The Cost of Agreement in Synchronization

Philip Howard

Portland State University

May 12, 2009

- 4.77 Mhz 8088
- 64K RAM

- 2.53 Ghz Quad-core processor
- 4G RAM
- 6M L2 Cache

- 4.77 Mhz 8088
- 64K RAM

- 2.53 Ghz Quad-core processor
- 4G RAM
- 6M L2 Cache

Performance
vs.
Scalability

What is synchronization?

- Means of coordinating access to shared data
- Agreement

What makes synchronization slow?

- The need for agreement

Hypothesis

The scalability of synchronization techniques is determined by the extent to which they need to agree on something.

Research Process

- What does a synchronization technique need to agree on?
  - Analyze spinlock
- What's the cost for each of those components?
  - micro-benchmarks
- Can we use these results to predict performance/scalability of other synchronization techniques?
- What do these results tell us about future directions?

Talk outline

- Summarize micro-benchmark results
- Discuss higher level algorithms
  - Identify need-to-agree
  - Predict performance
  - Validate predictions
- Conclusions and ongoing work

Test system

- 16 processor system (4 quad-core Xeons)
- Private L1 cache
- 2-way shared L2 cache

Test Process

- Count operations within a timing window
- Report total operations for all threads
- Report average of 16 runs. Error bars show 90% confidence interval on value

How do we synchronize?

```
spinlock(int *lock)
{
   while ( *lock == LOCKED )
   {}

   *lock = LOCKED;
}
```

How do we synchronize?

Compare And Swap (CAS)

```
spinlock(int *lock)
{
   while ( ! CAS(lock, UNLOCKED, LOCKED)  )
   {}
}
```

Instruction level need-to-agree

- All the processors need to agree on who gets to do the swap
- Enforced in hardware
- How expensive is it?

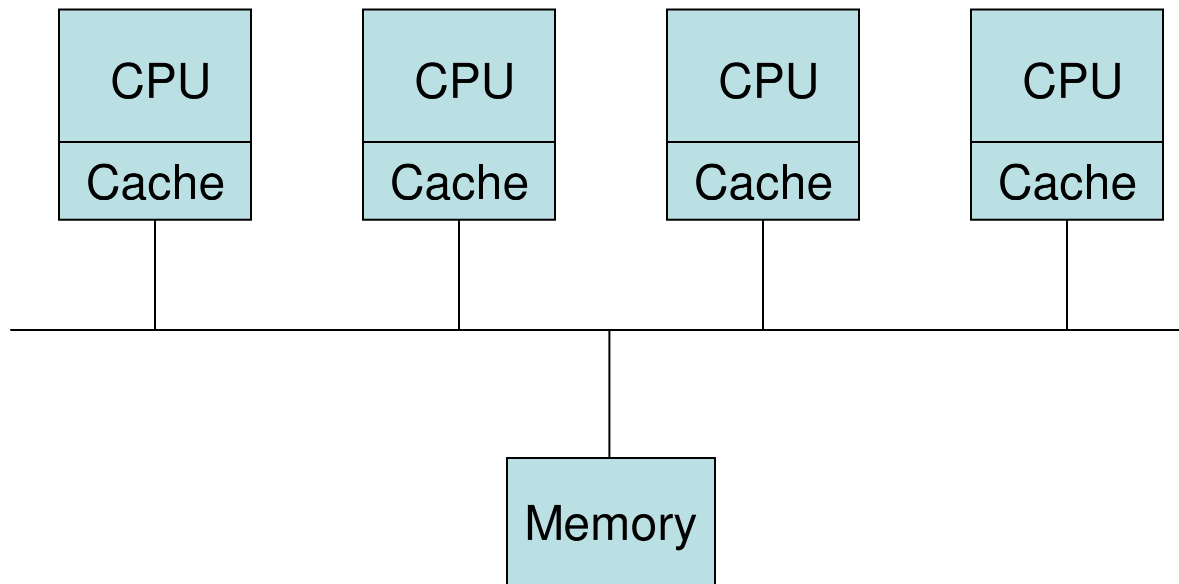
- Affects performance, but not scalability

Memory hierarchy need-to-agree

- What's the value of the lock variable?
- Where is it read from?
- What difference does that make?

The UMA Myth

Uniform Memory Access



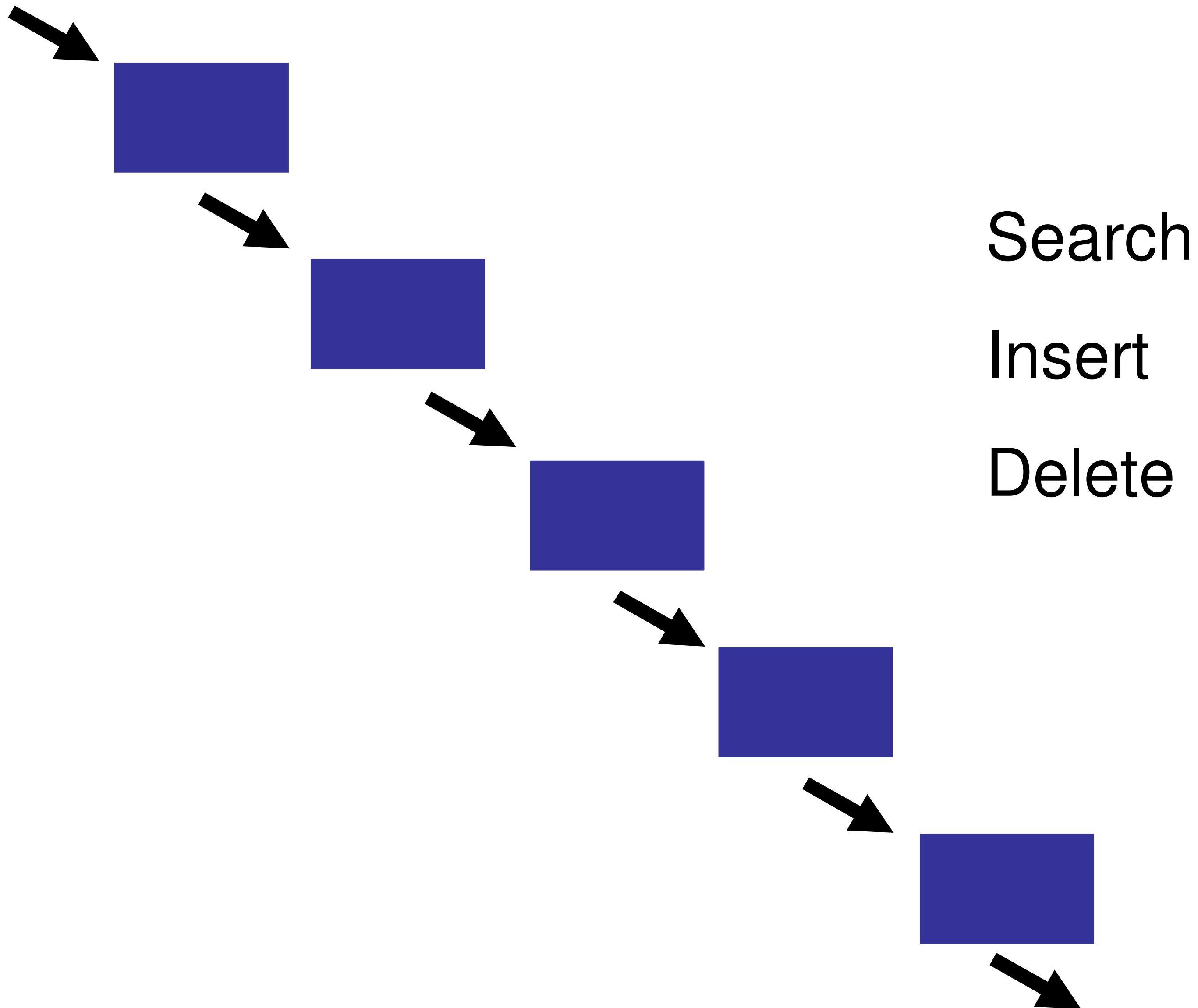
What did the micro-benchmarks tell us?

- Memory hierarchy need-to-agree kills scalability
- Instruction level need-to-agree hurts performance, but not scalability

Talk outline

- Summarize micro-benchmark results
- Discuss higher level algorithms
  - Identify need-to-agree
  - Predict performance
  - Validate predictions
- Conclusions and ongoing work

Sorted Linked List



Pessimistic Approaches

- All other threads are evil—mutual exclusion
- All threads touching my node are evil—fine grained locking
- All writers are evil—Reader Writer Locking (RWL)

Need-to-agree Observations for Reader/Writer Locking

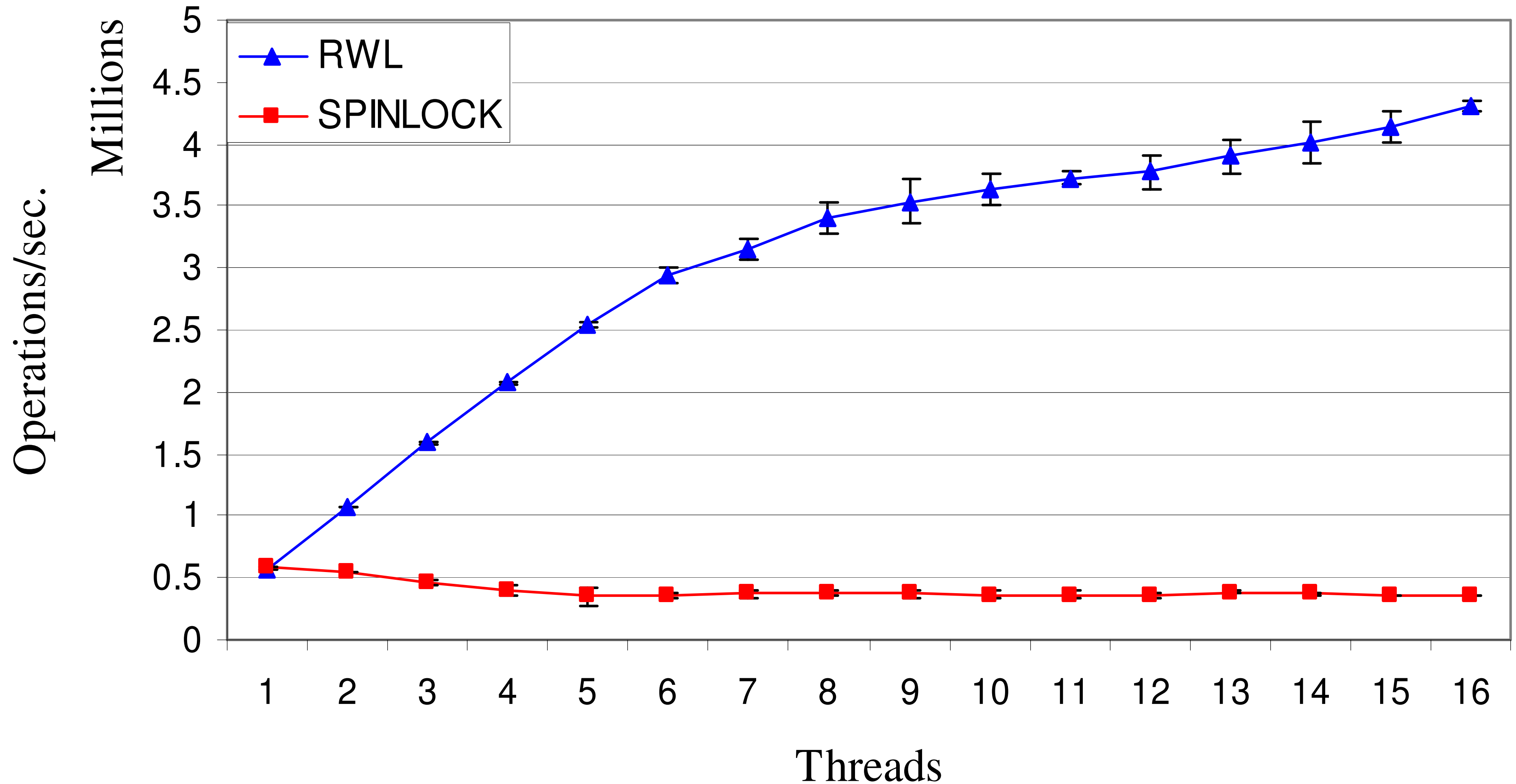
- Instruction level – two atomic operations per lock/unlock
- Algorithmic – enforce semantics of RWL
- Memory hierarchy – share a lock variable

Mellor-Crummey and Scott's reader preference algorithm

Observations/Predictions

- Because of algorithmic need-to-agree, writers are not expected to scale
- Because of memory hierarchy need-to-agree, readers are not expected to scale

Scalability of RWL readers vs. Spinlock

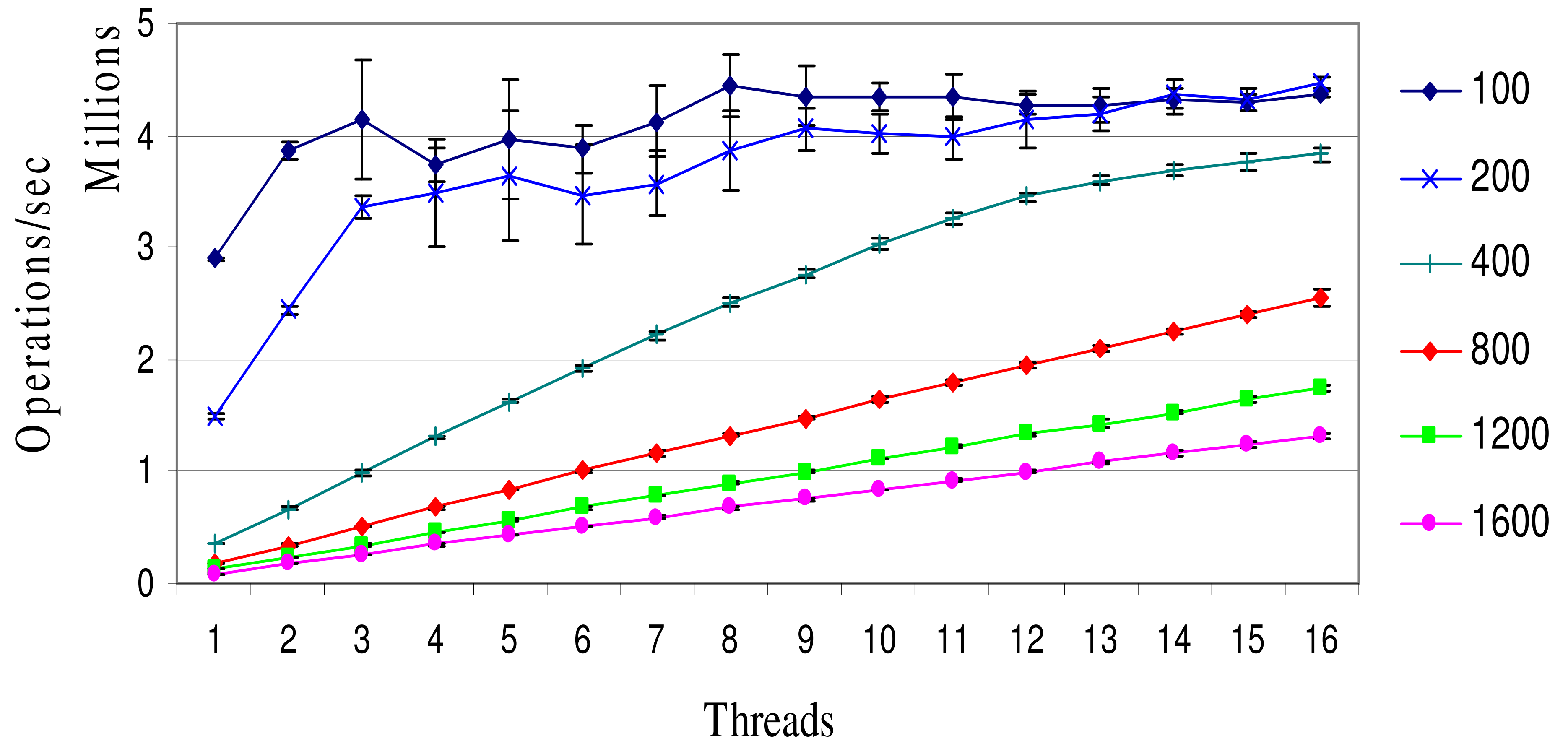


Mellor-Crummey and Scott algorithm

Amdahl's Law
Sequential Bottle Necks

```
while (timing_window_open)
{
  lock();
  do_work();
  unlock();
}
```

RWL read-side scalability



Optimistic Approaches

- Herlihy suggested that maybe most threads aren’t evil after all
- Assume the operation will complete without conflict and check assumption just before committing

Non-Blocking Synchronization (NBS)

1. Save the state of the data structure

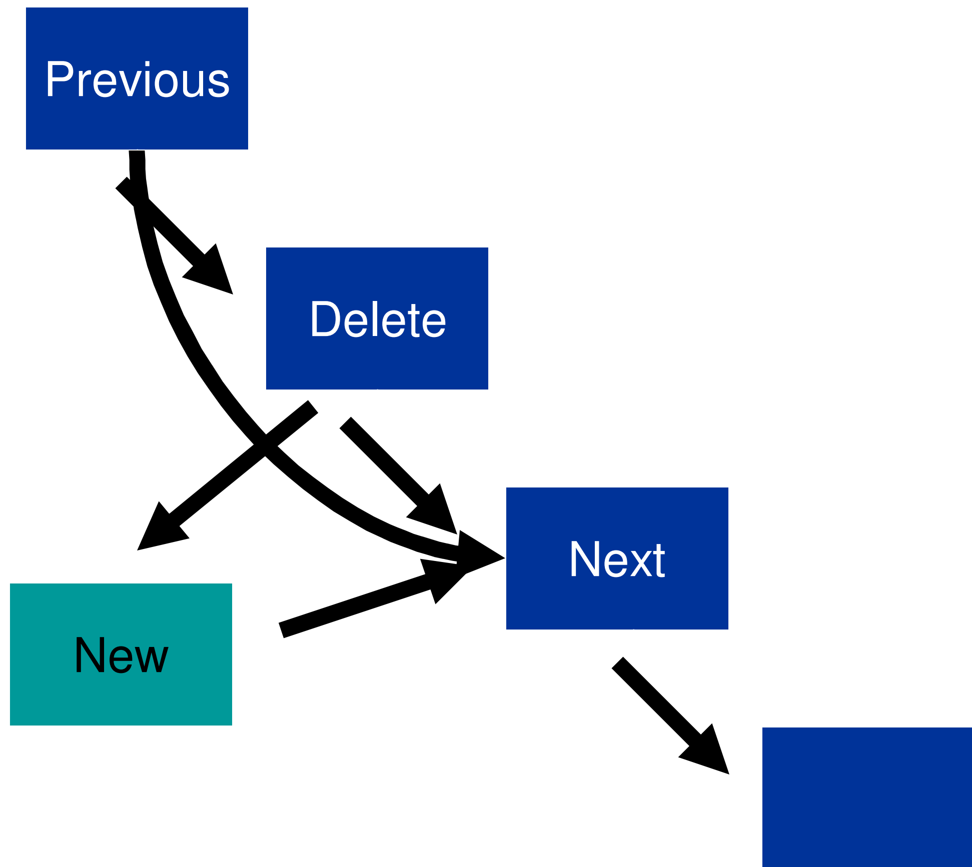
2. Make a change in private memory

3. Check state: Commit if no change; Rollback if changed

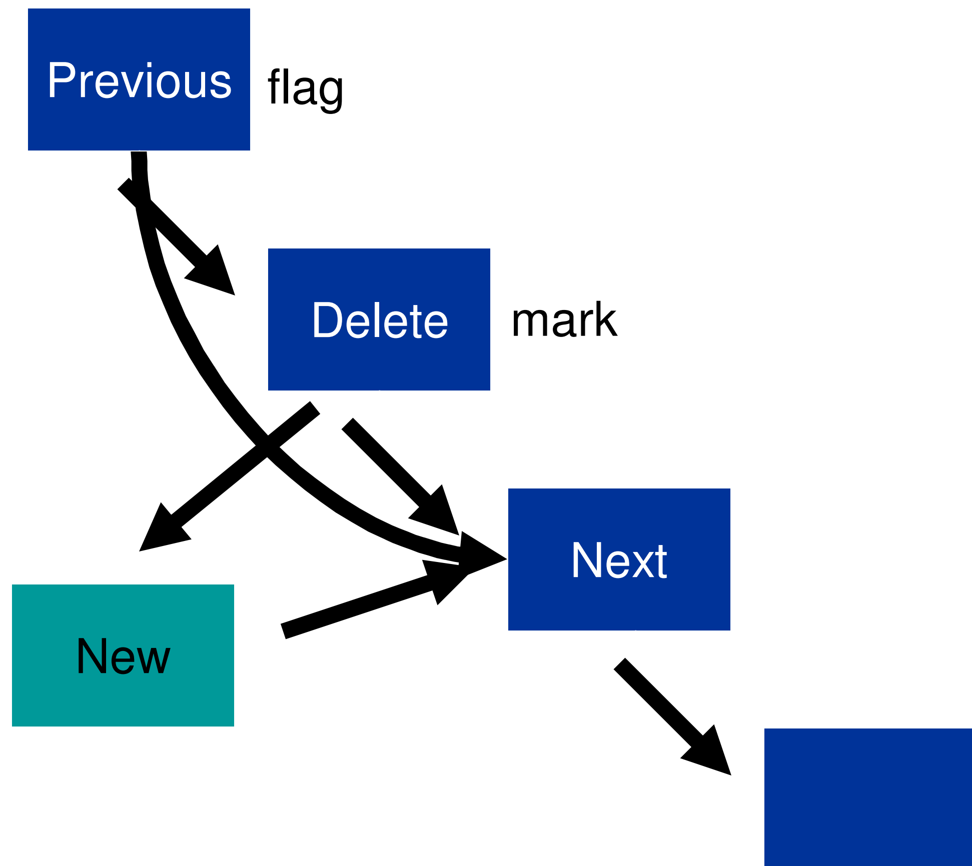
Pessimistic vs. Optimistic

- Pessimistic agrees at the beginning that it's your turn
- Optimistic agrees at the end that it's OK to commit
- Pessimistic wastes effort in spinning
- Optimistic wastes effort in retries on contention

NBS Linked List
Delete



NBS Linked List
Delete



Fomitchev and Rupert's algorithm

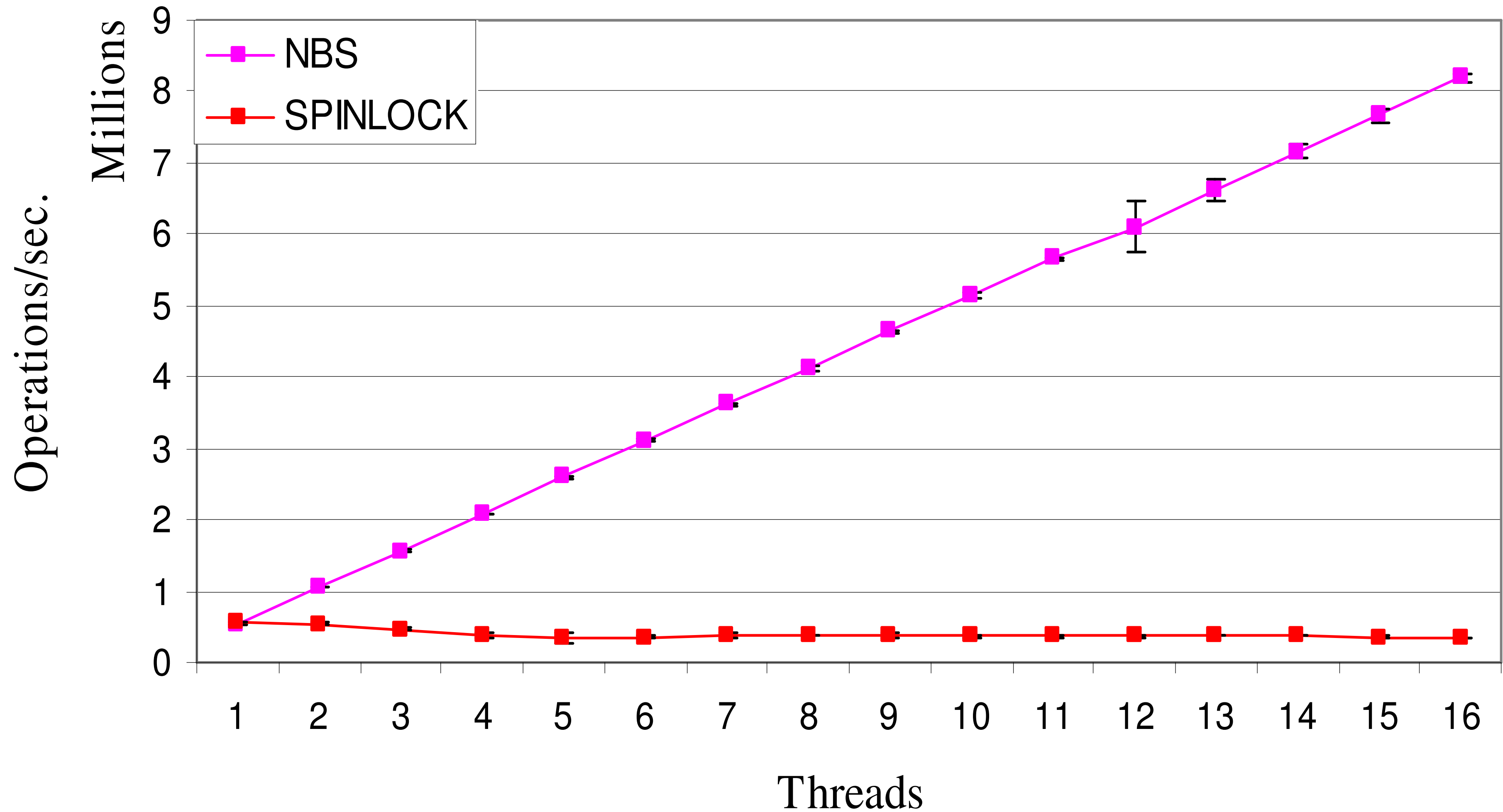
NBS need-to-agree

- Reads in the absence of deletes have no need-to-agree
- Inserts in the absence of updates require one atomic instruction and update one shared location
- Deletes in the absence of contention require three atomic instructions and update two shared locations

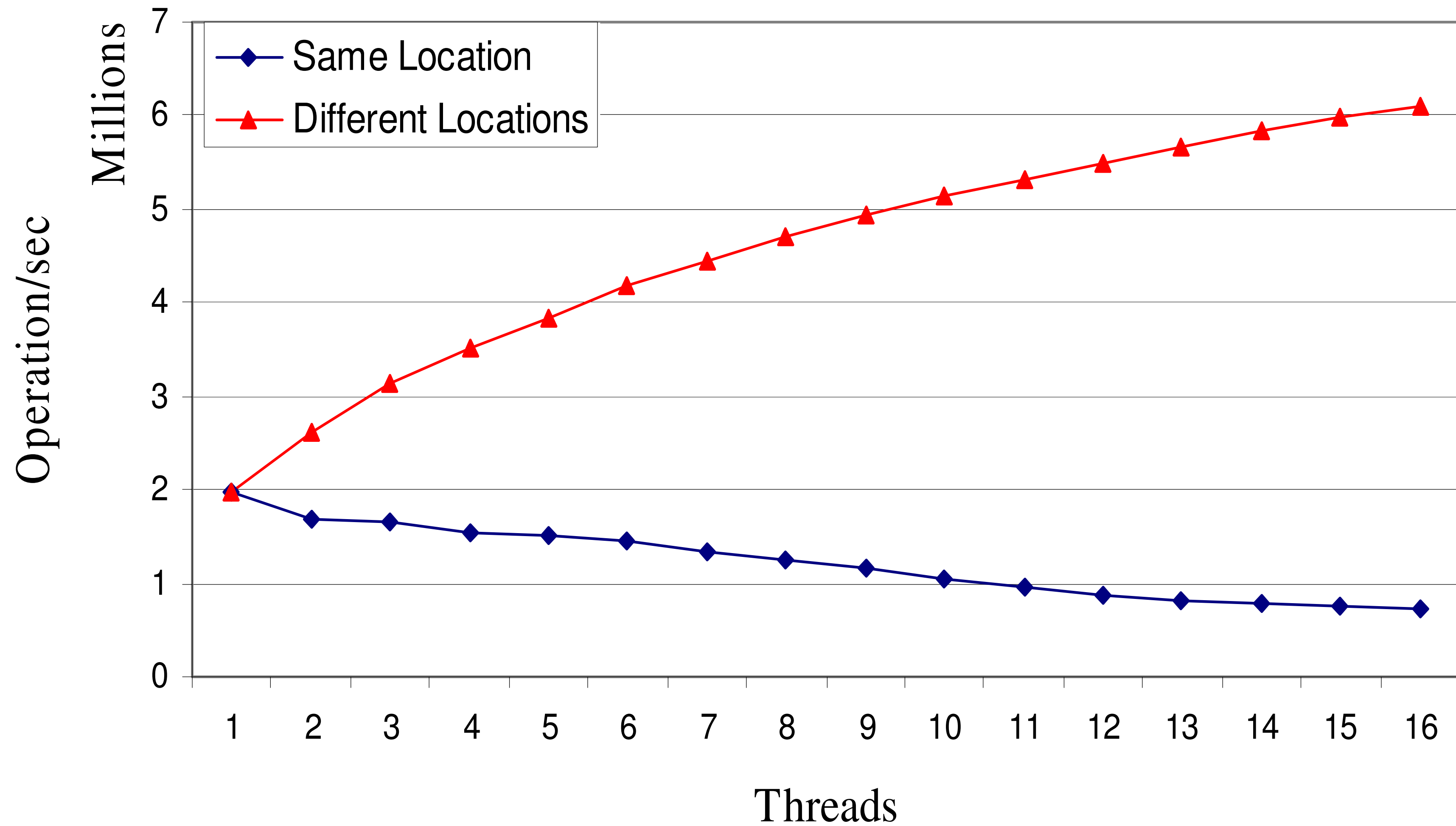
NBS Predictions

- Reads in the absence of updates should scale
- Deletes will interfere with the scalability of reads
- Concurrent updates of different locations should scale
- Concurrent updates of the same location will not scale

NBS read-side scalability



NBS writer scalability



Deterministic

- Unimpeded readers
- Reduced need to agree
  - no agreement between readers and writers
  - don't require agreement on order of operations

Relativistic Programming

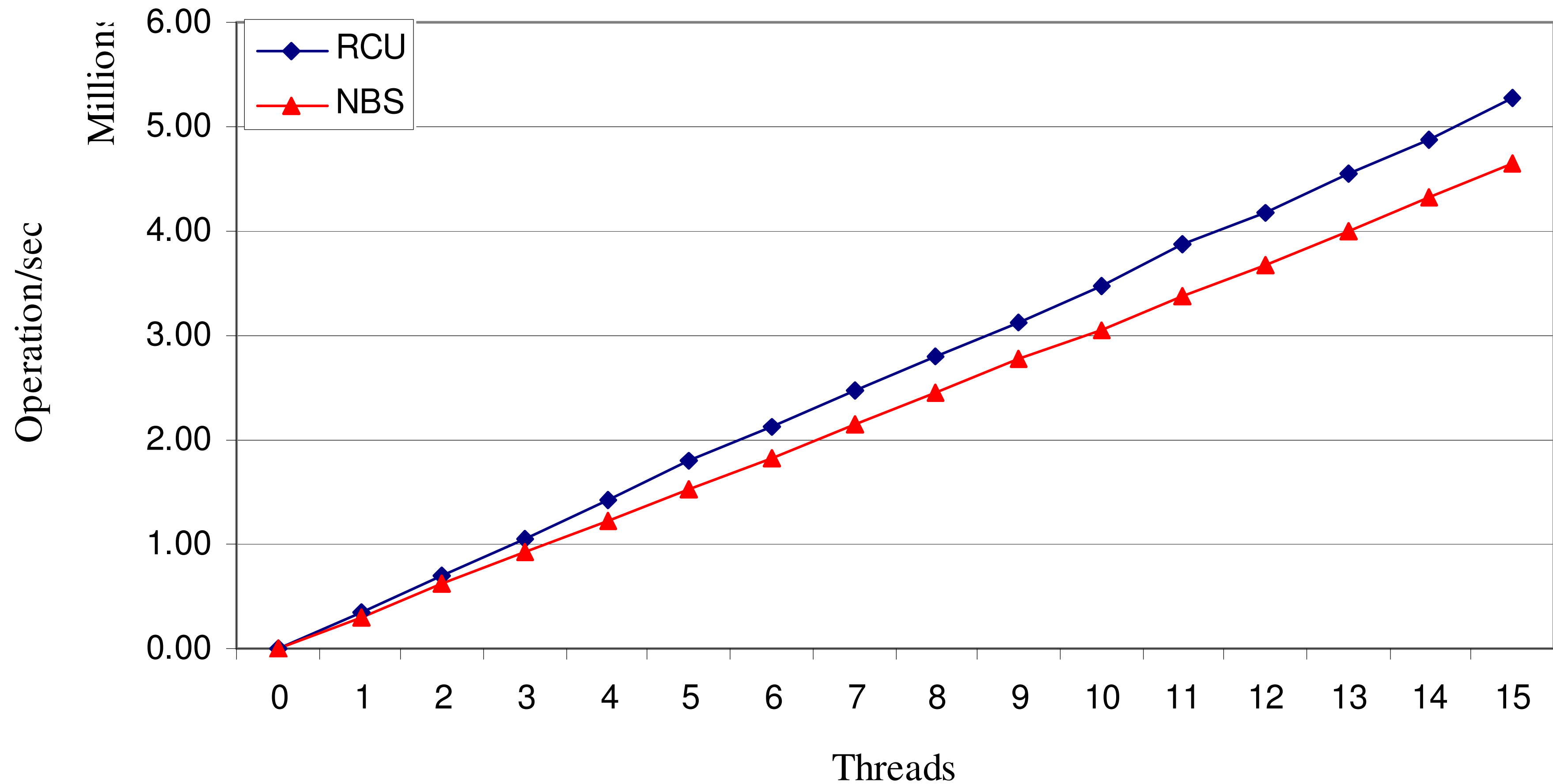
Read/Copy-Update (RCU)

NBS vs. RCU

NBS: Deletes will interfere with the scalability of reads

RCU: No agreement between readers and writers

Read-side scalability in the presence of a writer



Conclusions (part 1)

Need-to-agree is a useful predictor of an algorithm's performance and scalability

Conclusions (part 2)

- **Instruction level** need-to-agree limits performance, but not scalability
- **Algorithmic** need-to-agree limits scalability
  - limits parallelism (spinlock)
  - increases workload (NBS)
- **Memory hierarchy** need-to-agree limits scalability

Conclusions (part 3)

- Algorithms that purport to allow parallelism may not
  - Implications for system calls
- We need to reexamine the memory abstraction that is supplied to programmers
  - UMA vs. NUMA

Ongoing and future work

- Benchmarks to test other predictions
- Benchmarks for fine grained locking
- How does shared cache affect need-to-agree?

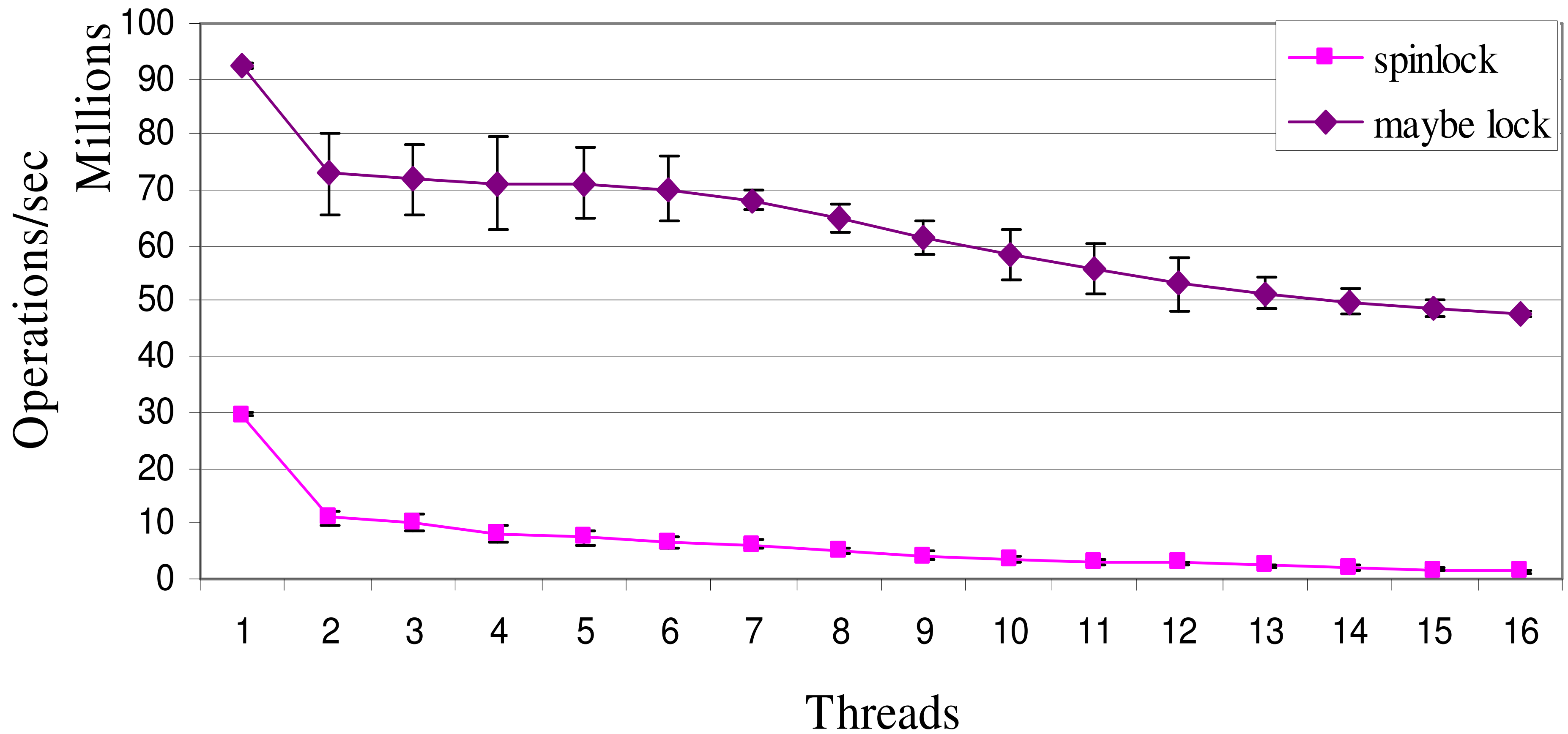
- Develop new relativistic techniques
  - Are they also performance equivalent to NBS?

- What is the right abstraction for many-core?

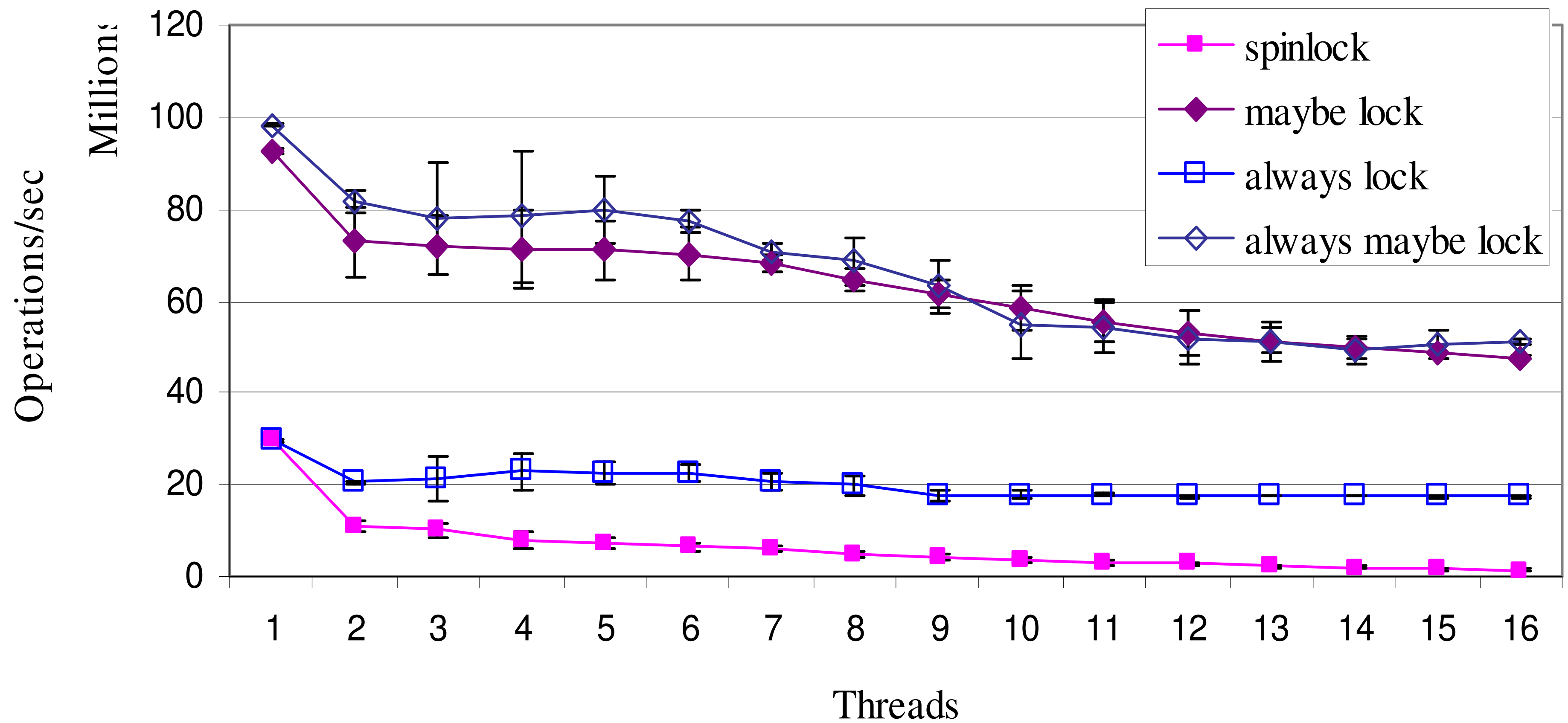
Contributions

- Identified need-to-agree as a useful metric in predicting performance and scalability
- Identified different forms of agreement
- Found an unexpected similarity between RCU and NBS

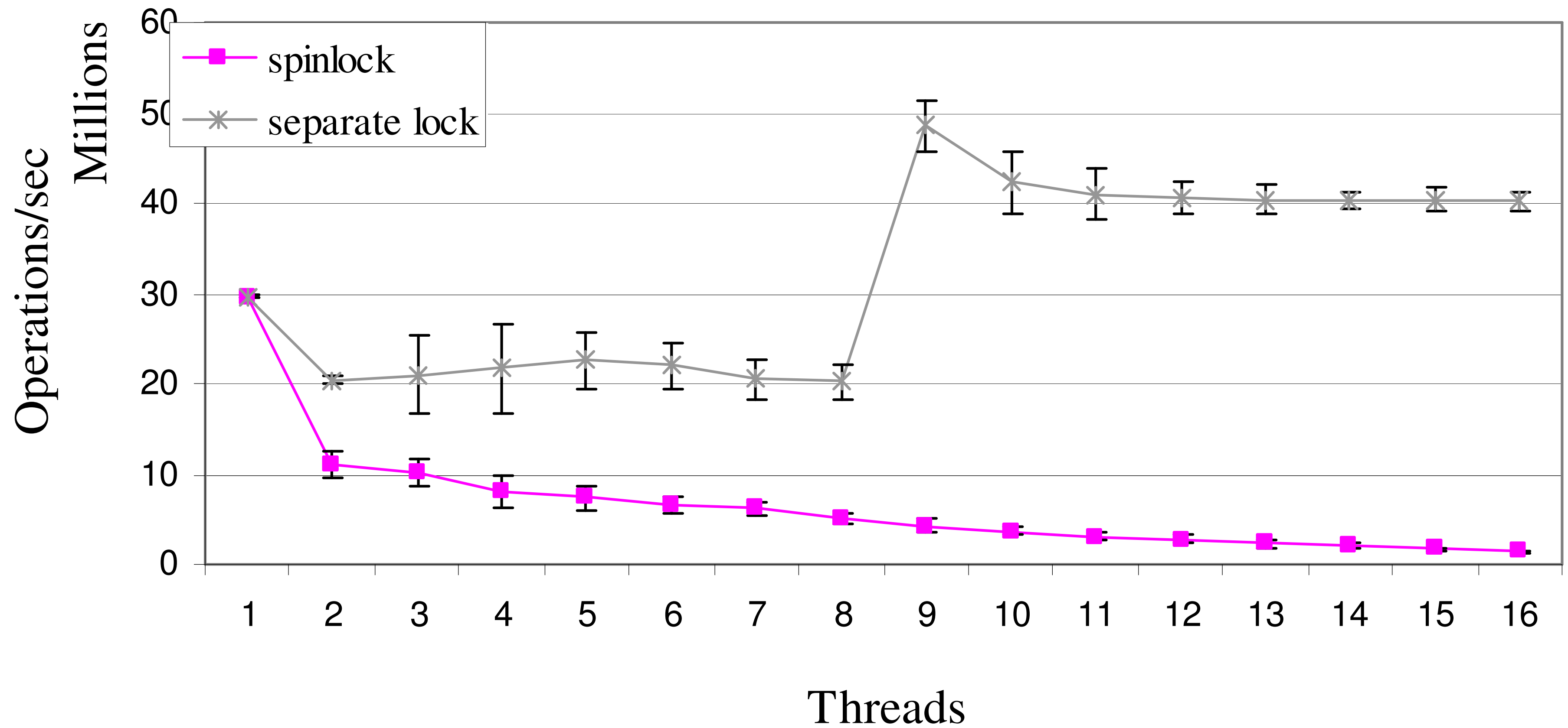
Scalability of *spinlock* vs. *maybe lock*



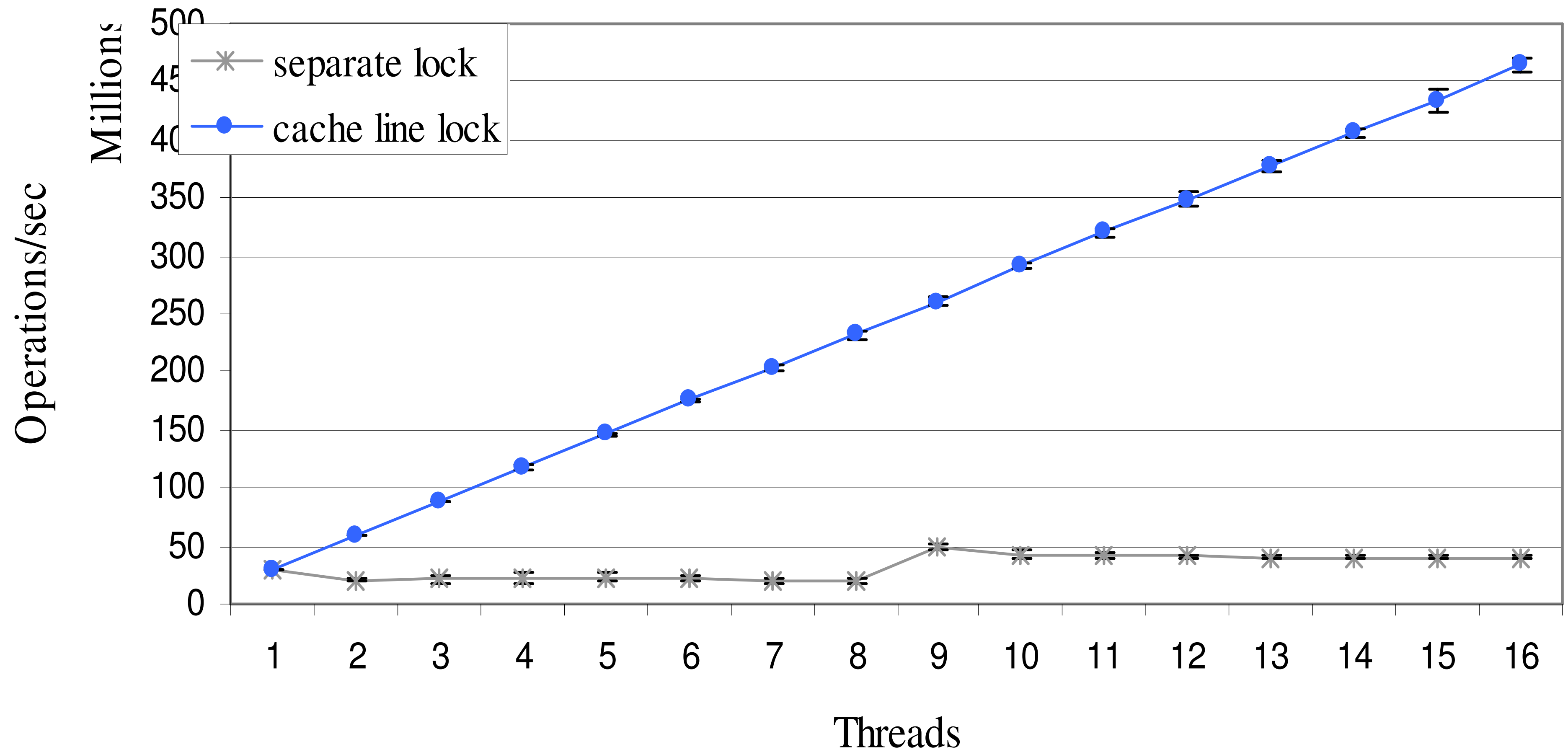
Scalability of *spin*, *always*, and *maybe lock*



Scalability of *separate lock*



Scalability of *cache line lock*



Read-only Scalability