Speculative Lock Elision (SLE): Enabling Highly Concurrent Multithreaded Execution

Ravi Rajwar
James R. Goodman

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Originally presented at MICRO 2001
Presented today by: Torgin Mackinga

Elision Definition

elision – noun

The act of omitting (skipping) something

Executive Summary

- **Problem:** Conventional locking can introduce unnecessary serialization
- **Goal:** Dynamically elide unnecessary locks
- **Solution:** Speculatively execute critical sections
- **Hardware implementation:** Requires no changes to ISA
  - Programmer and compiler-transparent
  - Existing code can be sped up without any changes
- **Effect:** Reduces the burden on programmers by allowing them to use frequent serialization to write correct code

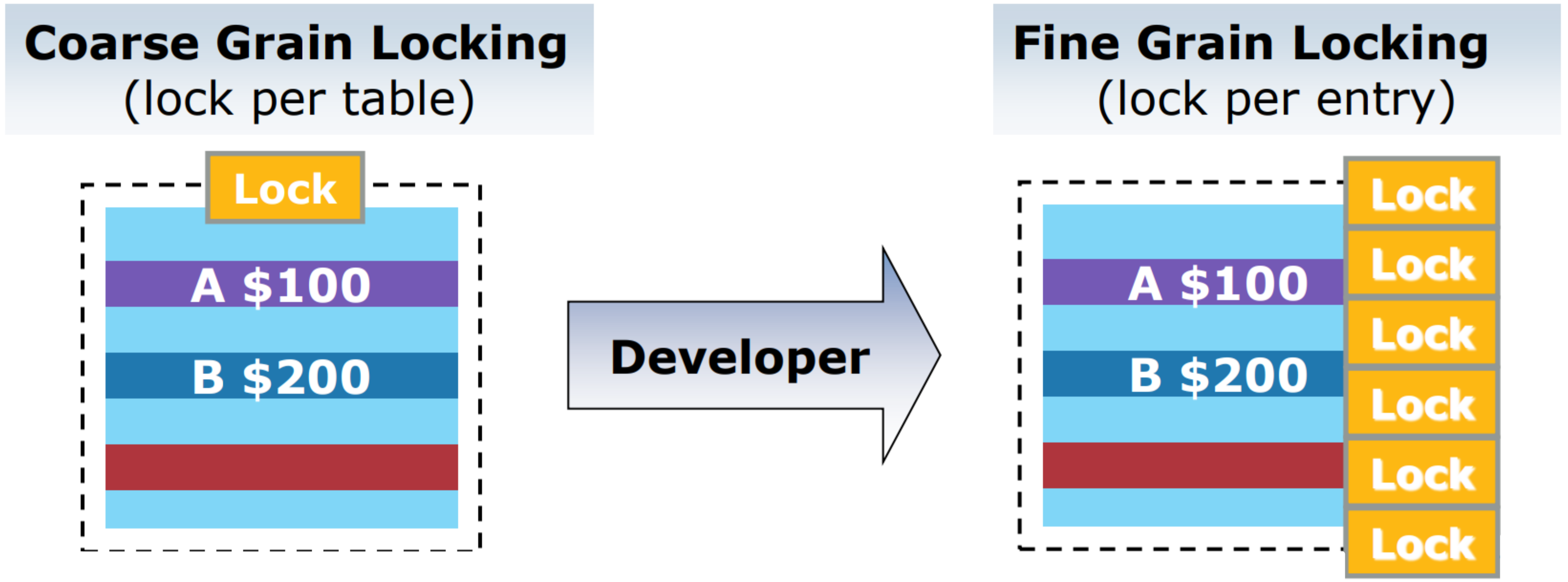
Outline

- Problem & Goal
- Mechanism Overview
- Implementation
- Methodology & Results
- Summary
- Strengths & Weaknesses
- Thoughts and Ideas
- Open discussion

Outline

- **Problem & Goal**
- Mechanism Overview
- Implementation
- Methodology & Results
- Summary
- Strengths & Weaknesses
- Thoughts and Ideas
- Open discussion

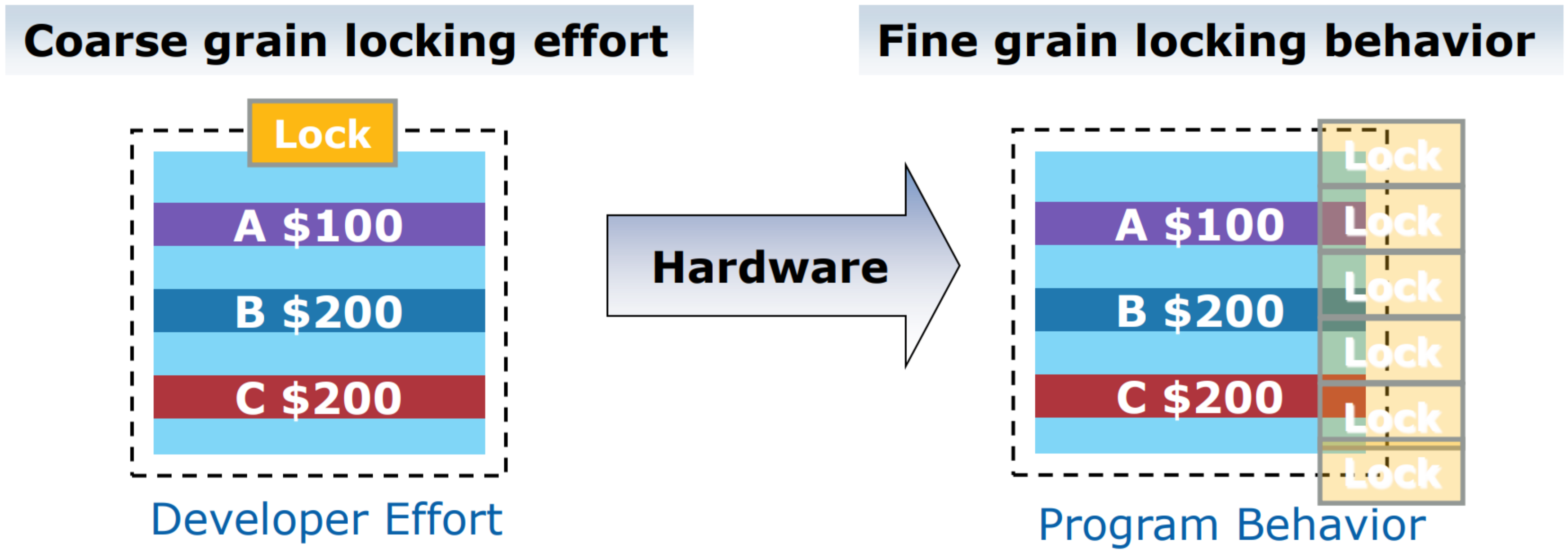
Problem



Unnecessary serialization

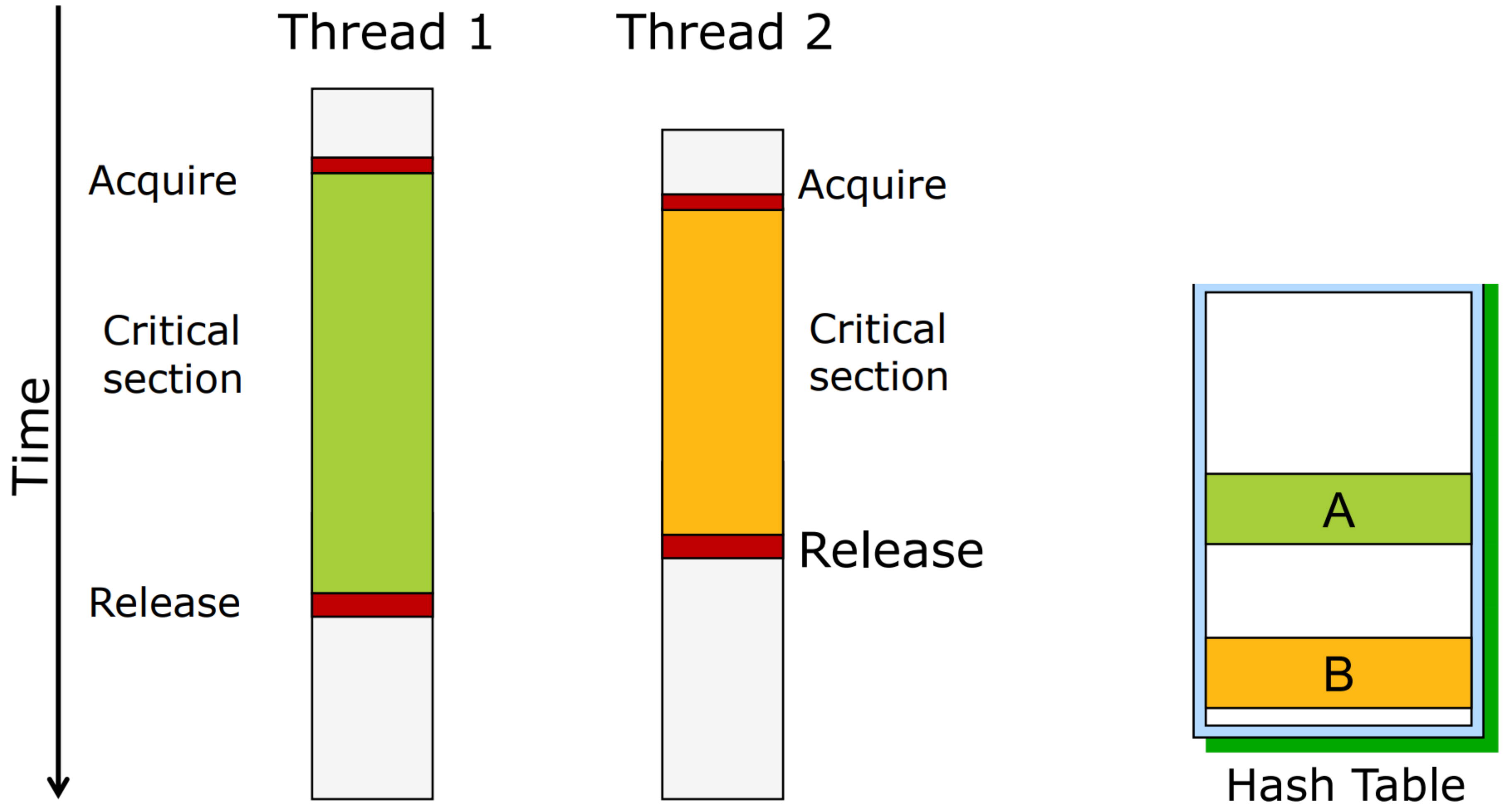
Expensive
Difficult to debug

What we really want…



Remove unnecessary serialization automatically through hardware

Unnecessary serialization example



Unnecessary serialization example

Thread 1

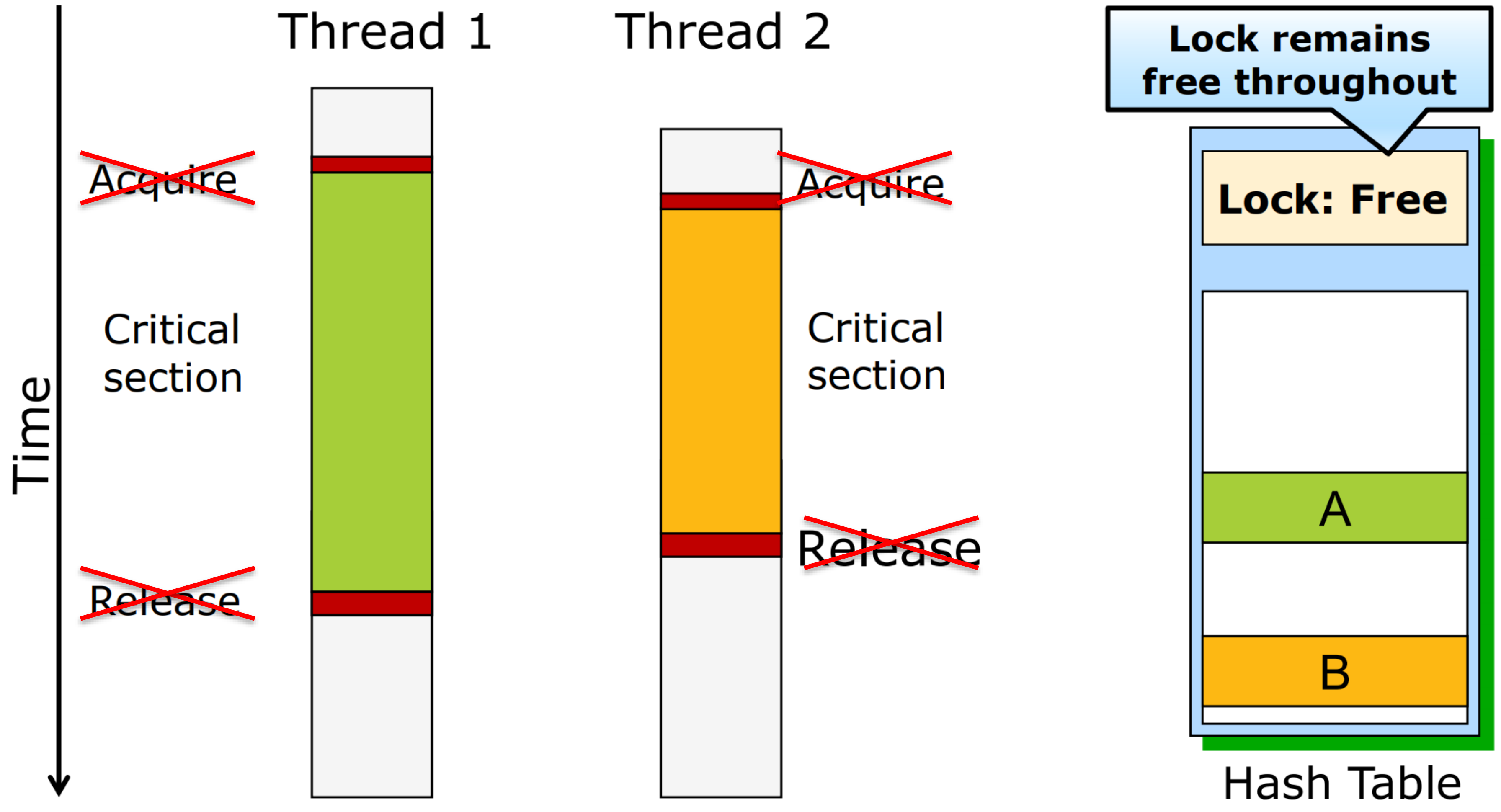
```
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(X)
if (!var)
    hash_tbl.add(X);
UNLOCK(hash_tbl.lock)
```

Thread 2

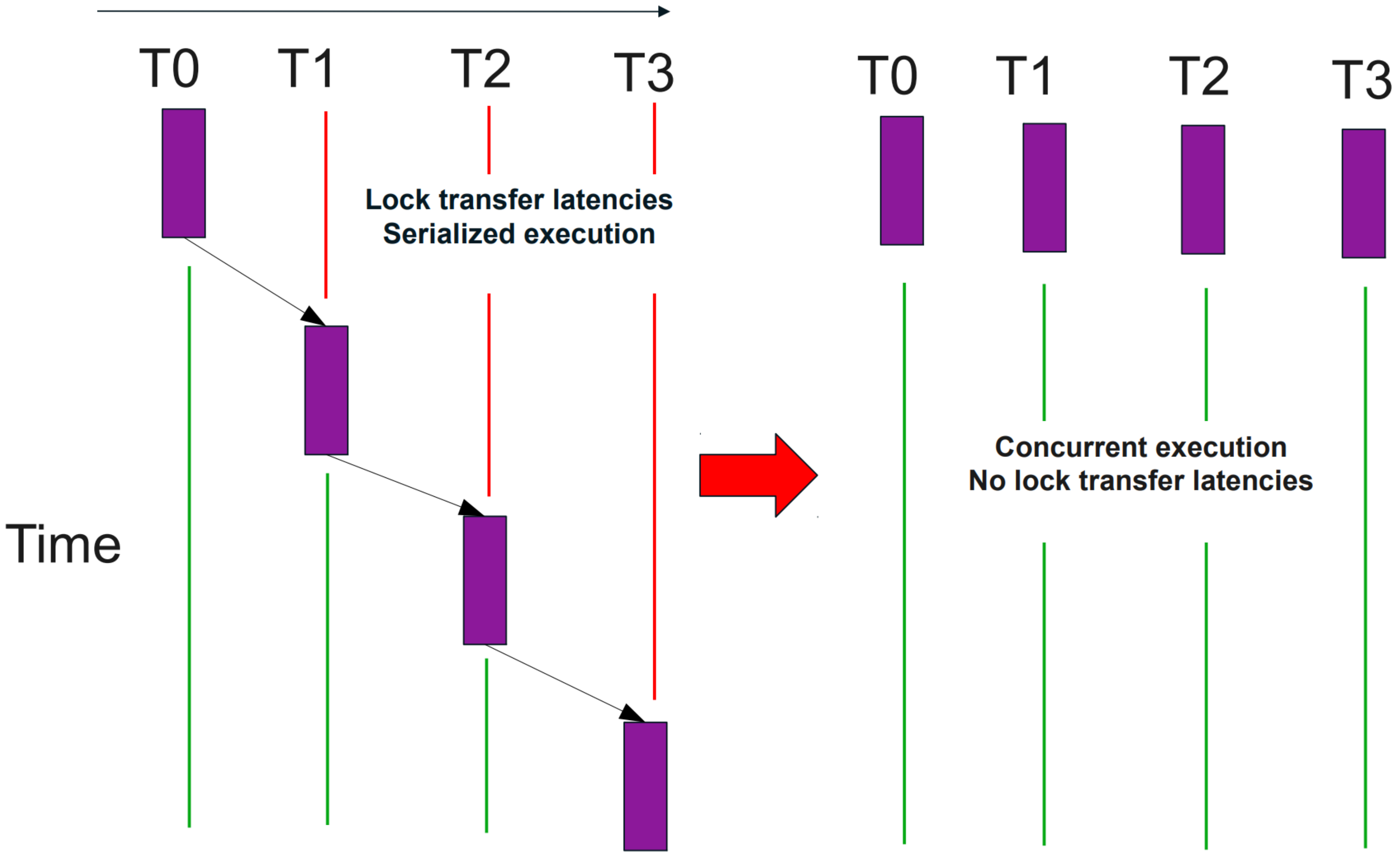
```
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(Y)
if (!var)
    hash_tbl->add(Y);
UNLOCK(hash_tbl.lock)
```

- Lock only required for add
- Difficult to statically fix

Goal: Elide unnecessary locks



Speculative execution



Outline

Atomicity conditions

- Locks trivially guarantee atomicity

- Sufficient conditions:

1. Data read within a speculatively executing critical section is not modified by another thread before it completes.

2. Data written within a speculatively executing critical section is not accessed by another thread before it completes.

SLE Algorithm

1. Whenever we would acquire a lock, predict that critical section will occur atomically and **elide** the lock acquire.

2. Execute critical section **speculatively** and buffer results.

3. If hardware cannot provide atomicity, trigger misspeculation, **recover** and explicitly acquire lock. (or retry)

4. If we encounter lock-release, atomicity was not violated. **Elide** lock-release operation, **commit** speculative state.

Outline

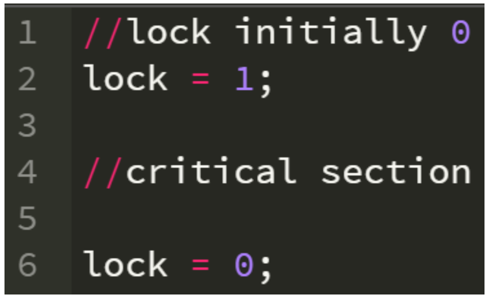
- Problem & Goal
- Mechanism Overview
- **Implementation**
- Methodology & Results
- Summary
- Strengths & Weaknesses
- Thoughts and Ideas
- Open discussion

Implementation

1. When can we speculate?
2. How is speculative state buffered?
3. How is misspeculation detected?
4. How is speculative state committed?

1. When can we speculate?

- Silent store-pairs
    - Store followed by another store which reverts it



```
//lock initially 0
lock = 1;

//critical section

lock = 0;
```

- Detect instructions which could be a lock acquire
    - ldl_l/stl_c, Compare and swap etc
- Predict if another store will form a silent store pair
    - If yes, initiate speculation and elide lock acquire
- Everything happens in hardware, no extra instructions needed

2. How is speculative state buffered?

Speculative register state

- Reorder Buffer (ROB)
  - Uses same recovery mechanisms as branch prediction
  - Limits size of critical section (w.r.t. dynamic instructions)
- Register Checkpoint
  - Architected register state is stored on entering speculation
  - Allows instructions to speculatively retire

Speculative memory state

- Processor write-buffer (between processor and L1-cache) doesn't commit into cache until elision is validated
  - Limits size of critical section (w.r.t number of unique cache lines modified)

3. How is misspeculation detected?

Atomicity violations

- Detected by cache coherency protocol
- If using register checkpoints, cache needs to be augmented with an access bit
  - On write, access bit is set. On commit, unset

Misspeculation due to resource constraints

- Write-buffer or ROB full
  - Does not always require a restart
    1. Take the lock
    2. Commit speculative state
    3. Continue executing
- Uncached accesses or events (e.g. some system calls)
  - Always triggers misspeculation

4. How is speculative state committed?

- Commit must appear atomically!

Registers

- Nothing to do

Memory

- State
  - Speculative store in write-buffer -> exclusive memory request
  - All speculative entries have a block in exclusive state
- Data
  - Write-buffer is set to have latest architectural state
  - Reading from write-buffer not on critical path

Key features of SLE

- Enables highly concurrent multithreaded execution
  - Execute multiple critical sections at once
- Simplifies writing correct multithreaded code
  - Use conservative synchronization without significant performance impact
- Can be implemented easily in microarchitecture
  - No ISA support or changes to coherence protocols necessary
- Transparent to programmer
  - Continue using well-understood synchronization routines

Outline

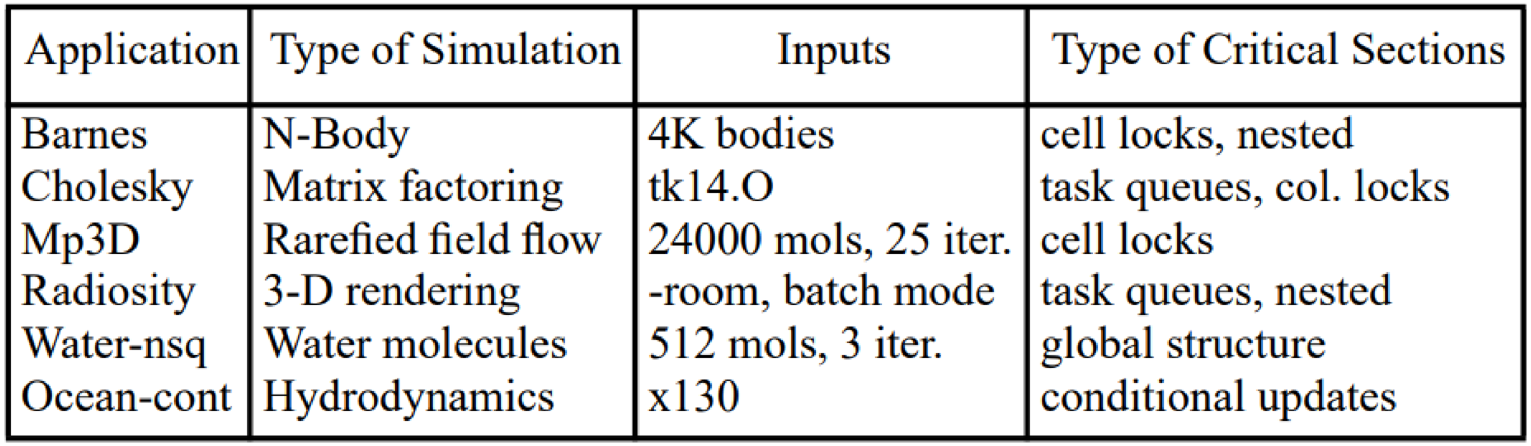
- Problem & Goal
- Mechanism Overview
- Implementation
- **Methodology & Results**
- Summary
- Strengths & Weaknesses
- Thoughts and Ideas
- Open discussion

Methodology

- 3 configurations: a chip multiprocessor (CMP), a bus system (SMP) and a directory system (DSM).
- 8-core and 16-core processor
- Memory consistency model: Total store ordering
  - same as x86
- Register checkpoint for register recovery
- SimpleMP: simulator for multi-threaded binaries

Benchmarks

- Microbenchmark:
  - N threads each increment a different counter (2^16)/N times. All counters protected by a single lock.
  - Worst case scenario for conventional locking
- 6 applications
  - Padded to reduce false sharing

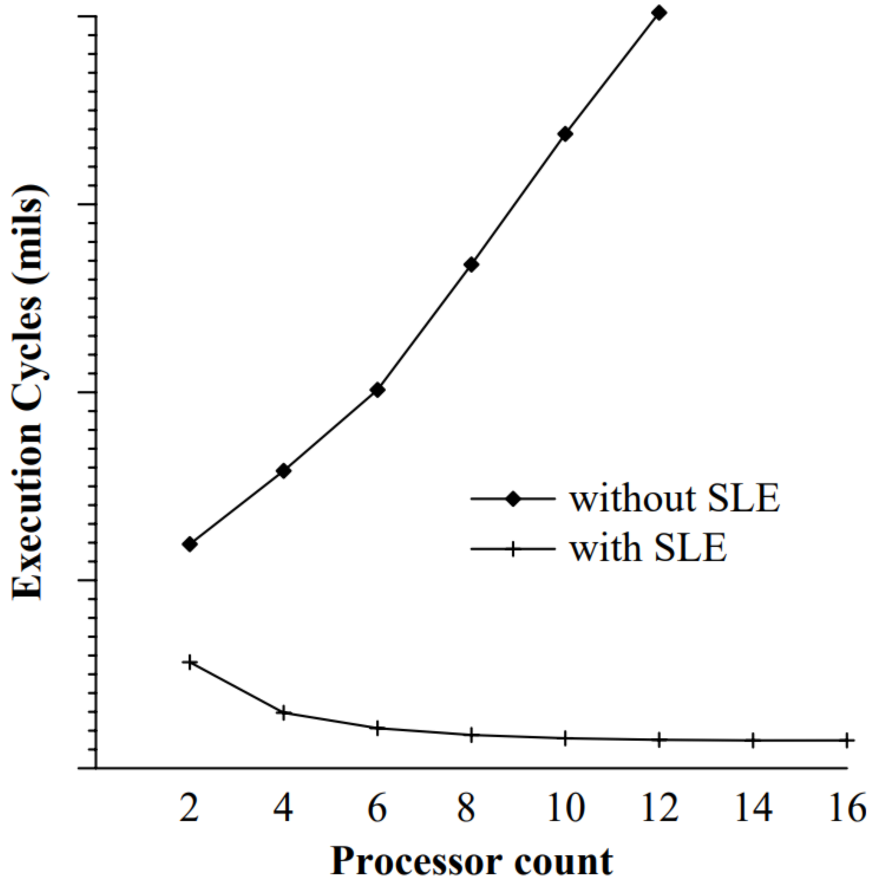


Application	Type of Simulation	Inputs	Type of Critical Sections
Barnes	N-Body	4K bodies	cell locks, nested
Cholesky	Matrix factoring	tk14.O	task queues, col. locks
Mp3D	Rarefied field flow	24000 mols, 25 iter.	cell locks
Radiosity	3-D rendering	-room, batch mode	task queues, nested
Water-nsq	Water molecules	512 mols, 3 iter.	global structure
Ocean-cont	Hydrodynamics	x130	conditional updates

Table 2: *Benchmarks*

Microbenchmark result

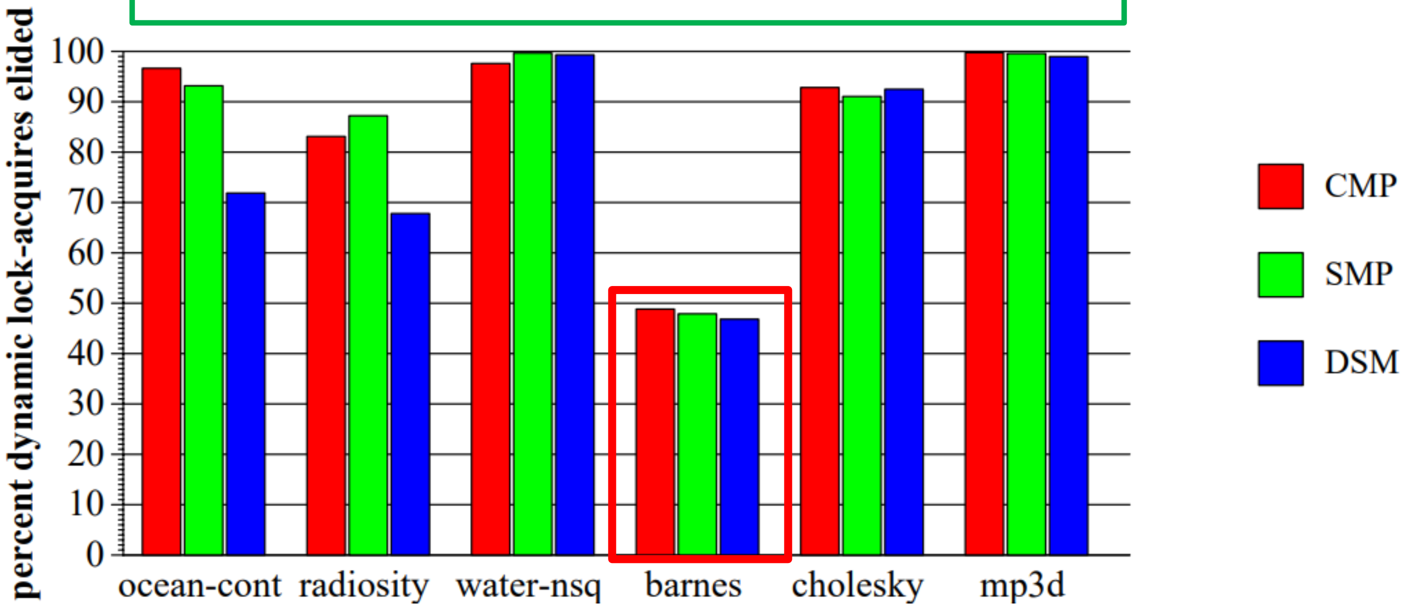
Perfect scaling with SLE!



Percentage of locks elided

- Barnes worst due to high lock contention
- Restart: 1
  - Restart 0 had 10-30% fewer locks elided

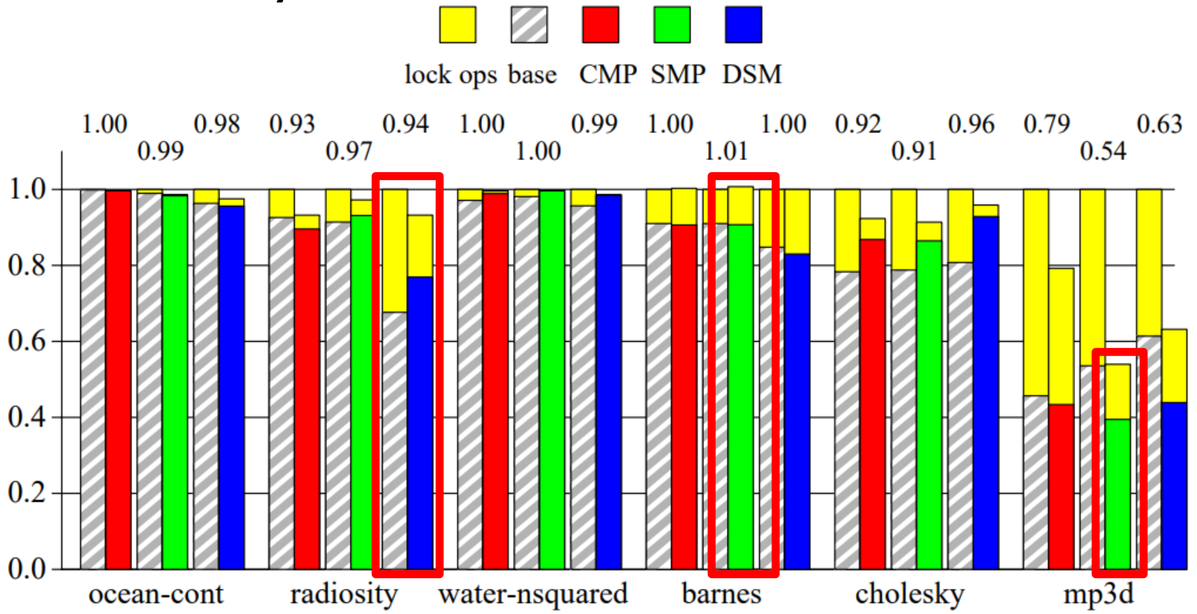
Some workloads up to 100%!



Percentage of dynamic lock acquires/releases elided
8 processors

Performance gains

- Base: Performance without SLE
- Lock ops: Time spent on lock variable accesses
- Non-lock part may increase, as it becomes critical path
- Main reasons for performance gains:
  - Concurrent critical section execution
  - Reduced memory latencies and traffic



Avg: 0.92

a) normalized execution time (y axis) for 8 processors

Outline

Summary

- **Problem:** Conventional locking can introduce unnecessary serialization
- **Goal:** Dynamically remove unnecessary locks
- **Solution:** Speculatively execute critical sections
- **Easy to implement:** Requires no changes to ISA or coherency protocol
  - Programmer and compiler-transparent
  - Existing code can be sped up without any changes
  - Best case: 46%, worst case: -1%
  - Existing code is guaranteed to remain correct
- **Effect:** Reduces the burden on programmers by allowing them to use frequent serialization to write correct code

Outline

- Problem & Goal
- Mechanism Overview
- Implementation
- Methodology & Results
- Summary
- **Strengths & Weaknesses**
- Thoughts and Ideas
- Open discussion

Strengths

- Big gains at low cost
- Reduces the burden on programmers
- Existing code can be sped up
- Implementation changes almost none of the existing design
- Tackled problems ahead of its time
- Answered many of my questions as they came up

Weaknesses

- Not convinced that every lock can be recognized
- Did not mention cost of register checkpoints
- Does not help lock-free code
- Could have explained more about how prediction works
- No explanation of differences between configurations

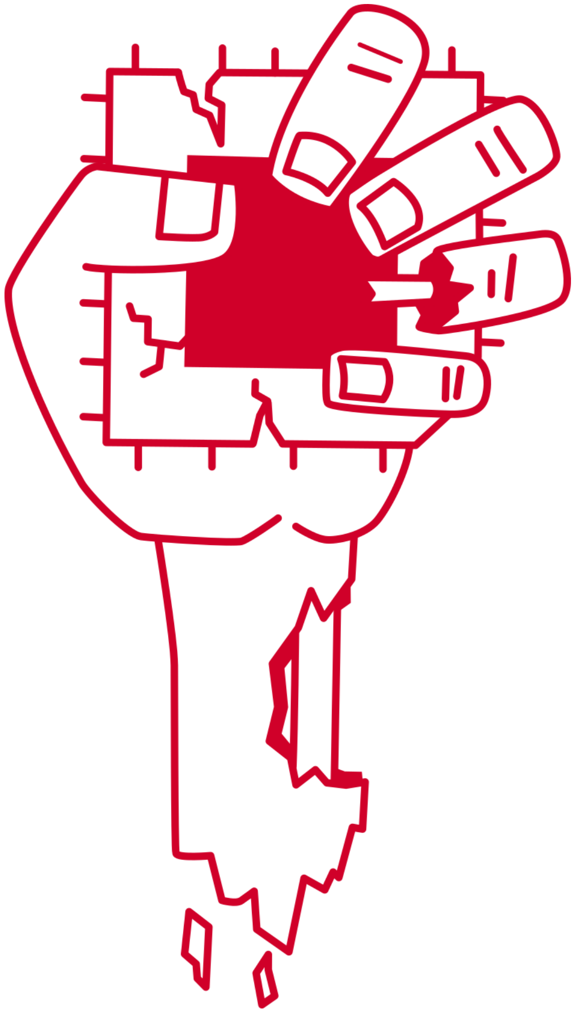
Questions?

Thoughts and Ideas

Is this implemented?

- Yes, Intel TSX in Intel Haswell 2013 (12 years later)
  - By Ravi Rajwar, author of this paper
- 2 variants: HLE and transactional memory
- Transactional memory is programmer visible.
  - Reset threshold: 9
- Still available in 2019 Intel CPUs
  - E.g Intel Core i9-9000K

ZombieLoad 2



- Meltdown-type attack
- Can read user and system data
  - Passwords, disk encryption keys etc.
- May 2019 Intel released microcode fix
- V2 works on Meltdown-safe CPUs
  - Exploits TSX to speculatively load, then intentionally causes conflict-abort
- Currently V2 is unfixed, according to paper authors

https://zombieloadattack.com/

Follow-up work

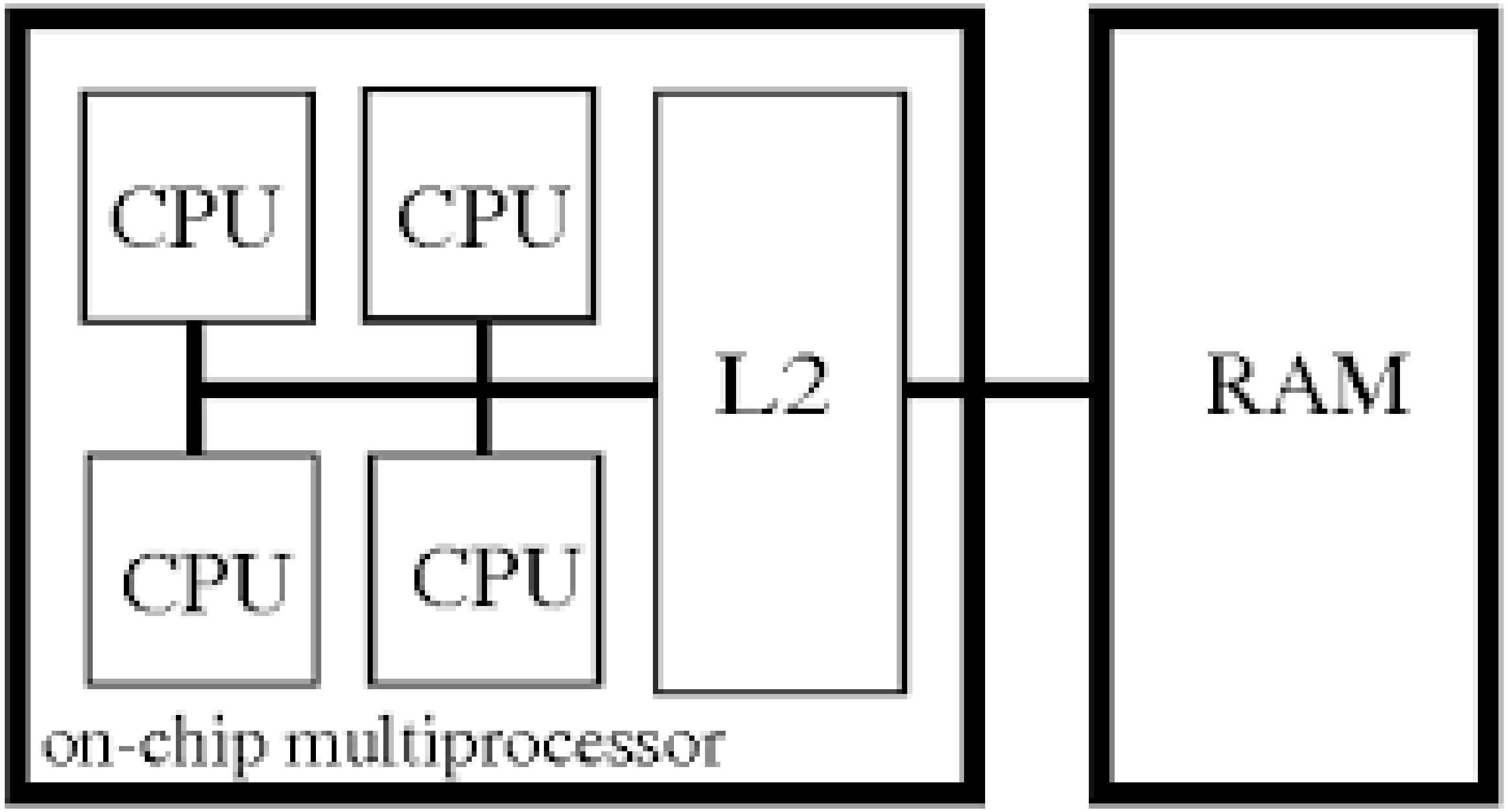
- 103 paper citations according to ieeexplore
- Gustavo Sousa, Alexandro Baldassin, **"FGSCM: A Fine-Grained Approach to Transactional Lock Elision"**, *Computer Architecture and High Performance Computing (SBAC-PAD) 2017 29th International Symposium on*, pp. 113-120, 2017.
- Seunghee Shin, James Tuck, Yan Solihin, **"Hiding the Long Latency of Persist Barriers Using Speculative Execution"**, *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 175, 2017.
- Milind Chabbi, Wim Lavrijsen, Wibe de Jong, Koushik Sen, John Mellor-Crummey, Costin Iancu, **"Barrier elision for production parallel programs"**, *ACM SIGPLAN Notices*, vol. 50, pp. 109, 2015.

Open Discussion

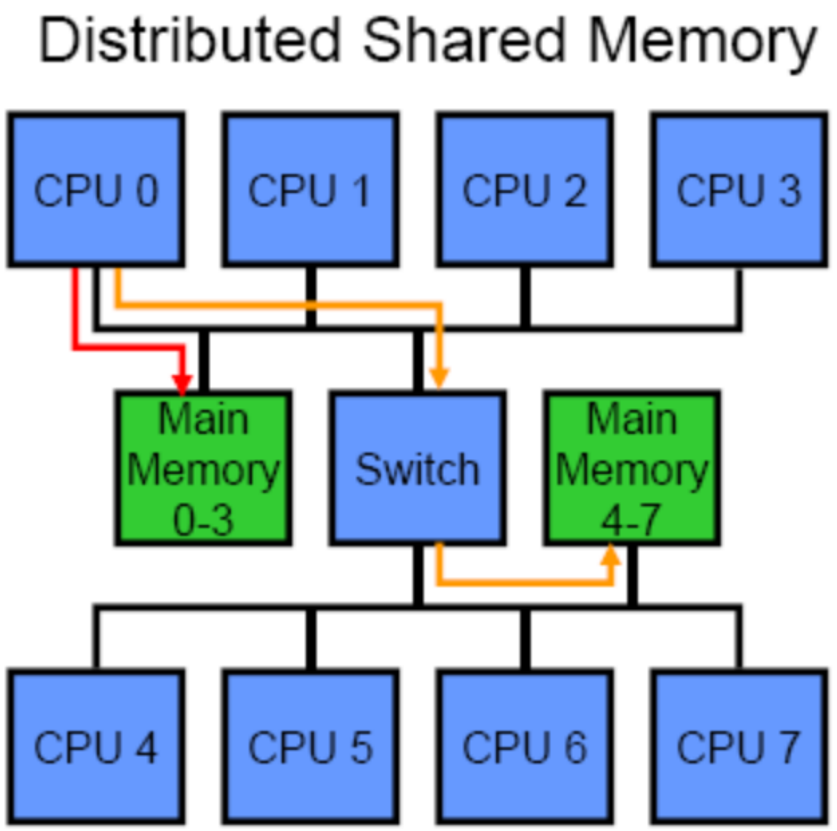
Discussion Starters

- Should Intel TSX be disabled until there is a fix?
- Do you see any downsides of SLE?
- Is there functionality in newer processors which makes implementation harder?
- Do you think hardware or transactional memory SLE implementations are better?
- Can you think of another way to elide locks?
- Are there non-lock synchronizations that could be elided?

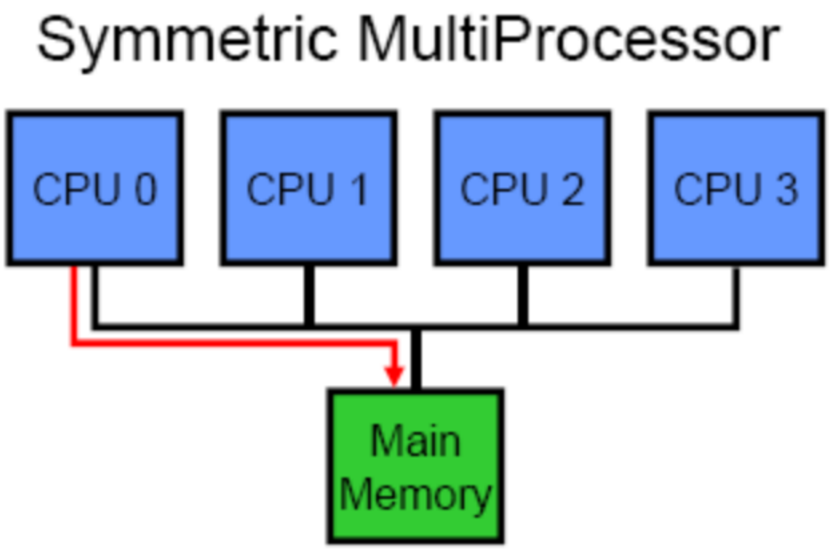
CMP



DSM vs SMP



Distributed Shared Memory
CPU 0
CPU 1
CPU 2
CPU 3
Main Memory 0-3
Switch
Main Memory 4-7
CPU 4
CPU 5
CPU 6
CPU 7



Symmetric MultiProcessor
CPU 0
CPU 1
CPU 2
CPU 3
Main Memory

Difference between SMP, DSM, CMP

“The SMP and DSM versions gain more than CMP because their large caches can hold the working set and thus have fewer read misses (and memory traffic) for the lock. For the CMP, the absence of a large cache hurts and thus there are more evictions of locks in clean state because the L1 suffers conflict and capacity misses.”