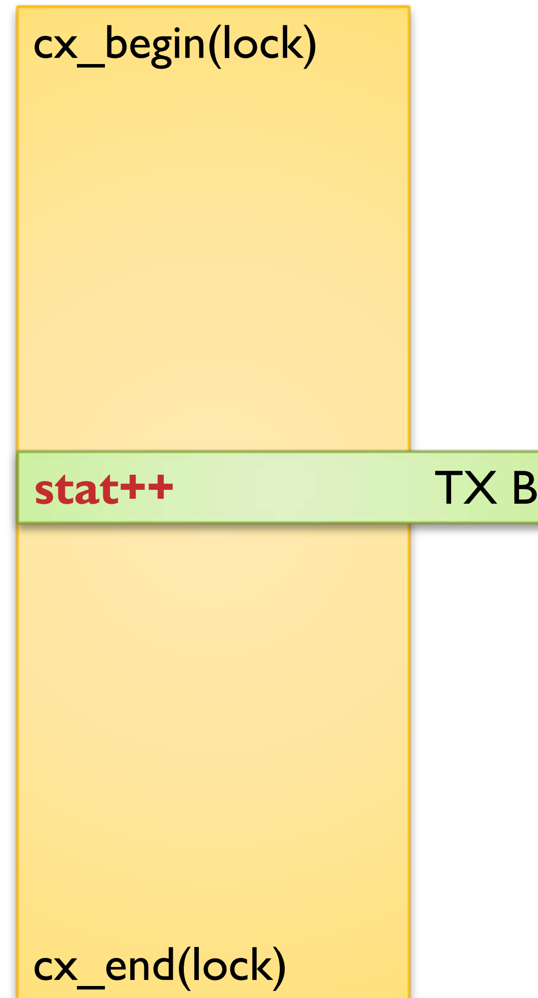


Adding HTM

- Spinlocks: good fit for best-effort transactions
  - Short, performance-critical synchronization
  - cxspinlocks (SOSP '07)
- 2.4 needs cooperative transactional mutexes
  - Must support blocking
  - Complicated interactions with BKL
  - cxmutex
  - Must modify wakeup behavior

Adding HTM, cont.

- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code
- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes
- Linux 2.4 ➔ TxLinux 2.4
  - Change synchronization, not use



Adding HTM, cont.

- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code
- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes
- Linux 2.4 ➔ TxLinux 2.4
  - Change synchronization, not use

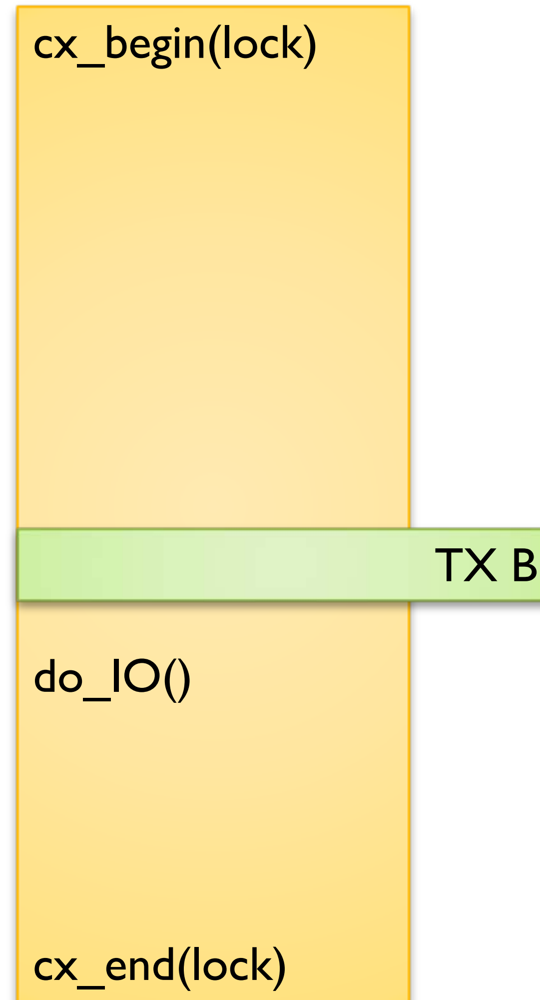
cx_begin(lock)

stat[CPU]++ TX B

cx_end(lock)

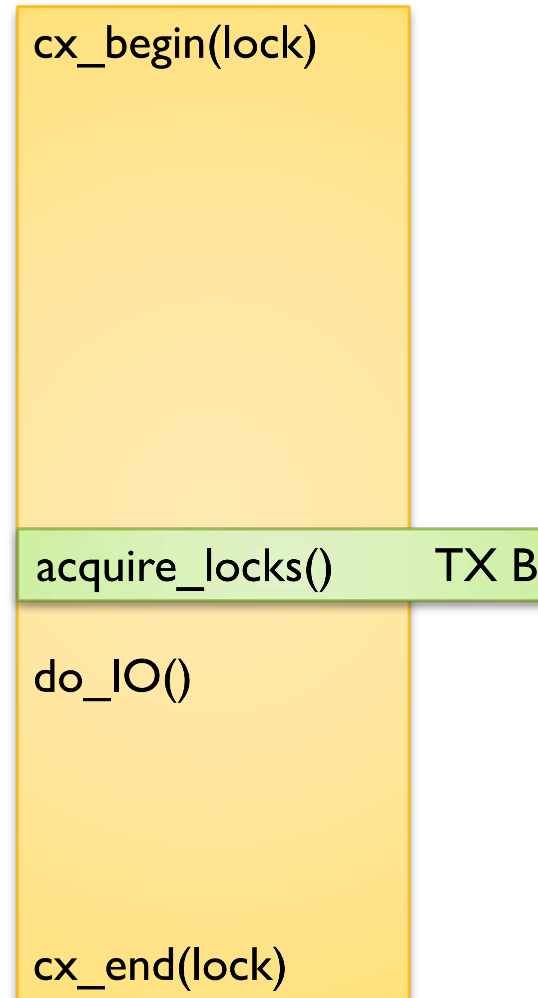
Adding HTM, cont.

- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code
- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes
- Linux 2.4 ➔ TxLinux 2.4
  - Change synchronization, not use



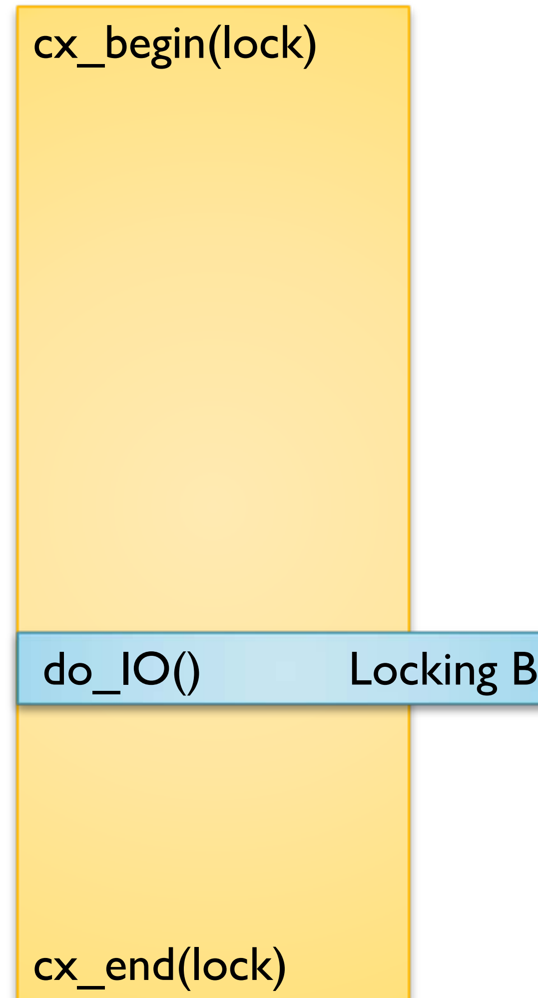
Adding HTM, cont.

- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code
- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes
- Linux 2.4 ➔ TxLinux 2.4
  - Change synchronization, not use



Adding HTM, cont.

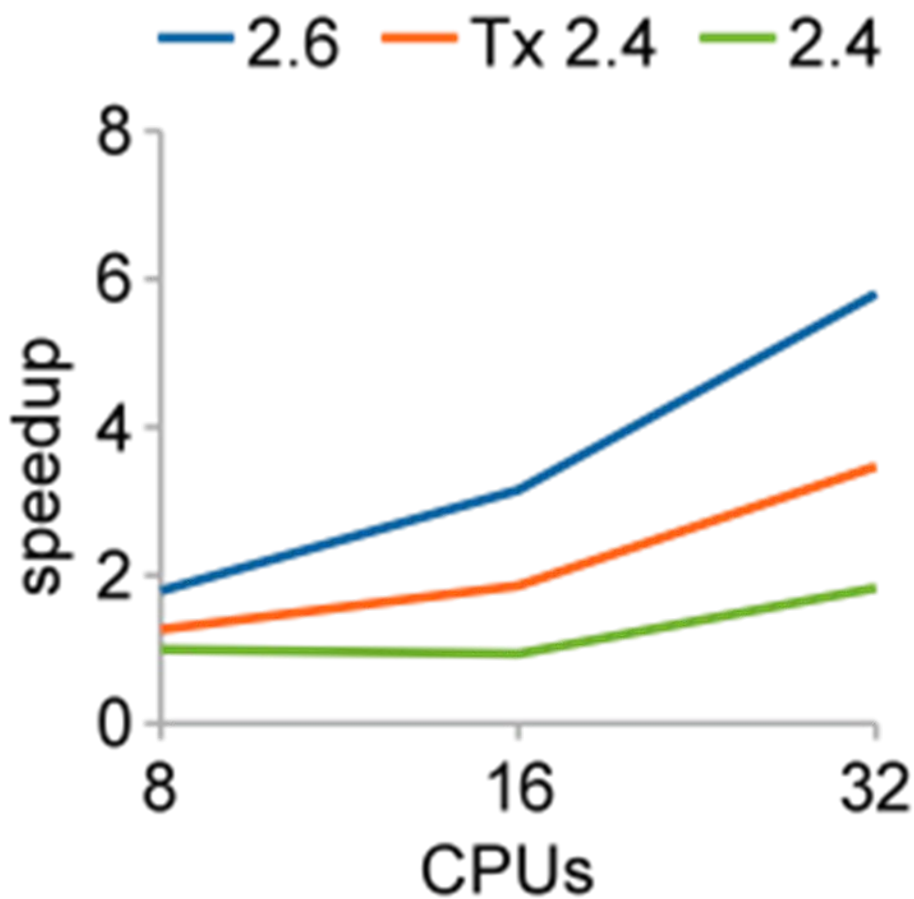
- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code
- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes
- Linux 2.4 ➔ TxLinux 2.4
  - Change synchronization, not use



Evaluating TxLinux

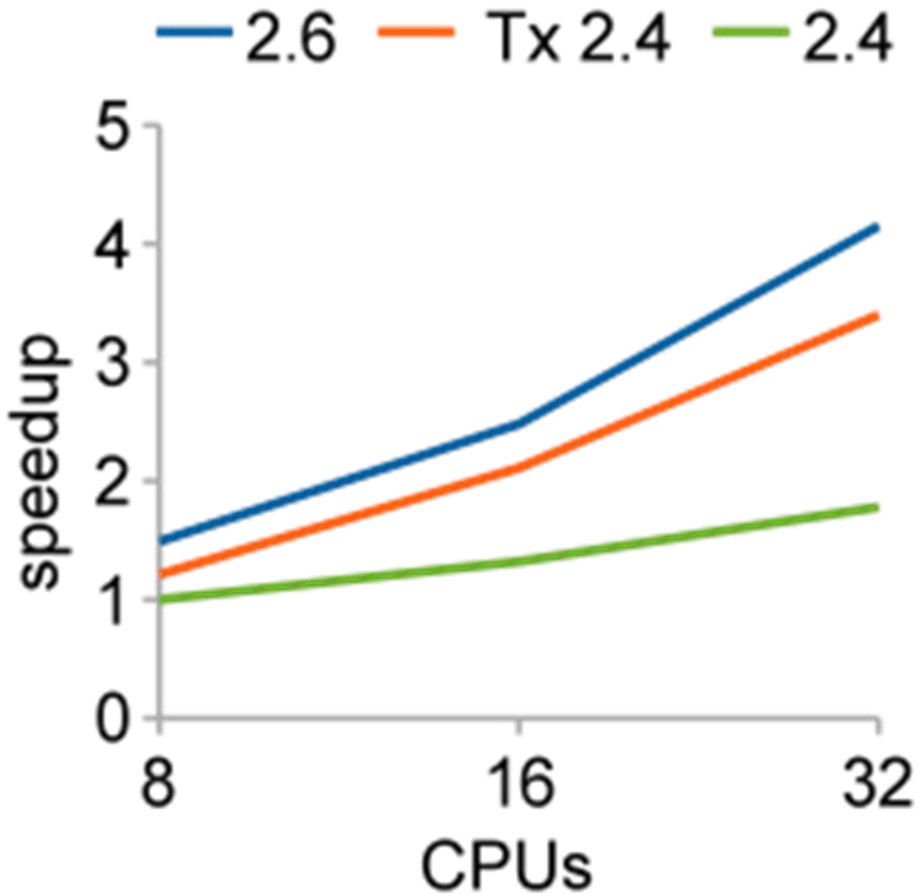
- MAB
  - Modified Andrew Benchmark
- dpunish
  - Stress dcache synchronization
- find
  - Parallel find + grep
- config
  - Parallel software package configure
- pmake
  - Parallel make

Evaluation: MAB



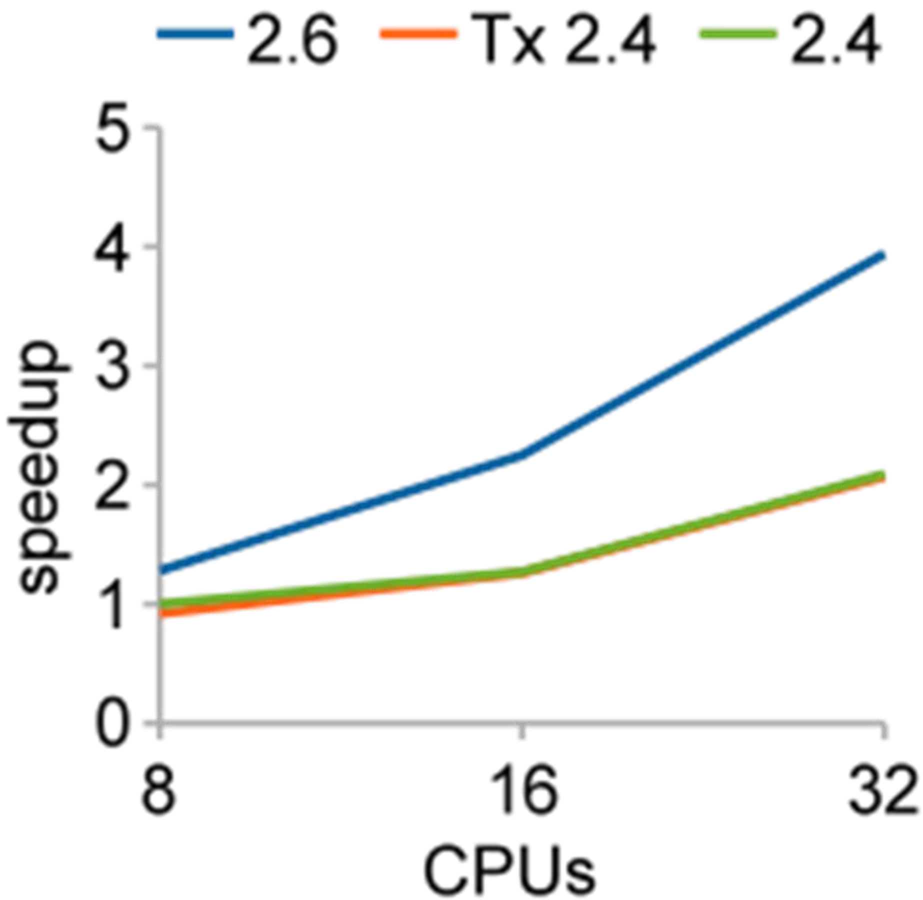
- 2.4 wastes 63% kernel time synchronizing

Evaluation: dpunish



- 2.4 wastes 57% kernel time synchronizing

Evaluation: config



- 2.4 wastes 30% kernel time synchronizing

From kernel to user

- Best-effort HTM means simpler locking code
  - Good programming model for kernel
  - Fall back on locking when necessary
  - Still permits concurrency

- HTM promises *transactions*
  - Good model for user
  - Need software synchronization fallback
  - Don’t want to expose to user
  - Want concurrency

Software, save me!

- HTM falls back on *software transactions*
  - Global lock
  - STM
- Concurrency
  - Conflict detection
  - HTM workset in cache
  - STM workset in memory
  - Global lock – no workset
- Communicate between disjoint SW and HW
  - No shared data structures

Hardware, save me!

- HTM has *strong isolation*
  - Detect conflicts with software
  - Restart hardware transaction
  - Only if hardware already has value in read/write set

- *Transaction ordering*
  - Commit protocol for hardware
  - Wait for concurrent software TX
  - Resolve inconsistencies
  - Hardware/OS contains bad side effects

Transaction ordering

char* r

int idx

Transaction ordering

char* r | int idx

Transaction A

```
begin_transaction()

r[idx] = 0xFF

end_transaction()
```

Transaction B

```
begin_transaction()

r = new_array

idx = new_idx

end_transaction()
```

- Invariant: `idx` is valid for `r`

Transaction ordering

`char* r` `int idx`

Transaction A

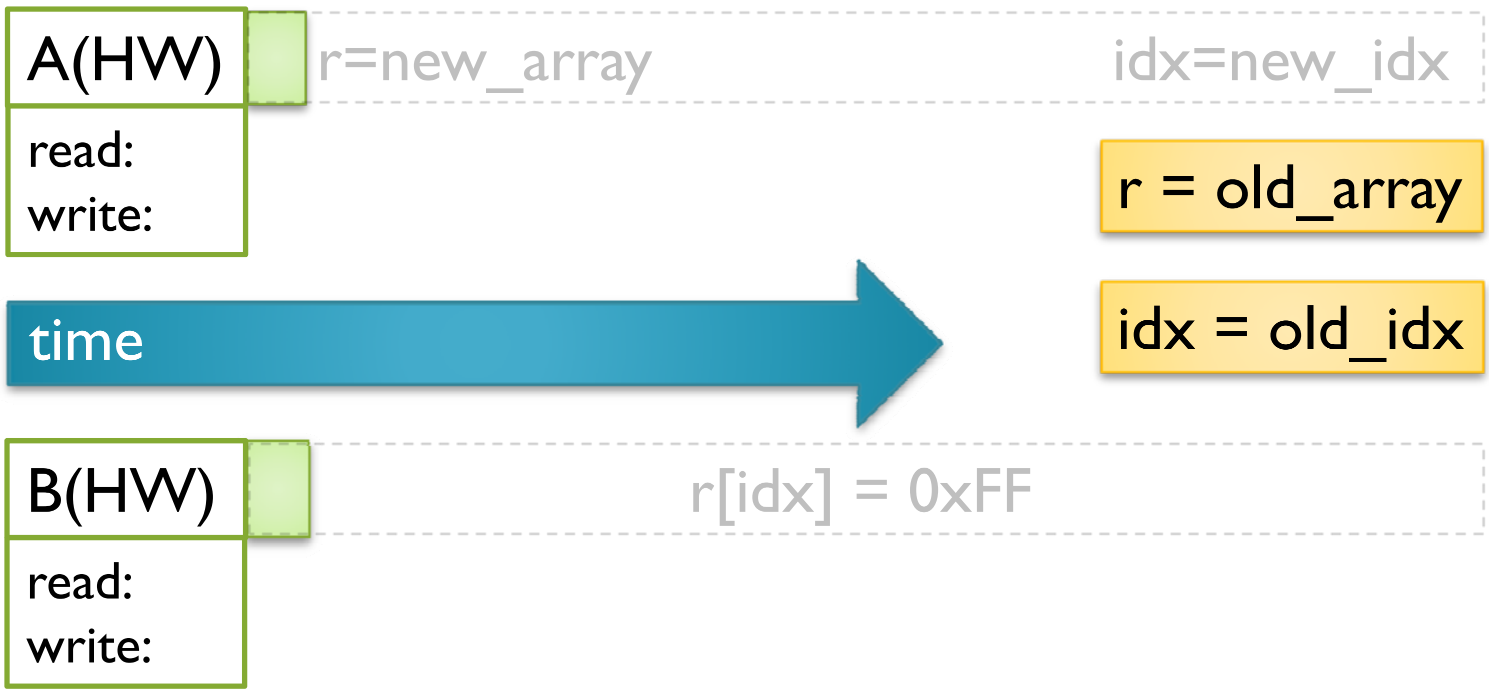
```
begin_transaction()

r[idx] = 0xFF

end_transaction()
```

Transaction B

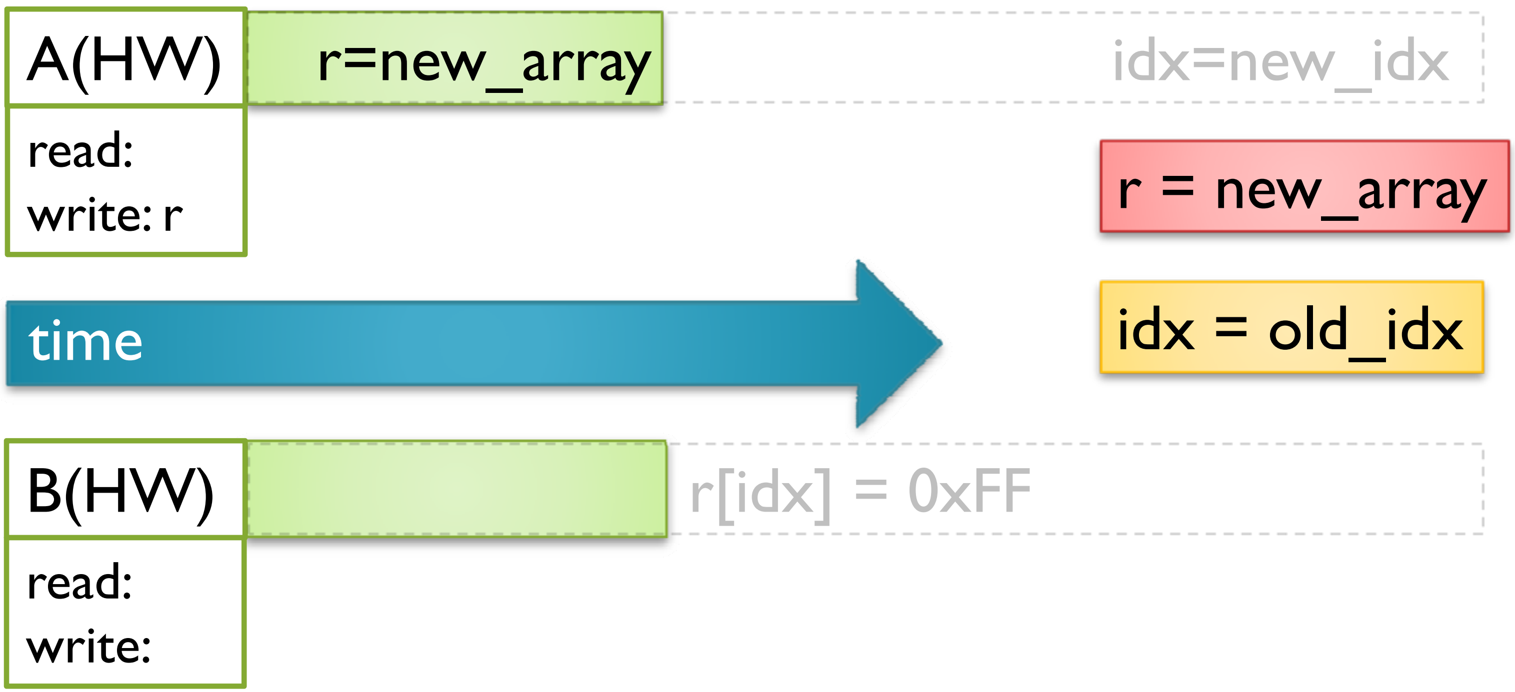
```
begin_transaction()

r = new_array

idx = new_idx

end_transaction()
```

- Invariant: `idx` is valid for `r`
- *Inconsistent* read causes bad data write

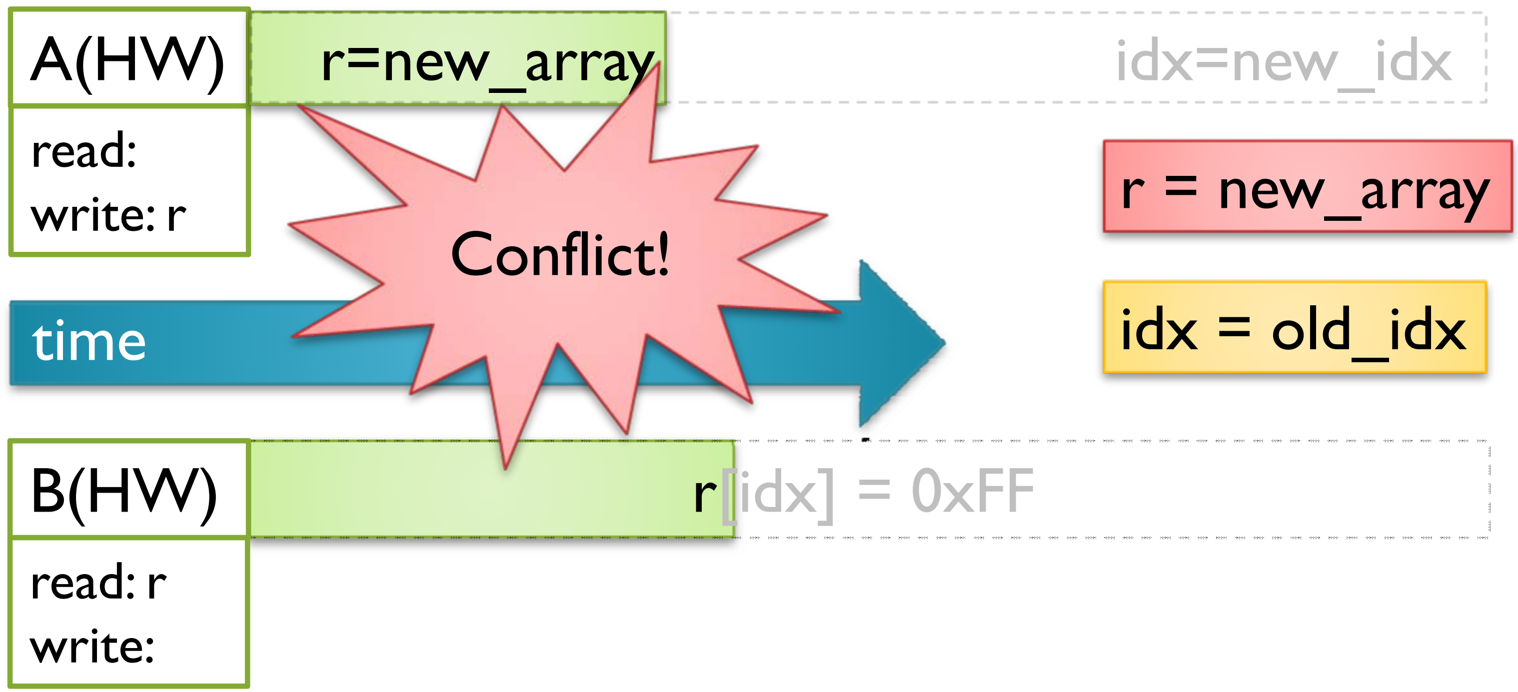
Transaction ordering



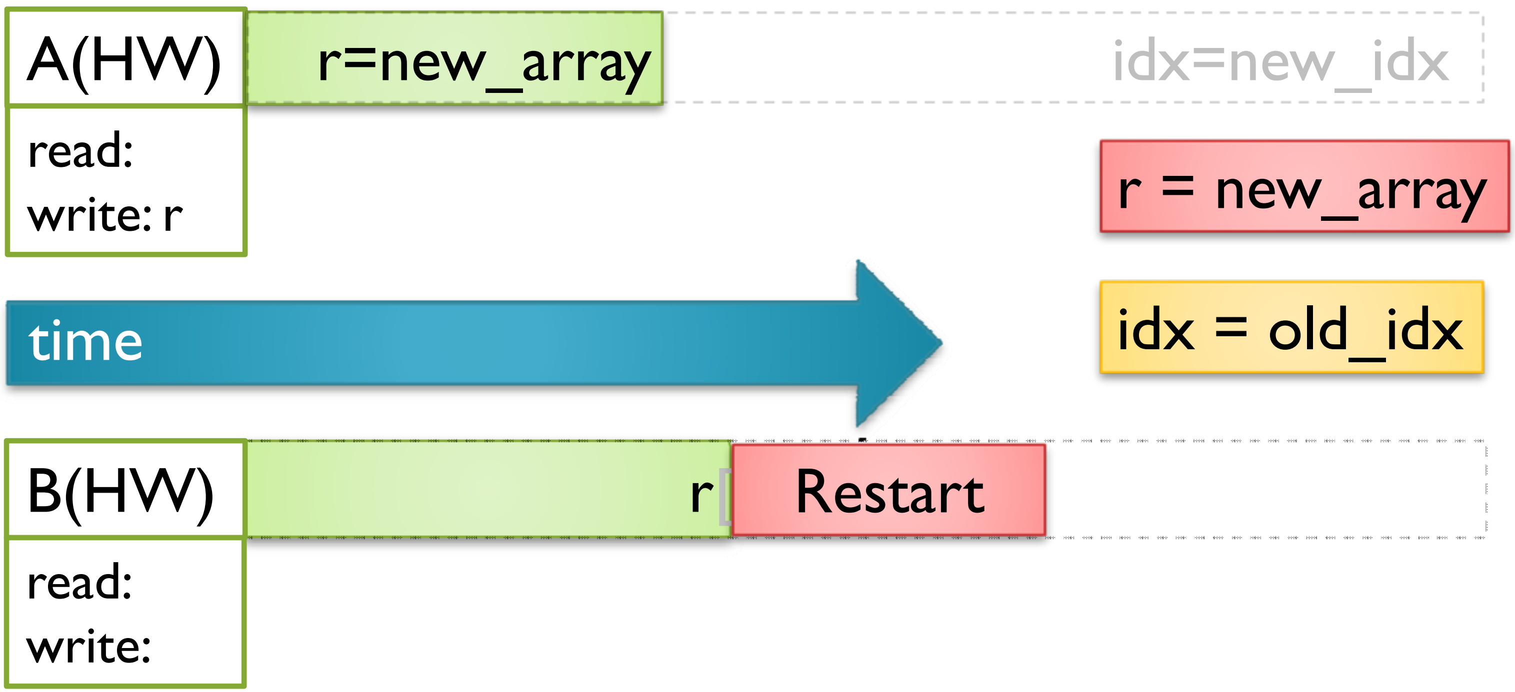
Transaction ordering



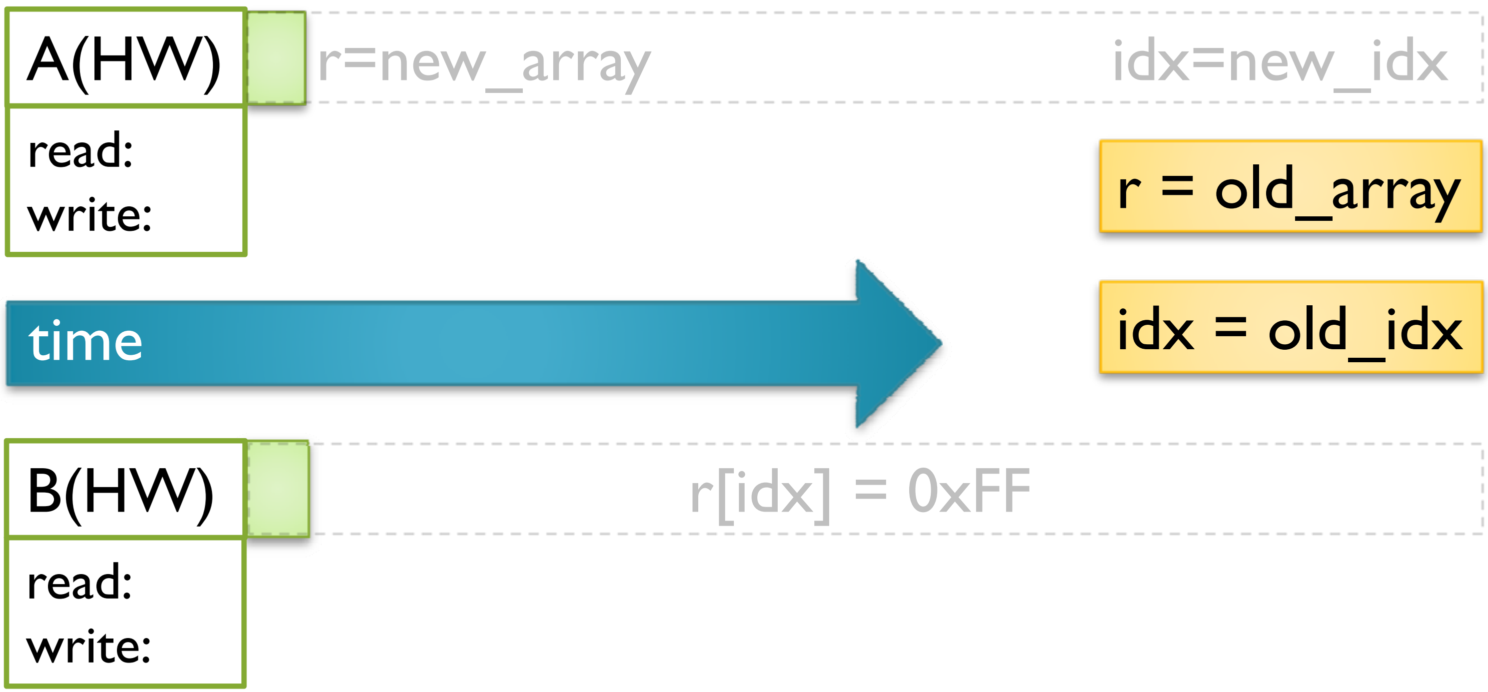
Transaction ordering



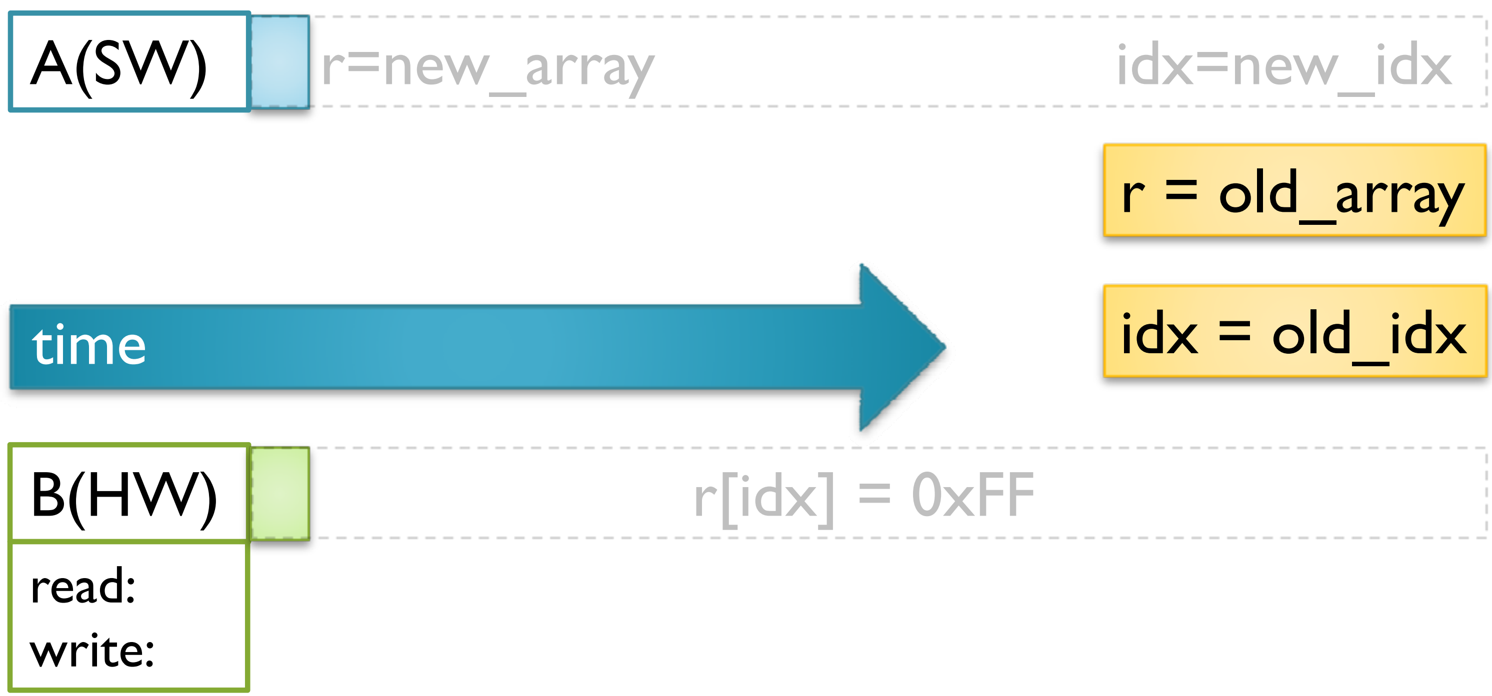
Transaction ordering



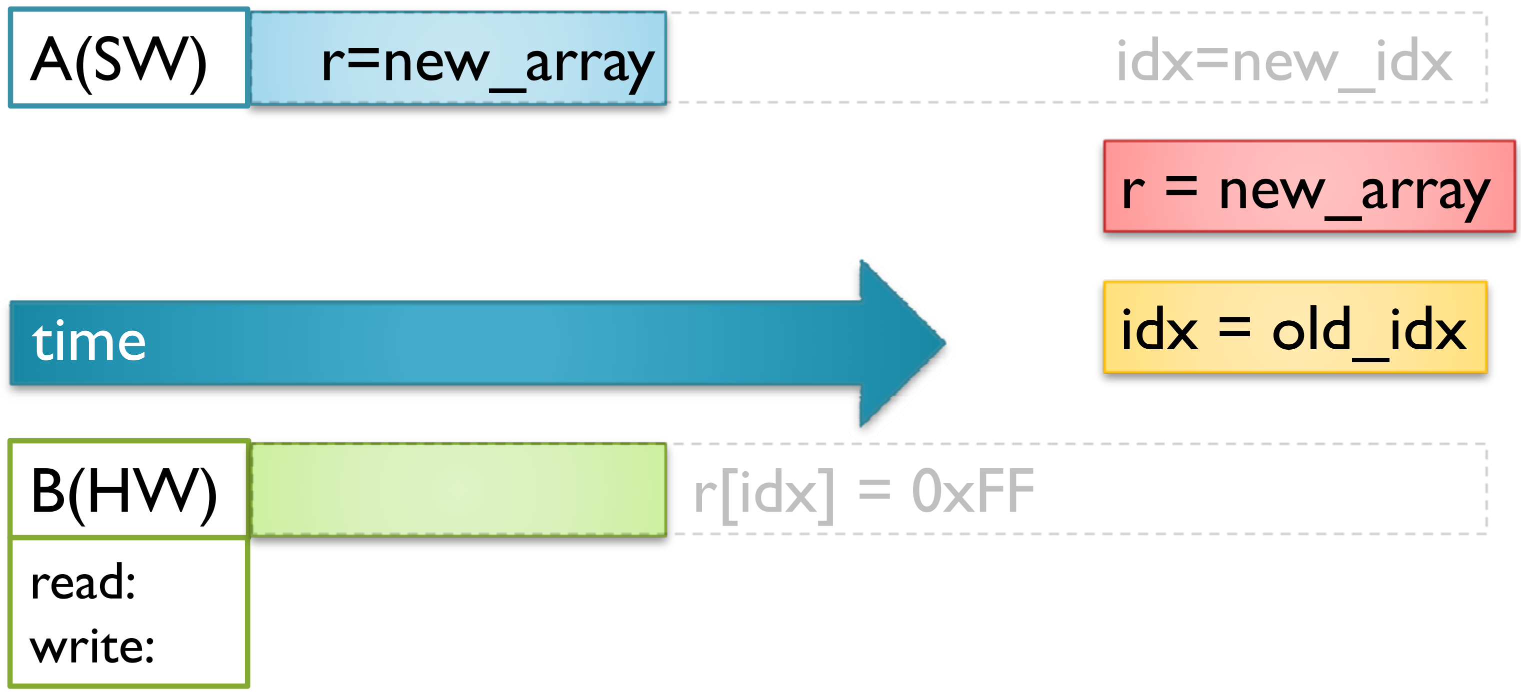
Transaction ordering



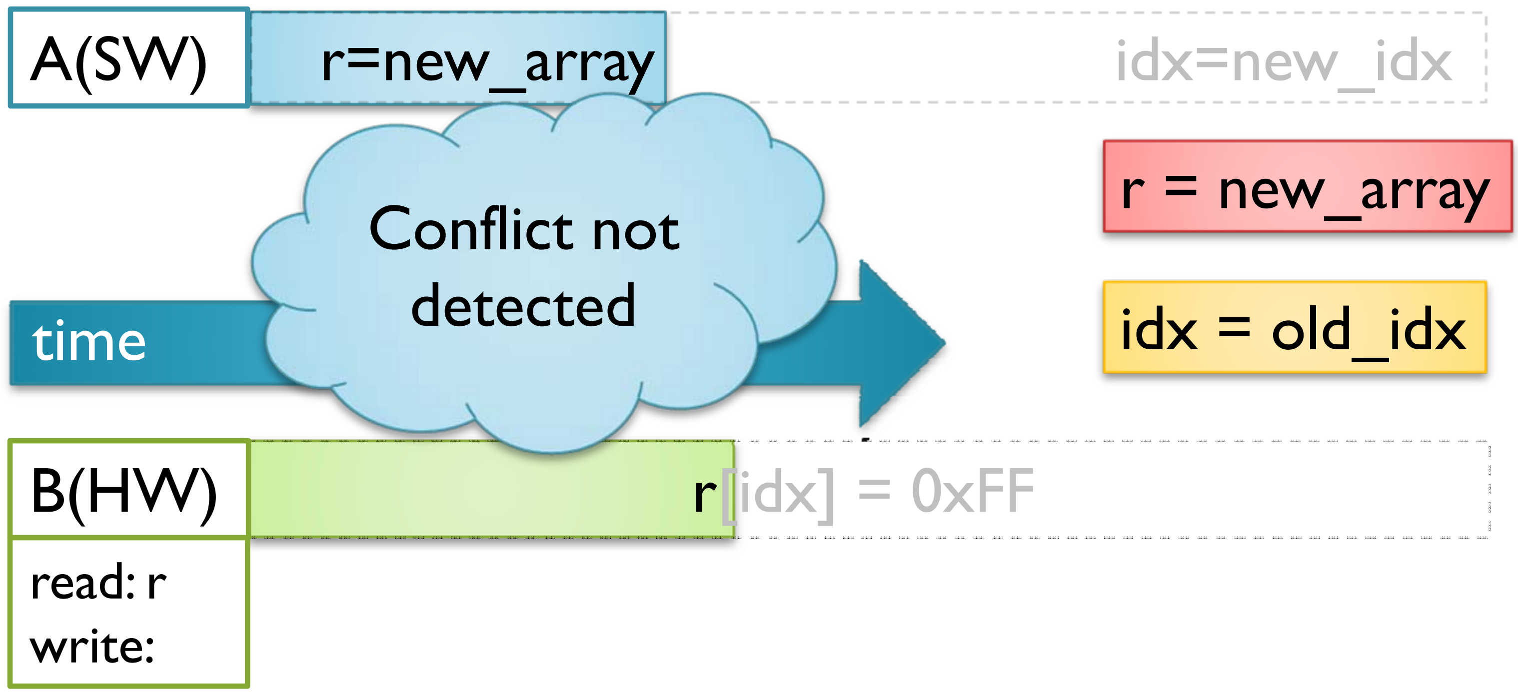
Transaction ordering



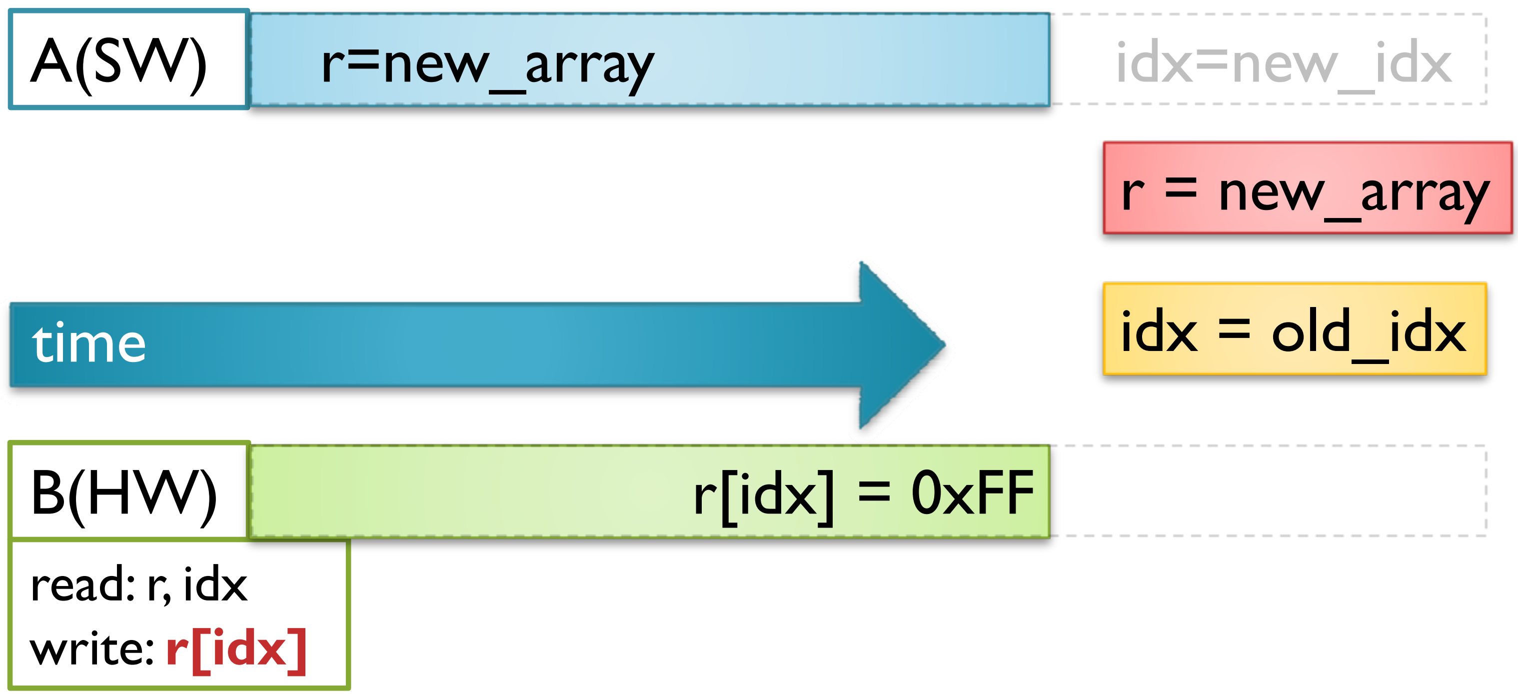
Transaction ordering



Transaction ordering

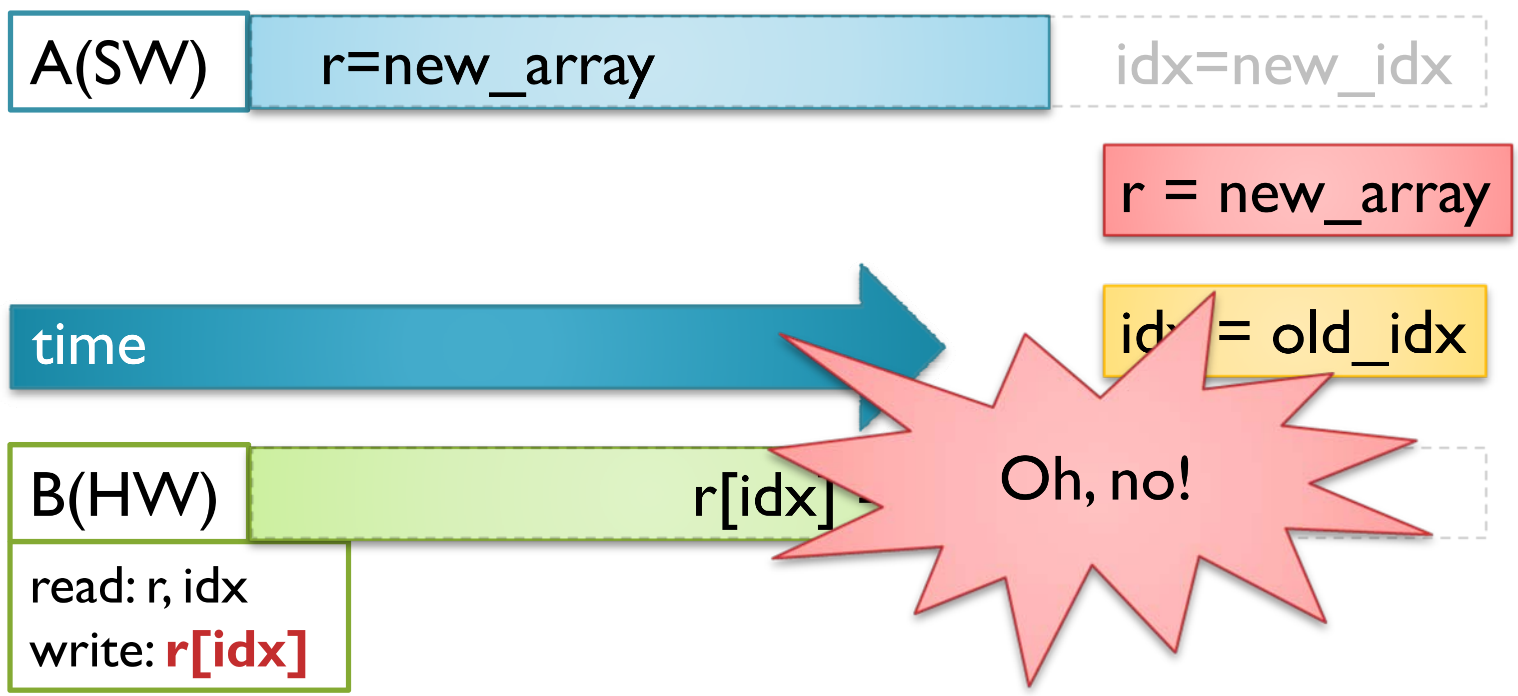


Transaction ordering



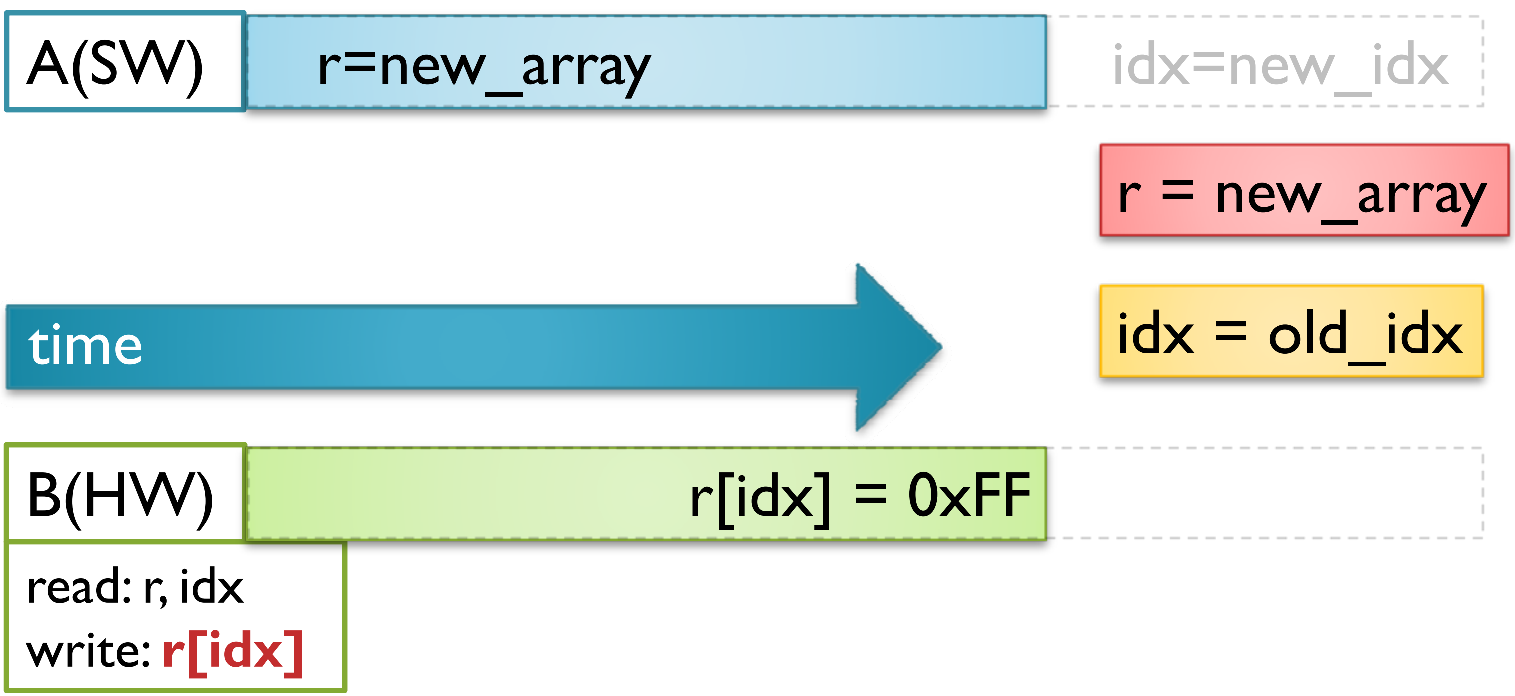
- new_array[old_idx] = 0xFF

Transaction ordering



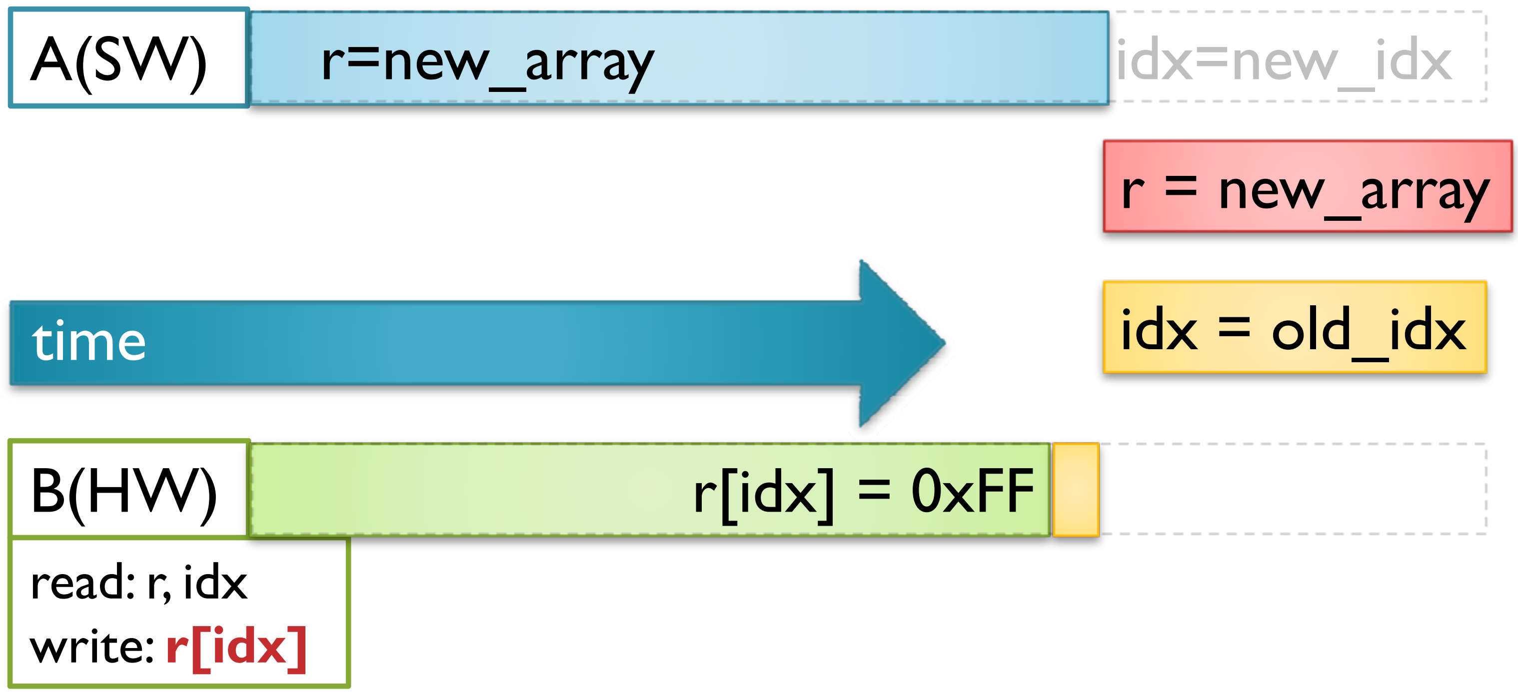
- new_array[old_idx] = 0xFF

Transaction ordering



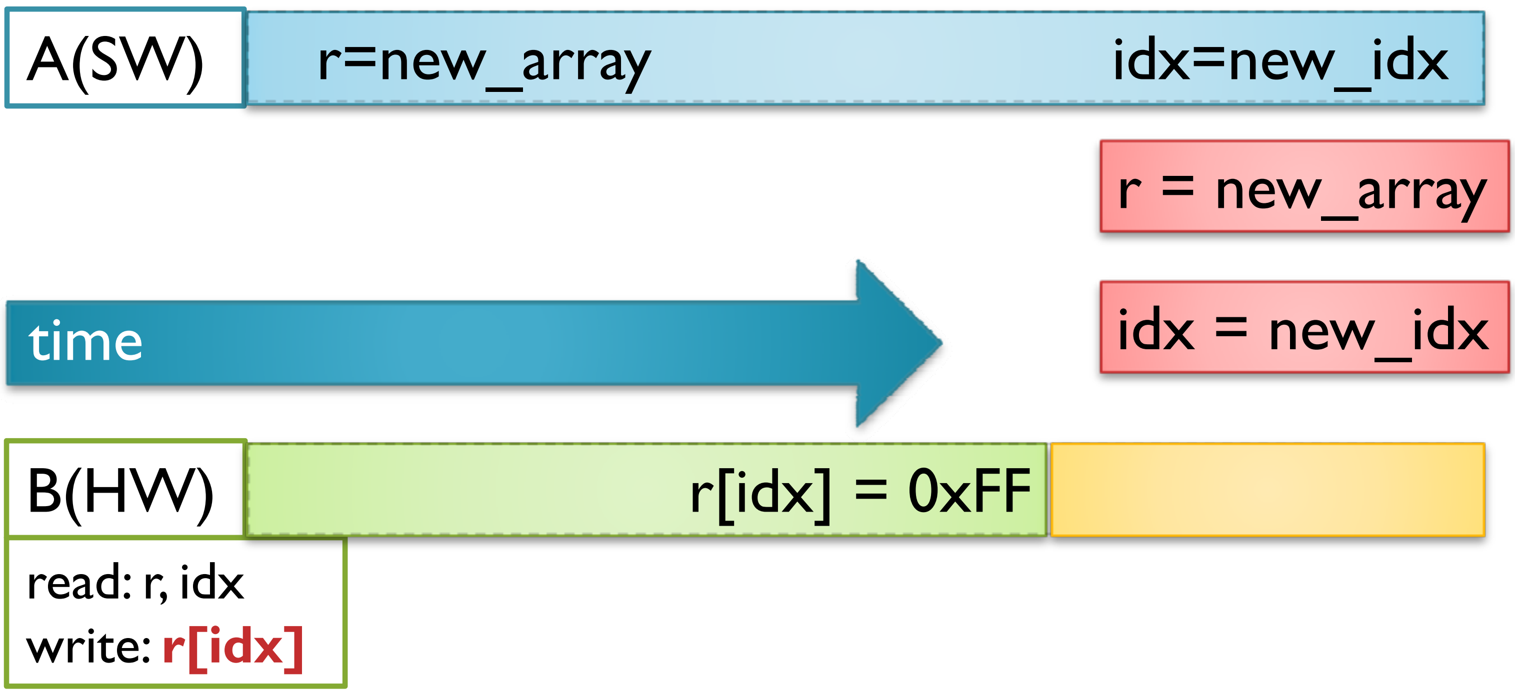
- Hardware contains effects of B
    - Unless B commits

Transaction ordering



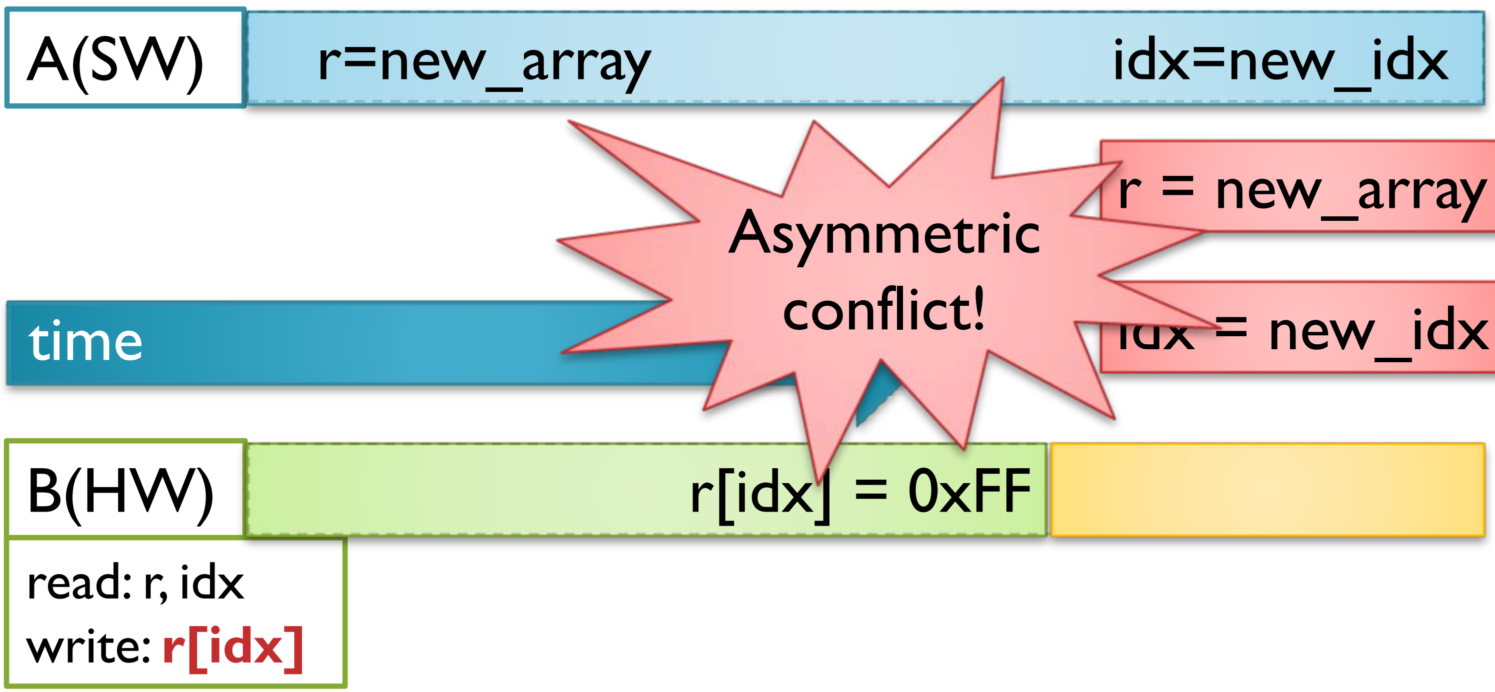
- Hardware contains effects of B
  - Unless B commits

Transaction ordering



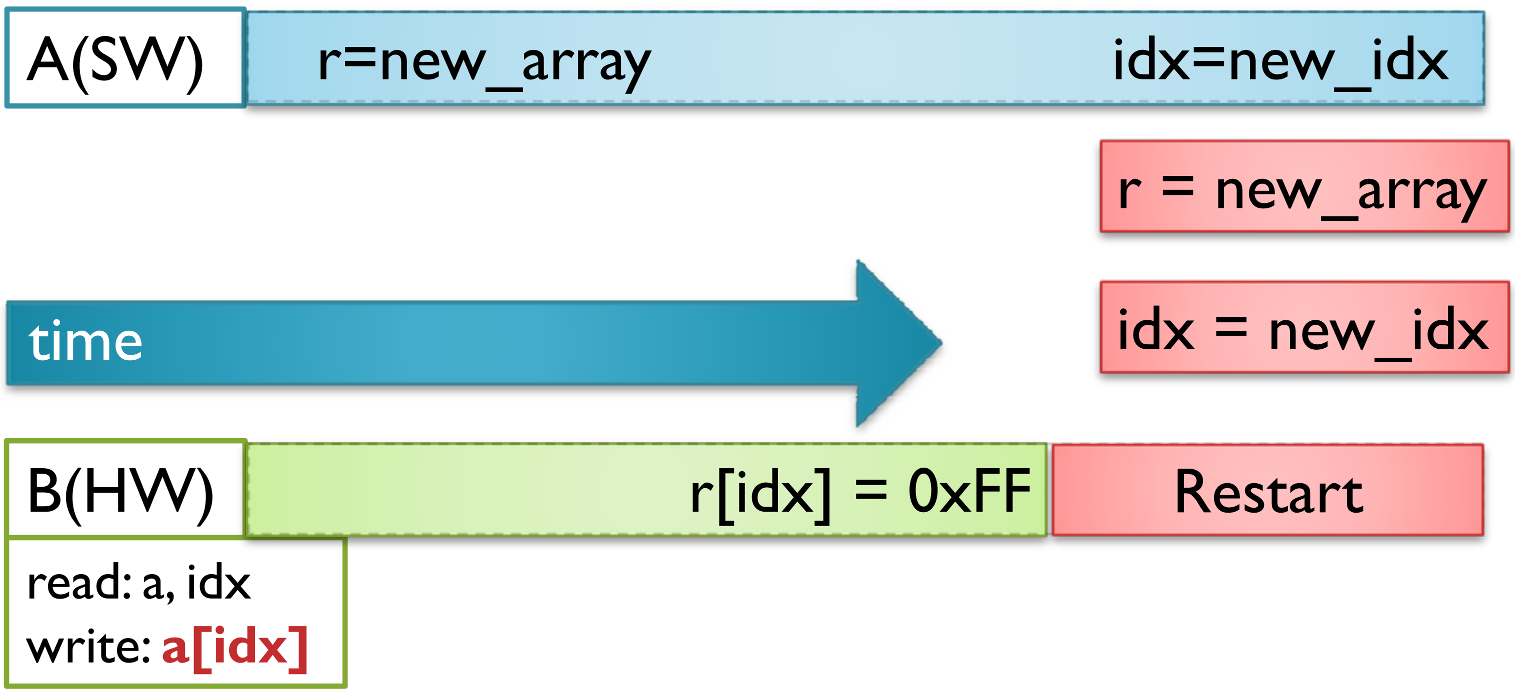
- Hardware contains effects of B
  - Unless B commits

Transaction ordering



- Hardware contains effects of B
  - Unless B commits

Transaction ordering



- Hardware contains effects of B
  - Unless B commits

Software + hardware mechanisms

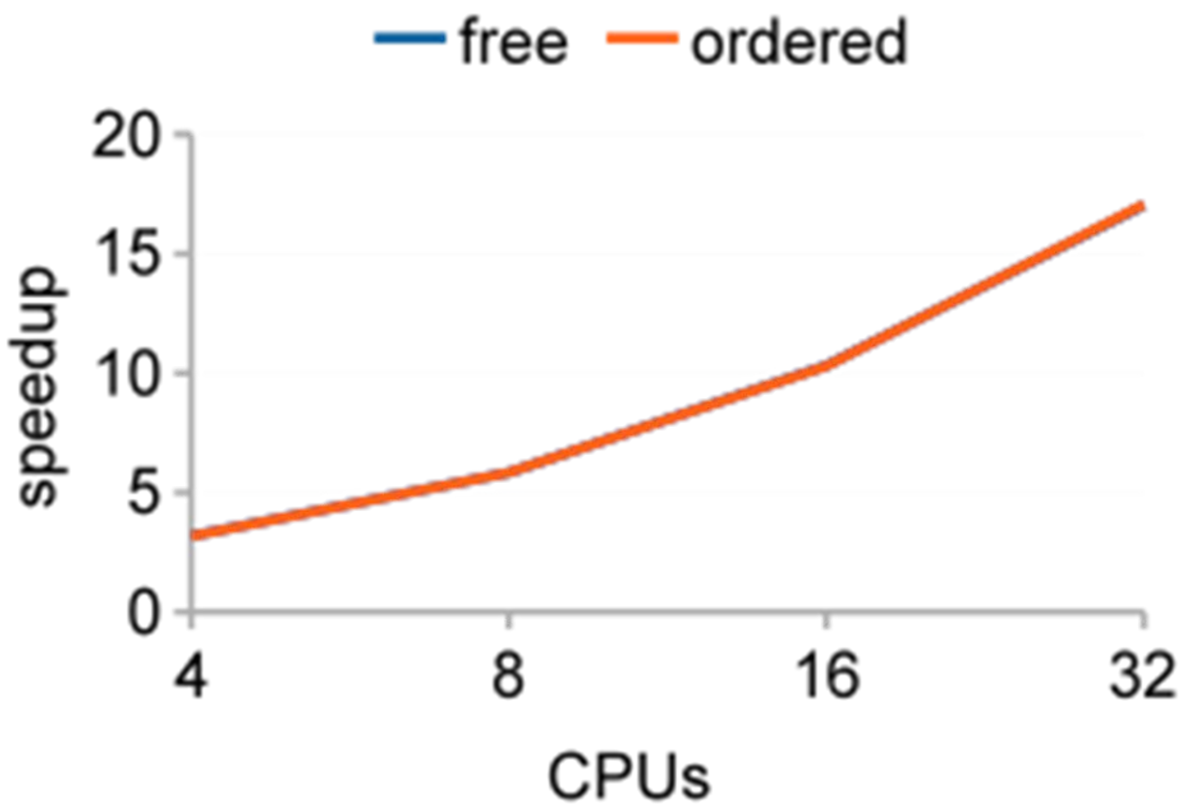
- Commit protocol
  - Hardware commit waits for any current software TX
  - Implemented as sequence lock
- Operating system
  - Inconsistent data can cause spurious fault
  - Resolve faults by TX restart
- Hardware
  - Even inconsistent TX must commit correctly
  - Pass commit protocol address to `transaction_begin()`

Transaction ordering

- Safe, concurrent hardware and software
- Evaluated on STAMP benchmarks
  - ssca2 – graph kernels
  - vacation – reservation system
    - High-contention
    - Low-contention
  - yada – Delauney mesh refinement

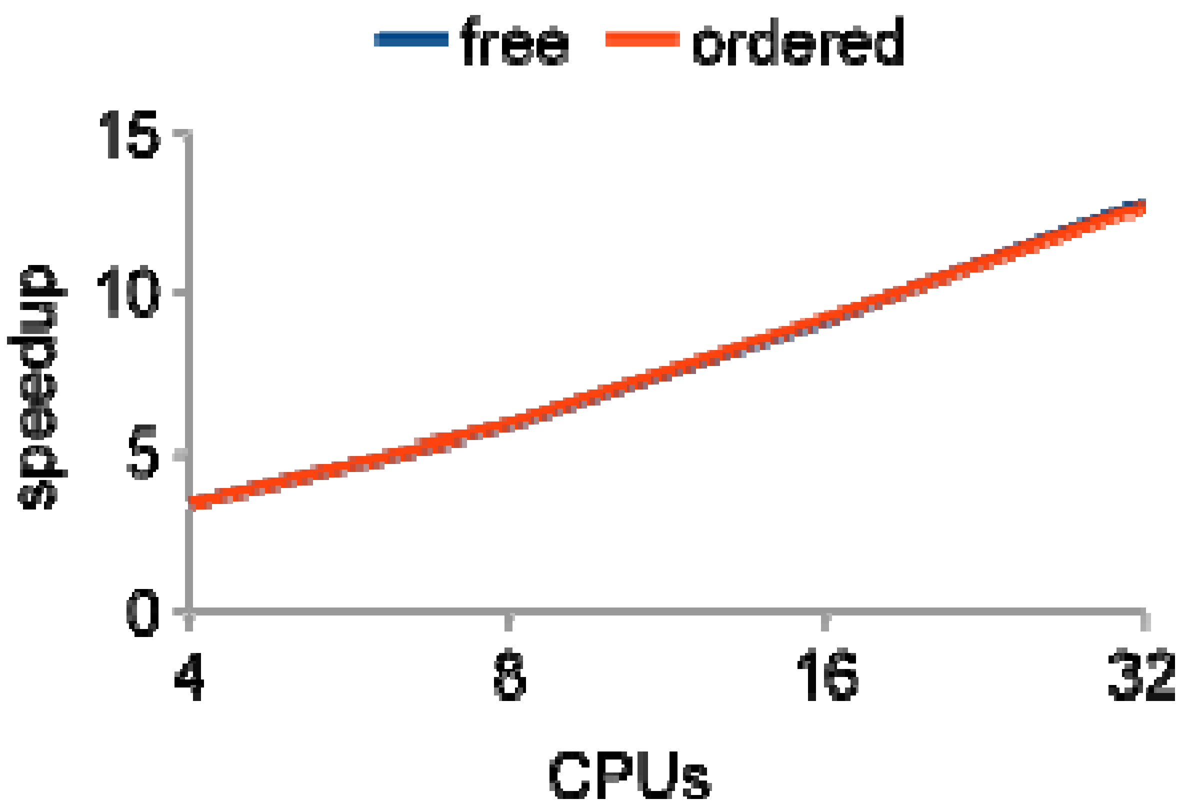
- Best-effort + Single Global Lock STM (ordered)
- Idealized HTM (free)

Evaluation: ssca2



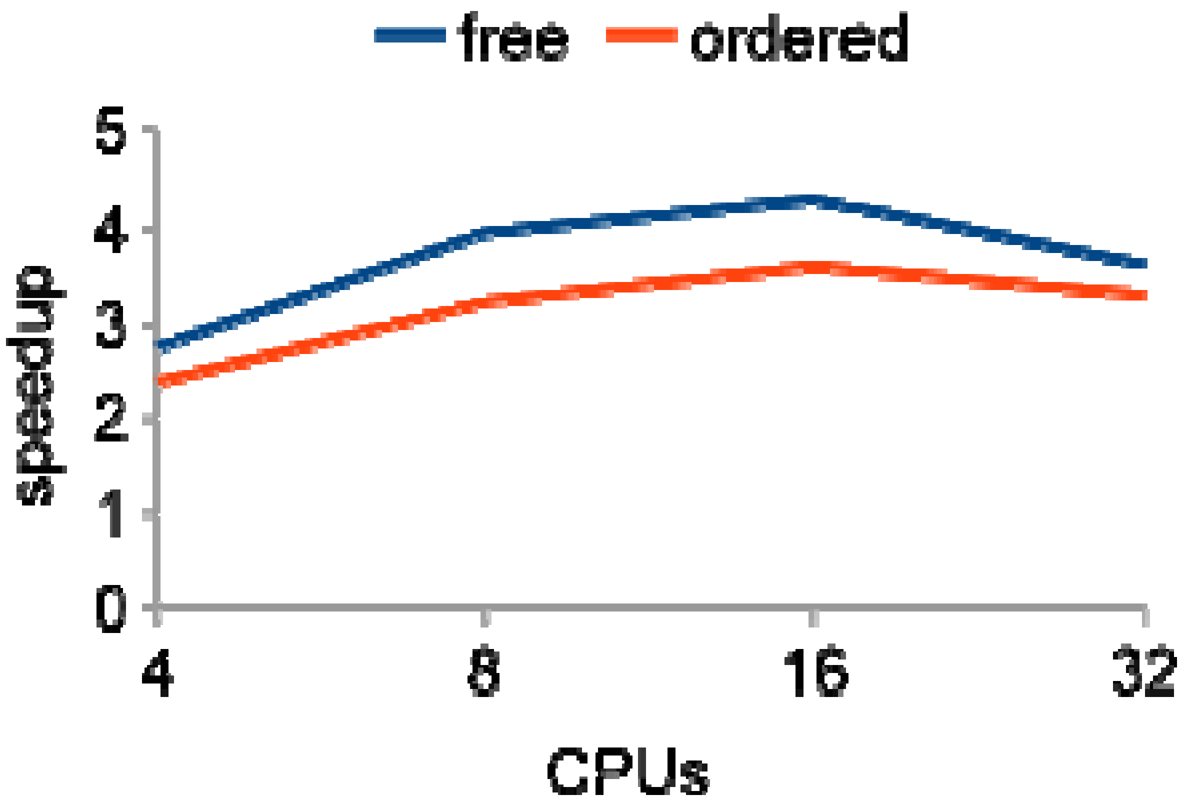
No overflow – performance == ideal

Evaluation: vacation-low



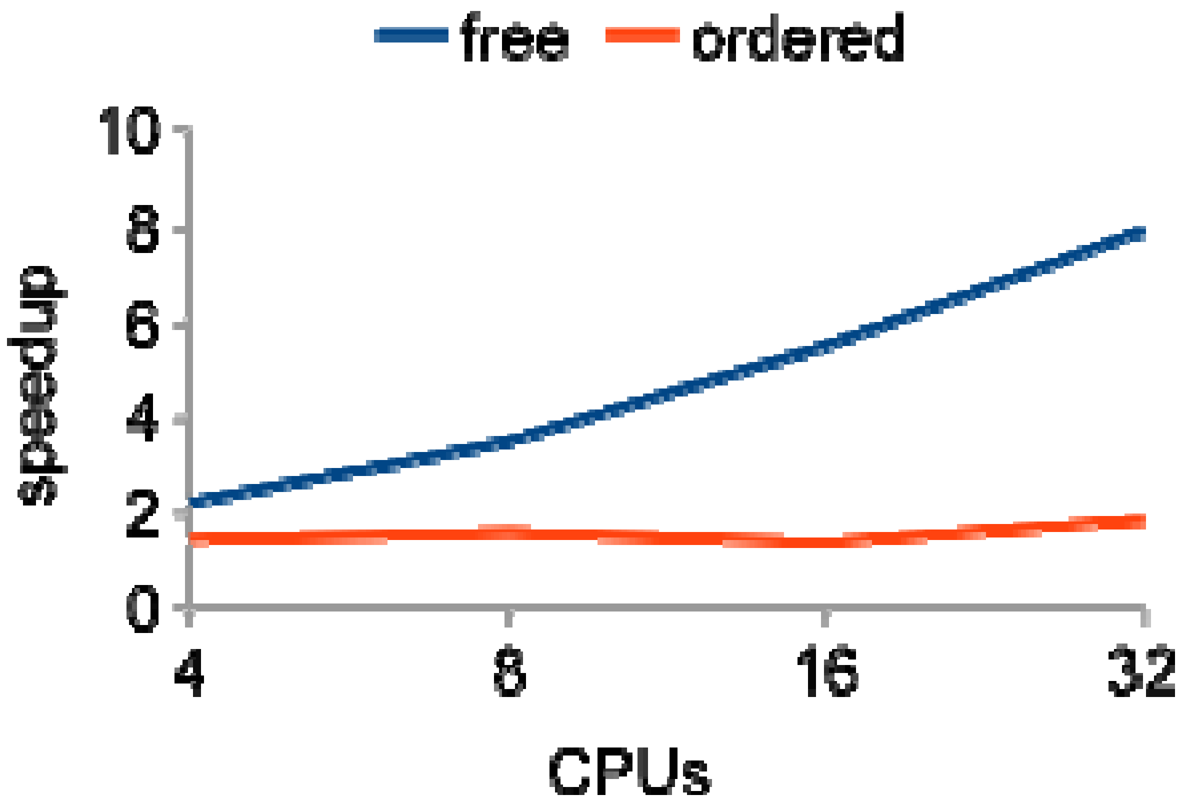
<1% overflow – performance == ideal

Evaluation: vacation-high



~3% overflow – performance near ideal

Evaluation: yada



- ~11% overflow – software bottleneck
  - 85% execution spent in software

Evaluation

- Small overflow rates, performance near ideal
  - Typical overflow unknown
  - TxLinux 2.4: <1%
- Can be limited by software synchronization
  - Global lock: yada has long critical path
  - STM can help

Related work

- Best-effort HTM
  - Herlihy & Moss ISCA '93
  - Sun Rock (Dice et al. ASPLOS '09)
- Speculative Lock Elision
  - Rajwar & Goodman MICRO '01, ASPLOS '02
- Hybrid TM
  - Damron et al. ASPLOS '06
  - Saha et al. MICRO '06
  - Shriraman et al. TRANSACT '06

We have the technology

- TxLinux 2.4
    - Add concurrency to simpler locking
- Transaction ordering
    - Best-effort becomes unbounded
- Creative software + simple ISA additions