



ROCK'S HARDWARE TRANSACTIONAL MEMORY AND HOW TO EXPLOIT IT

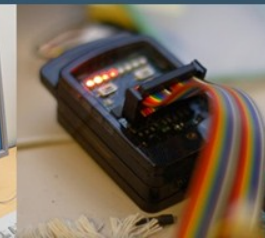
Mark Moir
Principal Investigator
Scalable Synchronization Research Group

with cameo by

Bob Cypher
Distinguished Engineer
Microelectronics



**2008
Sun Labs
Open House**



Outline

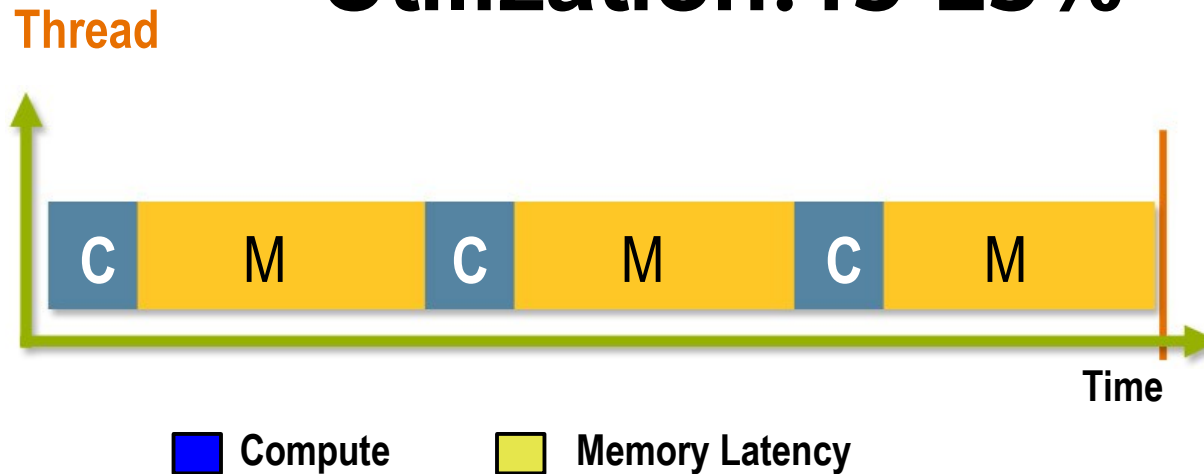
- Background
 - > Multicore revolution, Rock, Challenges of multicore programming, Transactional Memory (TM)
- Adaptive Transactional Memory Test Platform
 - > Our new simulator supporting Rock's TM feature
- Sampling of HTM-based techniques
- Selected results achieved and lessons learned using ATMTP

Single Threading

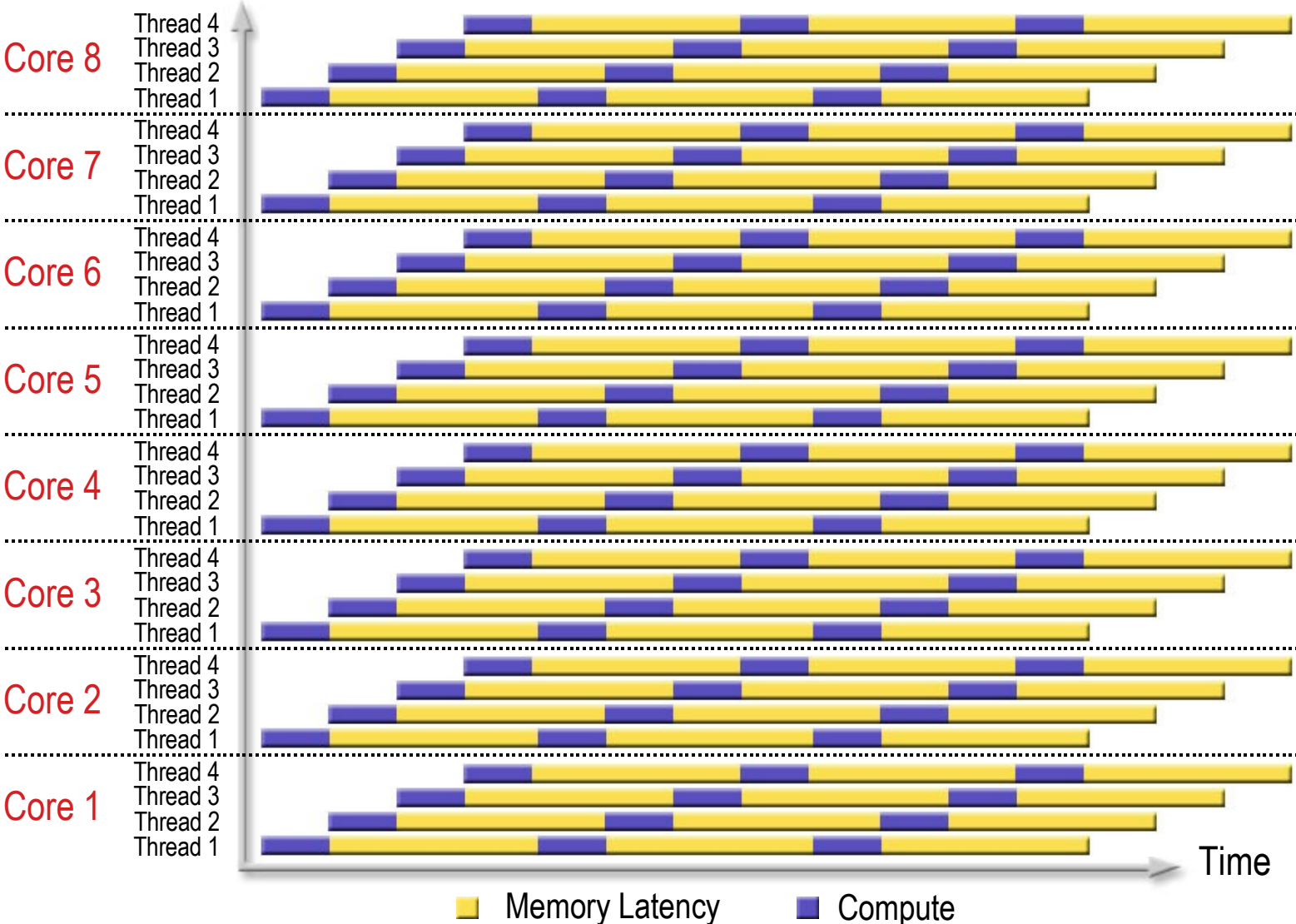
Up to 75% Cycles Waiting for Memory



Typical Processor Utilization: 15-25%



CMT – Multiple Multithreaded Cores



Instruction-Level Parallelism vs. Thread-Level Parallelism

- TLP is a huge lever for achieving high performance but...
- TLP does not eliminate need for ILP
- Too low ILP affects performance
 - > Amdahl's law
 - > Synchronization issues
 - > Data sharing issues
 - > Scalability issues
 - > Turn-around time of single-threaded applications

=> “Right amount of TLP and ILP”

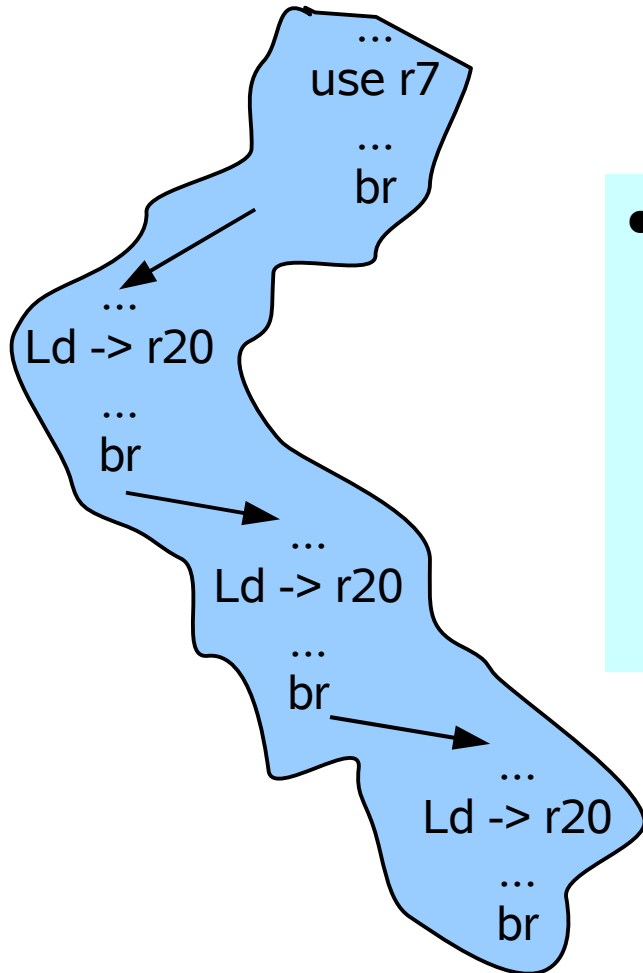
Key Idea

- Throughput is king
 - > Multi-threaded multi-cores
 - => small cores
- Latency matters
 - > Use/extend multi-threaded hardware to accelerate single thread
 - => slightly bigger cores
- The combination:
 - > Several powerful cores on a chip

Rock's Hardware Scout

Scouting Region

Ld -> r7 (miss)



- **Goals for the Scout Thread**
 - ▶ *Get to next misses*
 - ▶ *Bring data into caches*
 - ▶ *Warm-up instruction cache*
 - ▶ *Warm-up branch predictor*

Rock CMT Architecture

- Throughput performance and power efficiency:
 - > Multiple cores per chip
 - > Multiple threads per core
- Per-thread performance:
 - > Automatic hardware scouting on cache miss
 - > Architectural state checkpointed
 - > Scout thread operates without modifying architectural state
 - > Architectural state restored when cache miss is resolved

Key Challenge: Programmability

- Programming multithreaded systems is complex
 - > Need to create critical sections for modifications to shared data structures
- Standard approach: Locks
 - > Expensive
 - > Lock granularity tradeoff -> contention vs. complexity
 - > Scalability -> contention, pessimism
 - > Complexity -> deadlock avoidance
 - > Not composable

New Approach: Transactional Memory

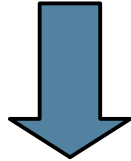
- Herlihy and Moss, 1993
- A transaction is a finite sequence of machine instructions, executed by a single process, that satisfies two properties:
 - > Serializability
 - > Atomicity
- Multiple transactions can execute simultaneously as long as their data accesses are non-interfering
- Useful for supporting promising new programming models, and improving performance of existing code

Rock's HW Transactional Memory (HTM)

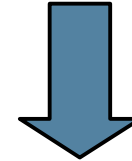
- Key ideas:
 - > Leverage Rock's checkpoint mechanism to execute transaction speculatively
 - > Implement atomicity guarantees in hardware
 - > Successful transaction modifies architectural state
 - > Failed transaction restores checkpointed architectural state
- Rock is the first processor to provide Hardware Transactional Memory!

Rock's HTM feature

- Best-effort HTM can abort transactions that exceed resources or encounter difficult events or instructions:
 - > Rock examples: function calls, `sdivx` instruction, exceptions
- “Generic” best-effort HTM + Rock extension
(as in ASPLOS 2006 paper)



```
chkpt <fail_addr>  
commit
```



```
rd %cps, <dest_reg>
```

ATMTP

- Provides first-order approximation of Rock's:
 - > success/failure characteristics
 - > feedback from cps register
- Goal is not to accurately simulate Rock implementation or performance.
- Goal is to enable experimentation with HTM code before Rock is widely available, debugging even after it is.
- Based on Wisconsin GEMS. Open source.

Preliminary exploration

- Overview of experiments described in our Transact 2008 paper:
 - > Transactional red-black tree with HyTM and PhTM
 - > Exposing TM to JavaTM programs, optimising Java-based STM
 - > DCAS-based collections in Java
 - > Eliding locks explicitly for libc and STL
 - > Eliding locks *implicitly* in Java
- Summary: variety of encouraging results, some pitfalls identified and some lessons learned

Preliminary exploration

- Overview of experiments described in our Transact 2008 paper:
 - > Transactional red-black tree with HyTM and PhTM
 - > Exposing TM to JavaTM programs, optimising Java-based STM
 - > DCAS-based collections in Java
 - > Eliding locks explicitly for libc and STL
 - > Eliding locks *implicitly* in Java
- Summary: variety of encouraging results, some pitfalls identified and some lessons learned

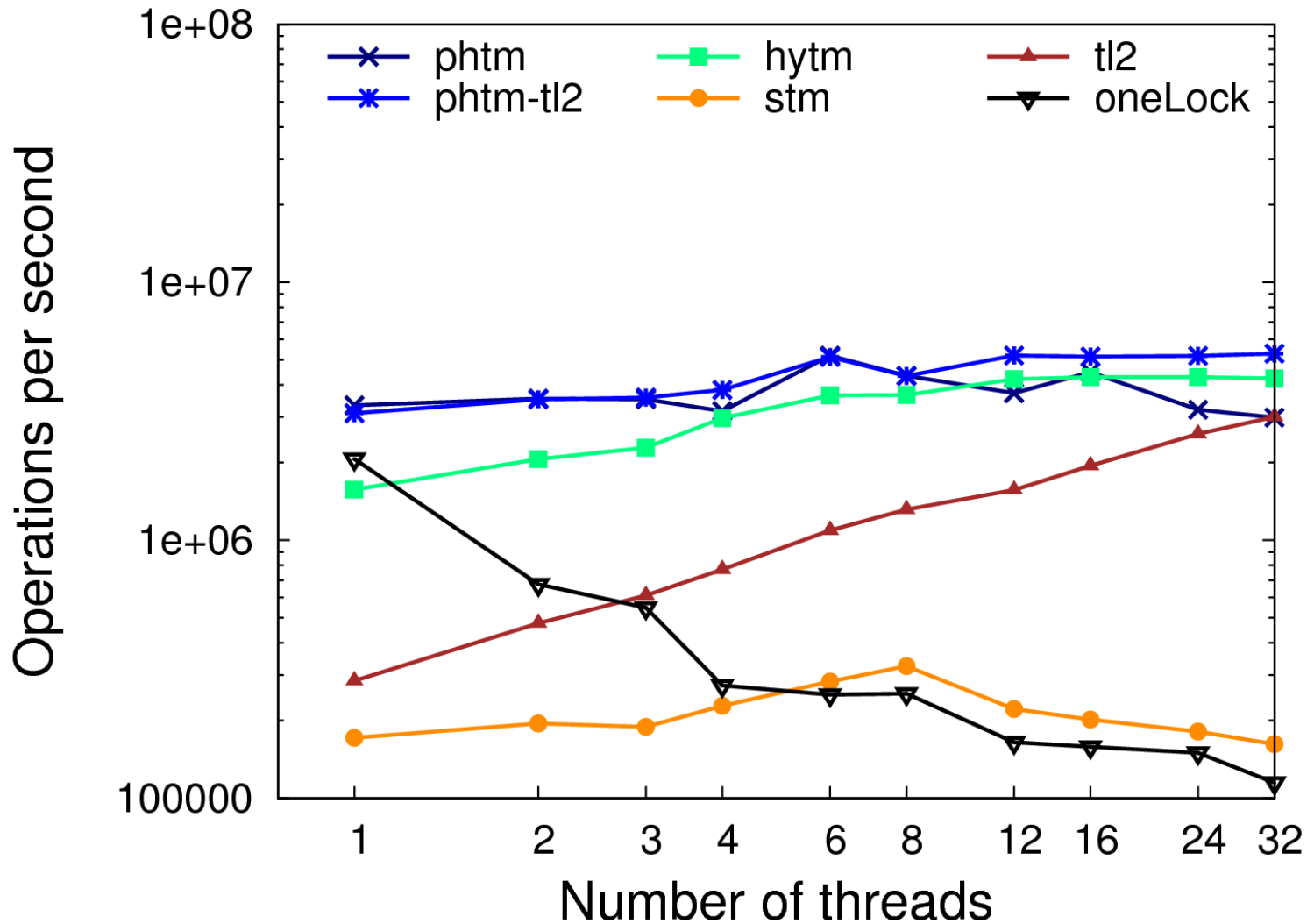
Background: HyTM and PhTM

- Fully functional software transactional memory
- Use best-effort hardware transactional memory (if available) to boost performance
- Programmer unaware of specific limitations of HTM; performance improves with improvements in HTM
- In HyTM, hardware and software transactions execute together, impose overhead on each other
- In PhTM, different kinds of transactions run in different “phases”, eliminating such overhead

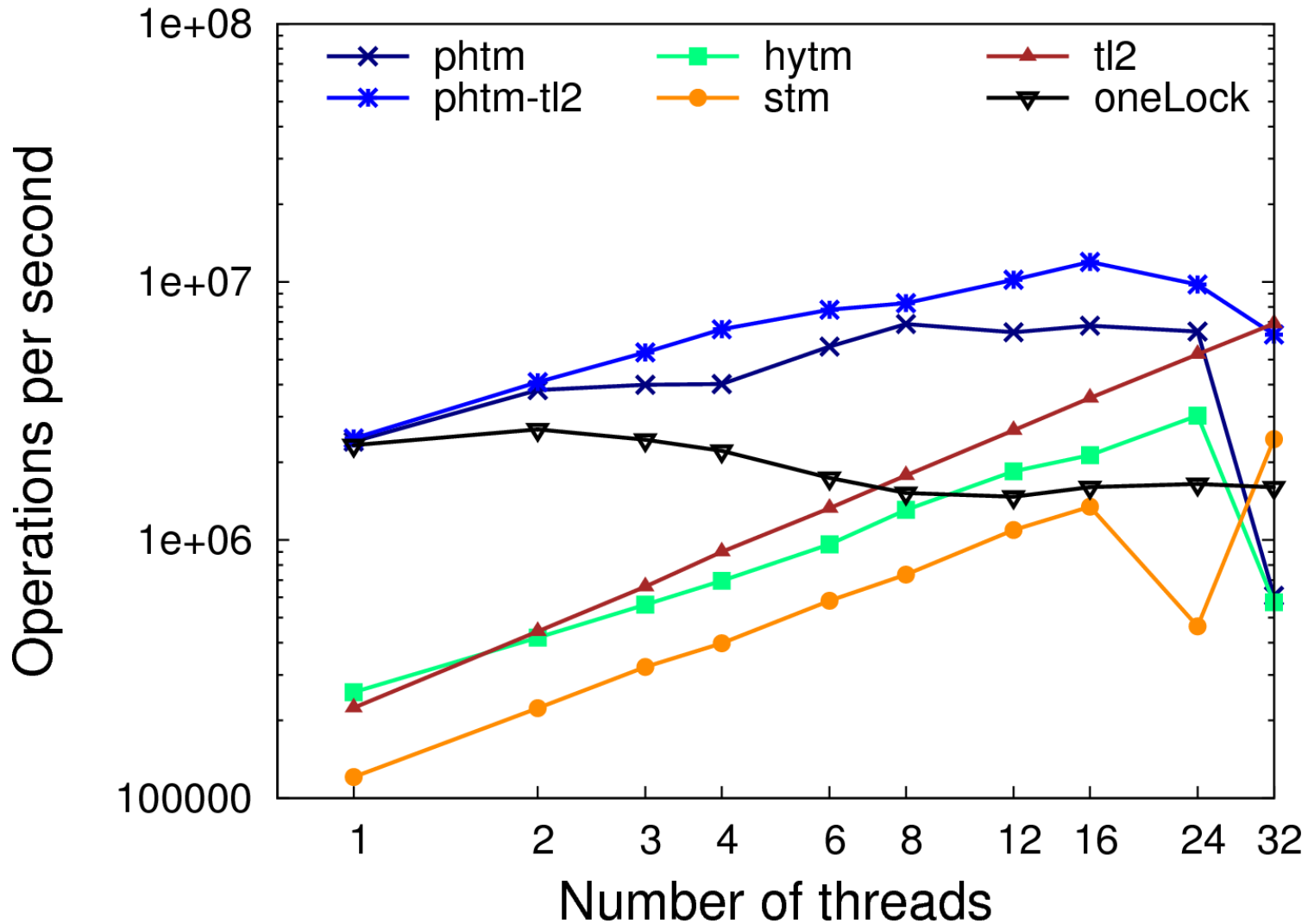
Transactional red-black tree

- Concurrent Red-Black trees very challenging
- We implemented transactional version using our HyTM compiler (ASPLOS 2006); working with compiler group towards more robust transactional programming model in C/C++ compilers
- Experiment:
 - > initialise tree with 2000 keys from range [0,4095]
 - > each thread repeatedly inserts (20%), deletes (20%) or looks up (60%) randomly chosen key
 - > measure total operations completed per second

Red-Black tree on old simulator



Red-Black tree on ATMTP



Red-Black tree with ATMTP

- Previous recursive version unsuccessful due to deeply nested function calls
- After switch to iterative version, inlining, PhTM successfully completes (almost) all operations using hardware transactions
- HyTM less successful: 30% of operations executed as software transactions
- Cause: TLB misses cause exceptions, txn is aborted so exception not processed, so TLB not loaded
- In this case, ITLB was the problem.
- Need “warm up” techniques to avoid repeated failure

Background: DCAS

Double Compare-And-Swap

```
bool DCAS(a0,a1,old0,old1,new0,new1) {  
    atomic {  
        if (*a0==old0 && *a1==old1) {  
            *a0 = new0; *a1 = new1;  
            return true;  
        }  
        return false;  
    }  
}
```

- Atomically modify two memory locations if they contain expected values
- Significant algorithmic power over existing synchronization operations
- Easy to implement using HTM

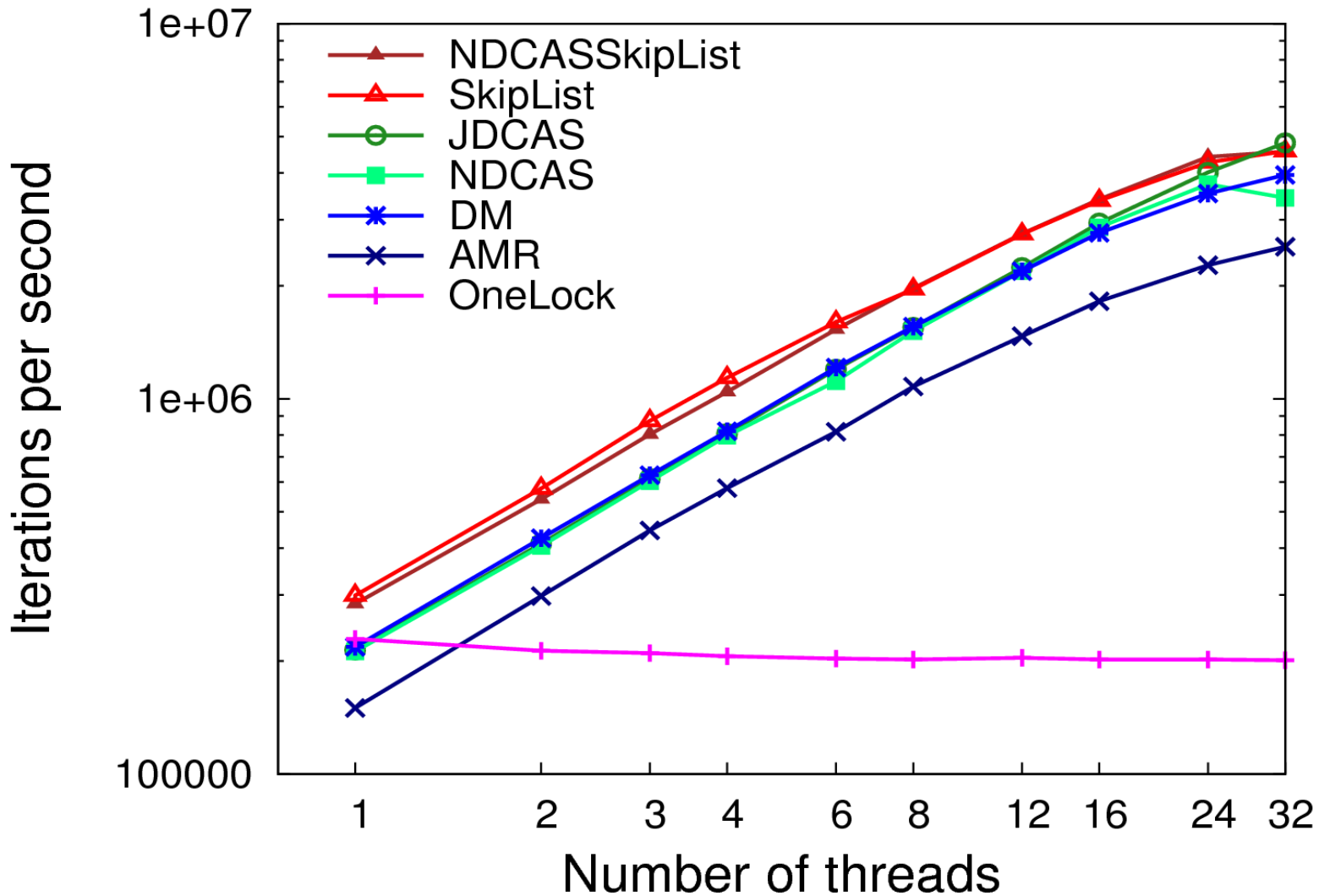
DCAS-based collections in Java

- First DCAS approach (JDCAS):
 - > Modify JVM to expose “unsafe” interface to HTM instructions
 - > implement DCAS in Java inside transaction
- Eventually worked well, after we learned some lessons:
 - > compilation “catch 22”
 - > data warmup tricky due to “hidden” code, e.g. class metadata needed for type cast
 - > nonobvious conflicts due to (false) sharing on GC metadata; changes going into JVM to avoid this

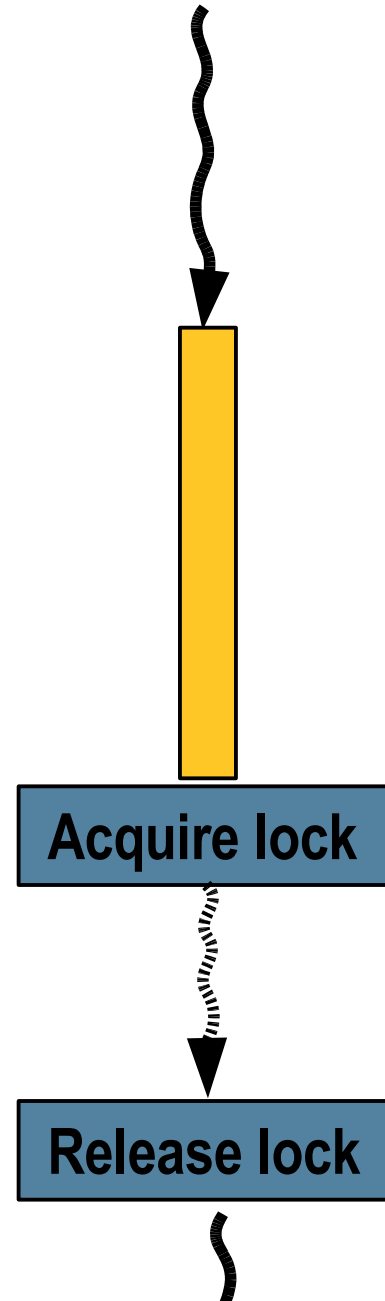
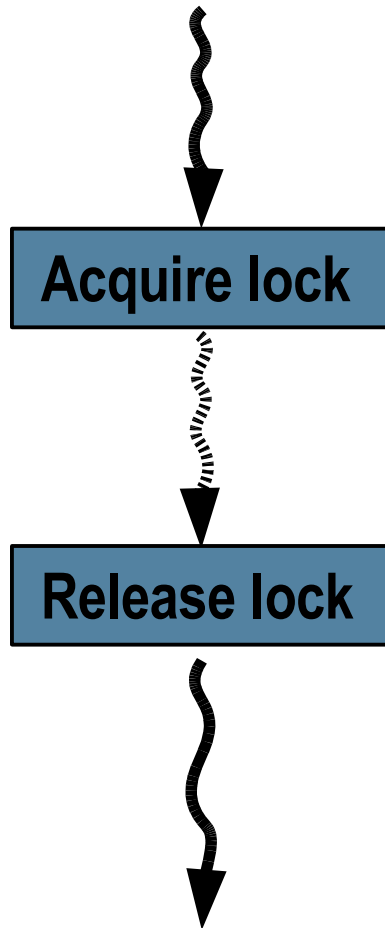
DCAS-based collections in Java

- Second DCAS approach (NDCAS):
 - > implement DCAS in native assembly code
 - > no “surprise” code executed
 - > factor GC metadata updates out of transactions
 - > works well, but less flexible than general Java code inside hardware transaction
- We designed new DCAS-based list and skiplist
- Both achieved good, scalable performance matching existing ones in `java.util.concurrent`

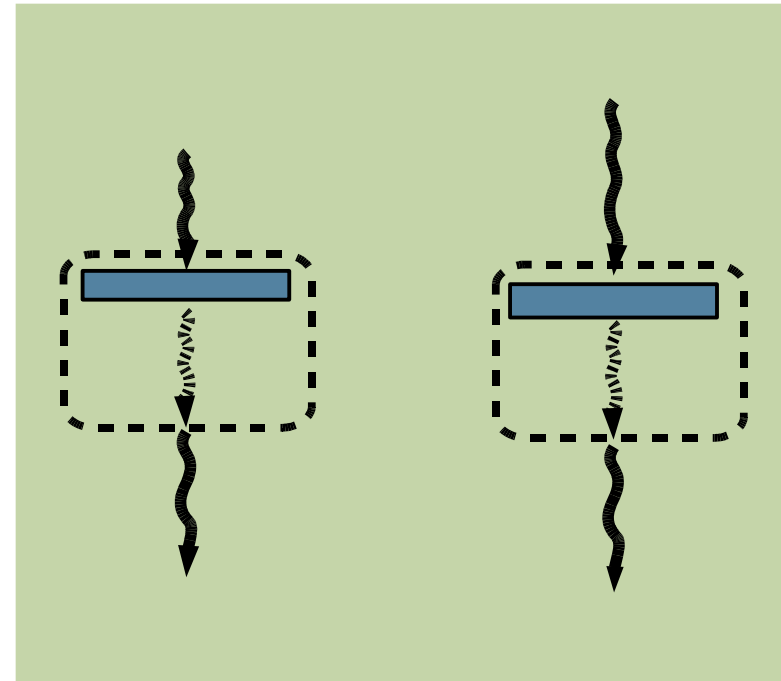
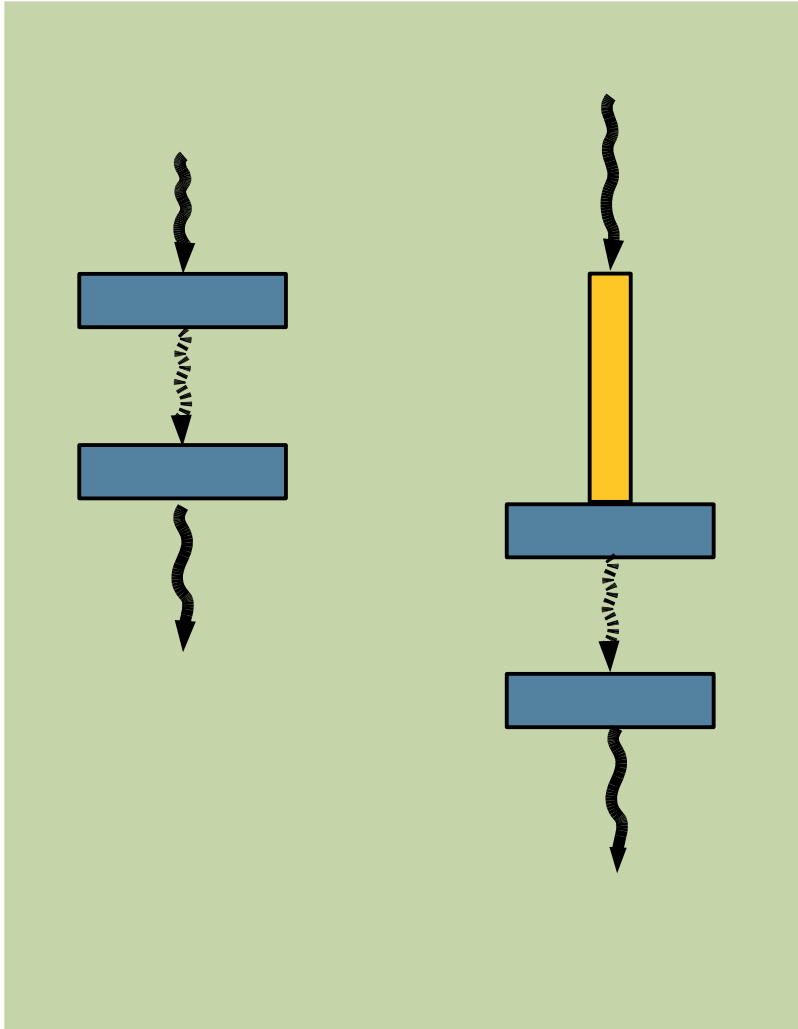
List and Skiplist Results



Lock bottlenecks



Eliminating lock bottlenecks with HTM



- Use HTM to execute critical section *without acquiring lock*

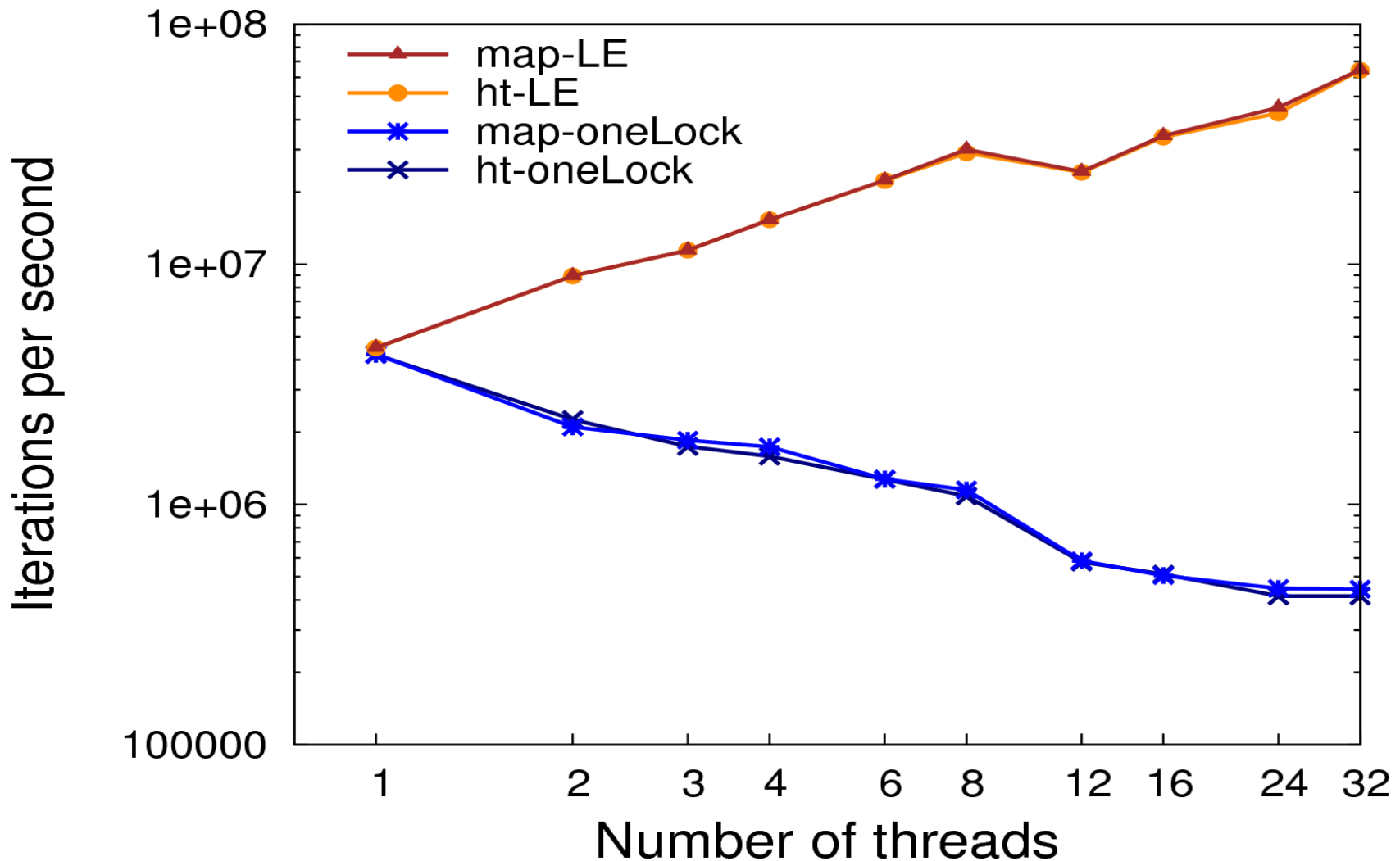
Eliding synchronization in Java

- Modified JVM attempts synchronized blocks and methods using hardware transactions that:
 - > start transaction
 - > check lock is not held
 - > execute critical section
 - > attempt to commit; retry if unsuccessful
 - > if repeatedly unsuccessful, eventually acquire lock
- Prototype does not yet recompile to original locking code if technique is unprofitable

Eliding synchronization in Java

- First experiment:
 - > Tested two collections from `java.util`:
 - **HashMap** (with synchronized wrapper)
 - **HashTable** (synchronization built in)
 - > Initialised collection with a set of objects
 - > Each thread repeatedly looks up randomly chosen object
 - > Measured number of lookups per second
- **HashMap** was successful; operations all completed in hardware
- **HashTable** required small modification to factor division (`sdivx` instruction) out of transaction

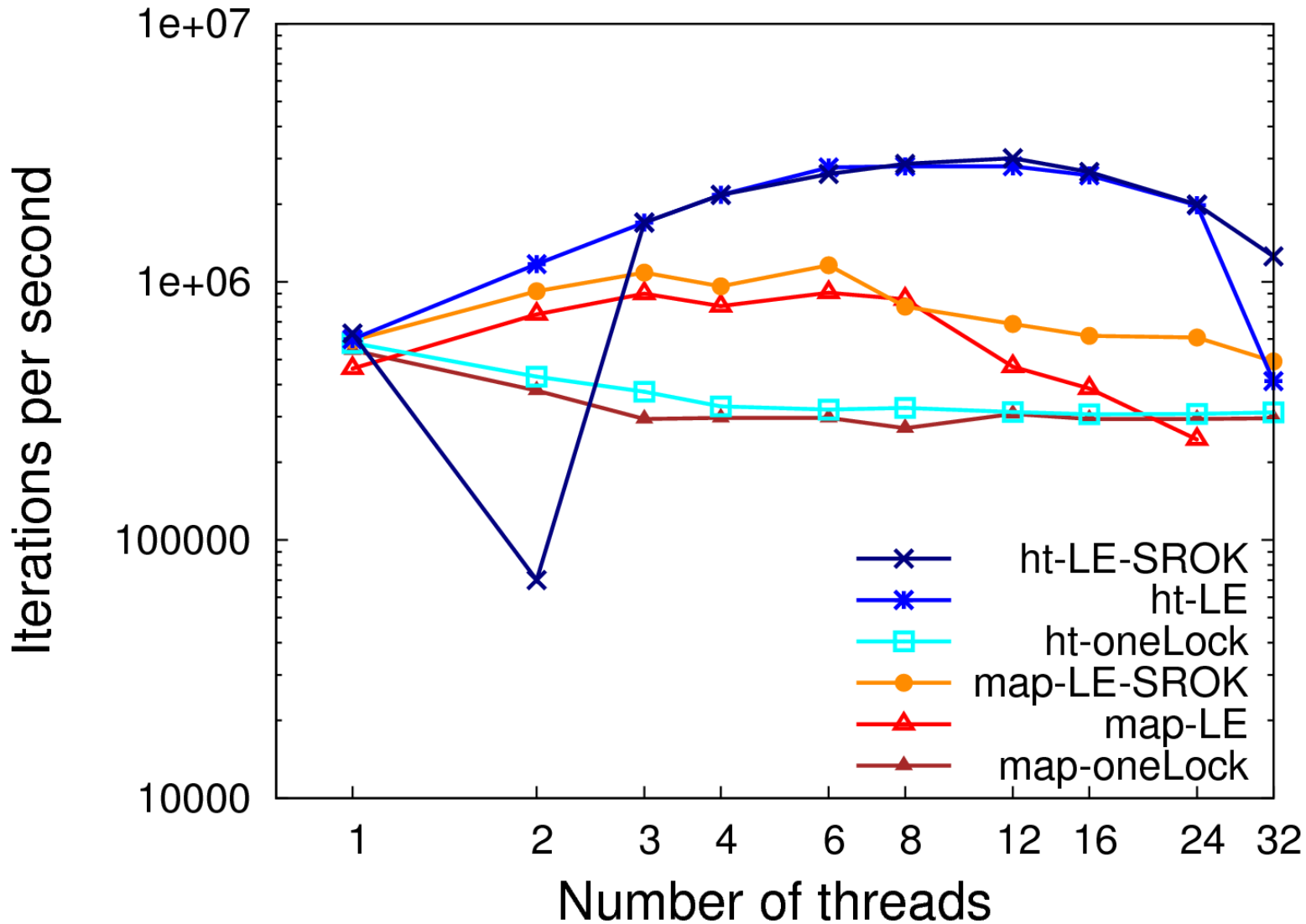
Java lock elision, 100% lookups



Eliding synchronization in Java

- Second experiment:
 - > Tested two collections from `java.util`:
 - **HashMap** (with synchronized wrapper)
 - **HashTable** (synchronization built in)
 - > Initialised collection with a set of objects
 - > Each thread repeatedly chooses an operation and key value at random: 80% lookups, 10% inserts, 10% deletes
- **HashTable** was quite successful; most operations completed in hardware
- **HashMap** less profitable so far, due to lack of inlining

Java lock elision, 80%:10%:10%



Concluding Remarks

- Best effort HTM is coming in Rock.
- We're exploring a variety of techniques for exploiting HTM for traditional code, libraries, and new transactional programming models
- ATMTP is a valuable tool for developing and testing HTM-based code before Rock is available; will also be useful afterwards
- Preliminary explorations have yielded:
 - > various encouraging results, some pitfalls that require workarounds, some valuable lessons
- Many ideas yet to be explored



<http://research.sun.com/scalable>

Mark Moir

mark.moir@sun.com

Bob Cypher

robert.cypher@sun.com



**2008
Sun Labs
Open House**



References

- “Hybrid Transactional Memory”, ASPLOS 2006
- “PhTM: Phased Transactional Memory”, Transact 2007
- “Integrating Transactional Memory into C++”, Transact 2007
- “The Adaptive Transactional Memory Test Platform: A Tool for Experimenting with Transactional Code for Rock”, Transact 2008
- “Applications of the Adaptive Transactional Memory Test Platform”, Transact 2008
- Wisconsin GEMS: <http://www.cs.wisc.edu/gems>
- ATMTP: <http://www.cs.wisc.edu/gems/doc/gems-wiki/moin.cgi/ATMTP>