Abstraction, Reality Checks, and RCU

Paul E. McKenney
IBM Beaverton

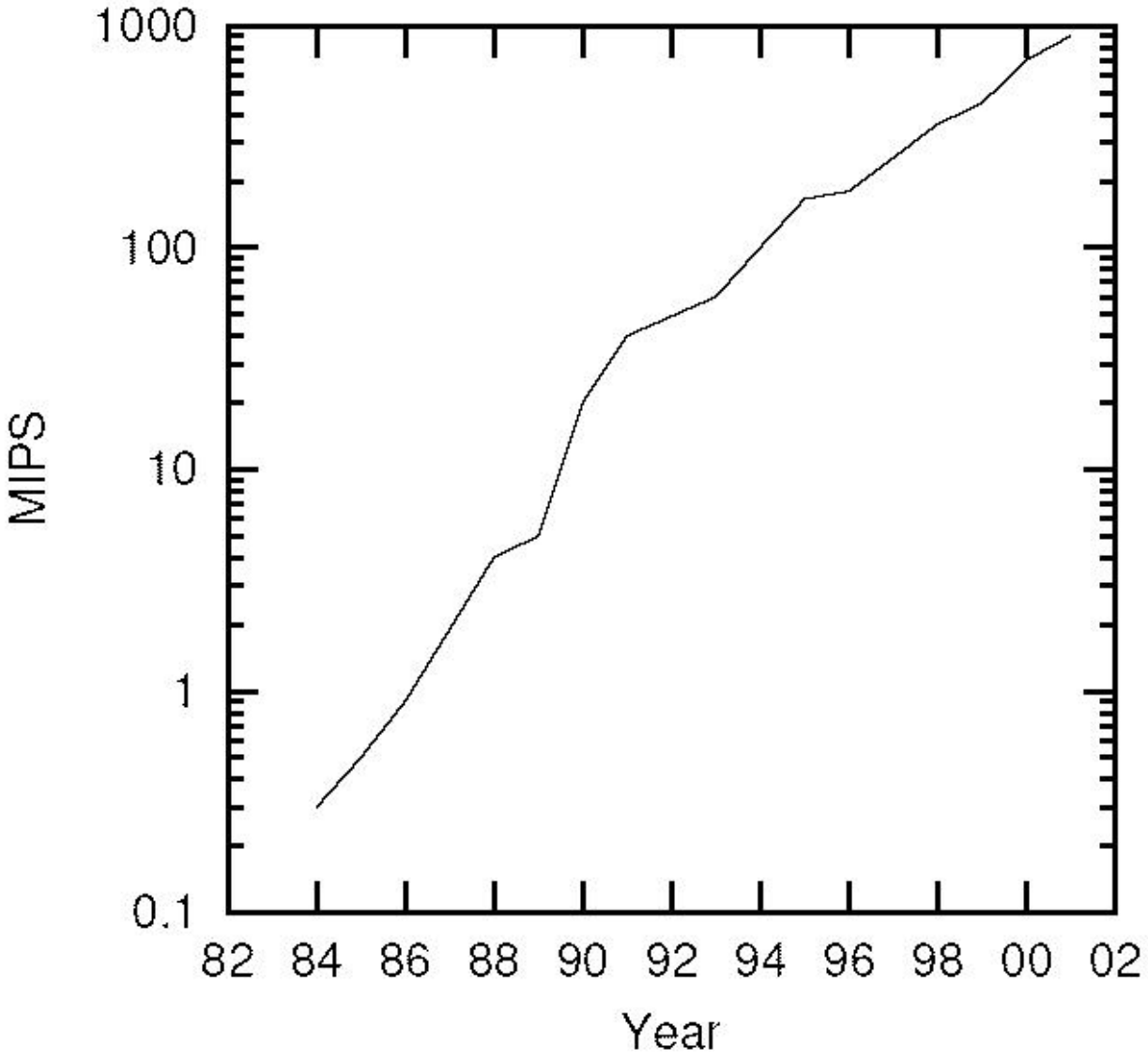
University of Toronto Cider Seminar
July 26, 2005

Overview

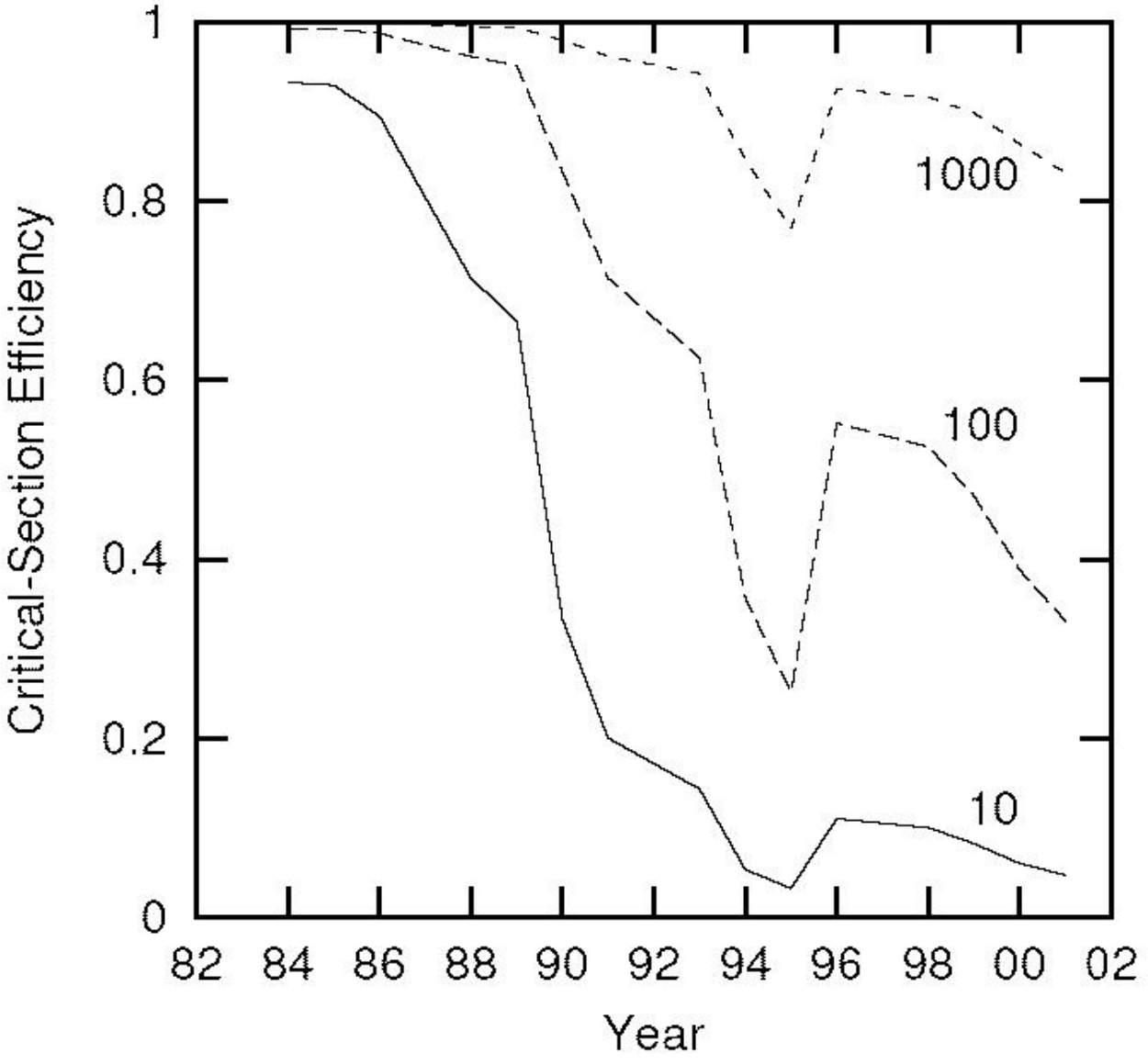
- Moore's Law and SMP Software
- Non-Blocking Synchronization (NBS)
- Read-Copy Update (RCU)
- Summary

Moore's Law and SMP Software

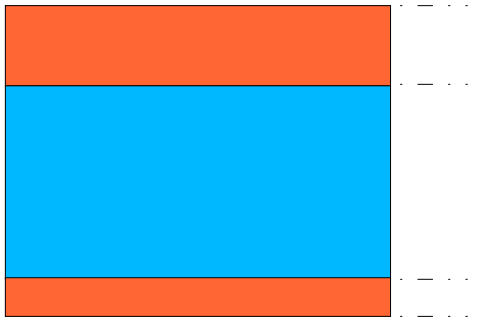
Instruction Speed Increased



Synchronization Speed Decreased



Critical-Section Efficiency



Lock Acquisition (T_a)

Critical Section (T_c)

Lock Release (T_r)

$$Efficiency = \frac{T_c}{T_c + T_a + T_r}$$

Assuming negligible contention and no caching effects in critical section!

Reality Check #1: this is ***not*** your father's CPU!!!

Instruction/Pipeline Costs on a 8-CPU 1.45GHz PPC

Operation	Nanoseconds
Instruction	0.24
Clock Cycle	0.69
Atomic Increment	42.09
Cmpxchg Blind Cache Transfer	56.80
Cmpxchg Cache Transfer and Invalidate	59.10
SMP Memory Barrier (eieio)	75.53
Full Memory Barrier (sync)	92.16
CPU-Local Lock	243.10

Visual Demonstration of Latency

SMP MB (eieio):75.53 ns, 314.7 insts
Full MB (sync): 92.16 ns, 384.5 insts

Each nanosecond represents
up to about four instructions

What is Going On? (1/3)

- Taller memory hierarchies
  - Memory speeds have not kept up with CPU speeds
    - 1984: no caches needed, since instructions slower than memory accesses
    - 2005: 3-4 level cache hierarchies, since instructions orders of magnitude faster than memory accesses
- Synchronization requires consistent view of data across CPUs, in other words, CPU-to-CPU communication
  - Unlike normal instructions, synchronization operations tend not to hit in top-level cache
  - Hence, they are orders of magnitude slower than normal instructions because of memory latency

What is Going On? (2/3)

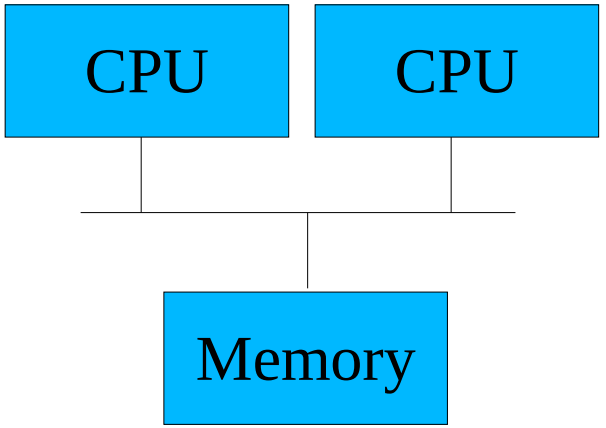
- Longer pipelines
  - 1984: Many clocks per instruction
  - 2005: Many instructions per clock – 20-stage pipelines
- Modern super-scalar CPUs execute instructions out of order in order to keep their pipelines full
  - But musn't reorder a critical section before its lock!!!
- Therefore, synchronization operations must stall the pipeline, decreasing performance

What is Going On? (3/3)

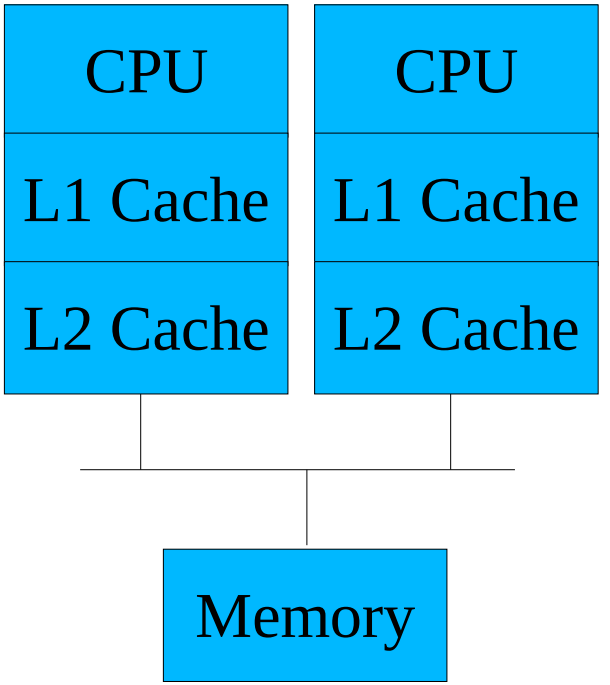
- 1984: The main issue was lock contention
- 2005: Even if lock contention is eliminated, critical-section efficiency must be addressed!!!
  - Even if the lock is *always* free when acquired, performance is seriously degraded
  - Some hardware guys tell me that this will all soon be better...
    - But I will believe it when I see it!!!

What is Going On?

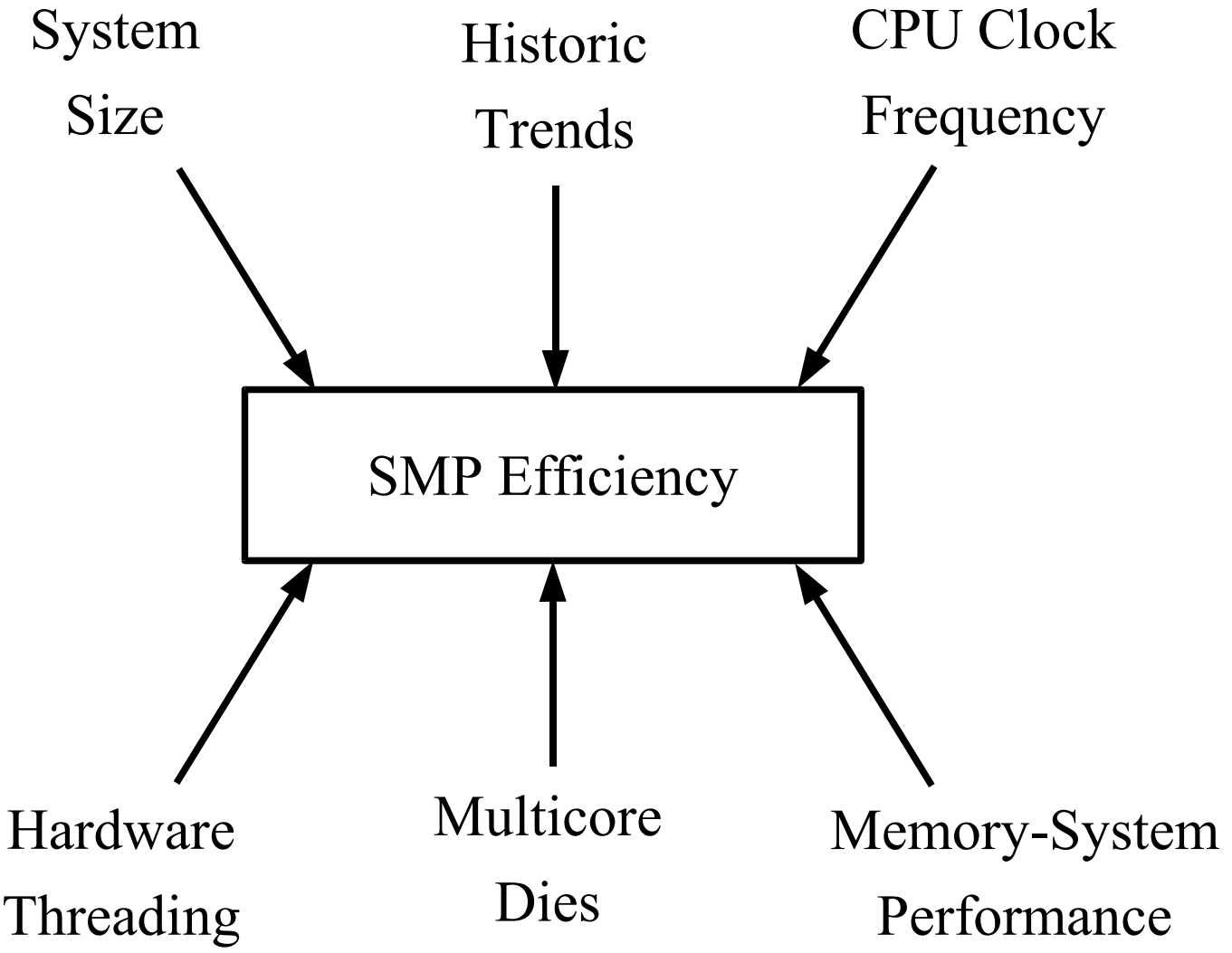
1984



2005



Forces Acting on SMP Efficiency



Locking Performance

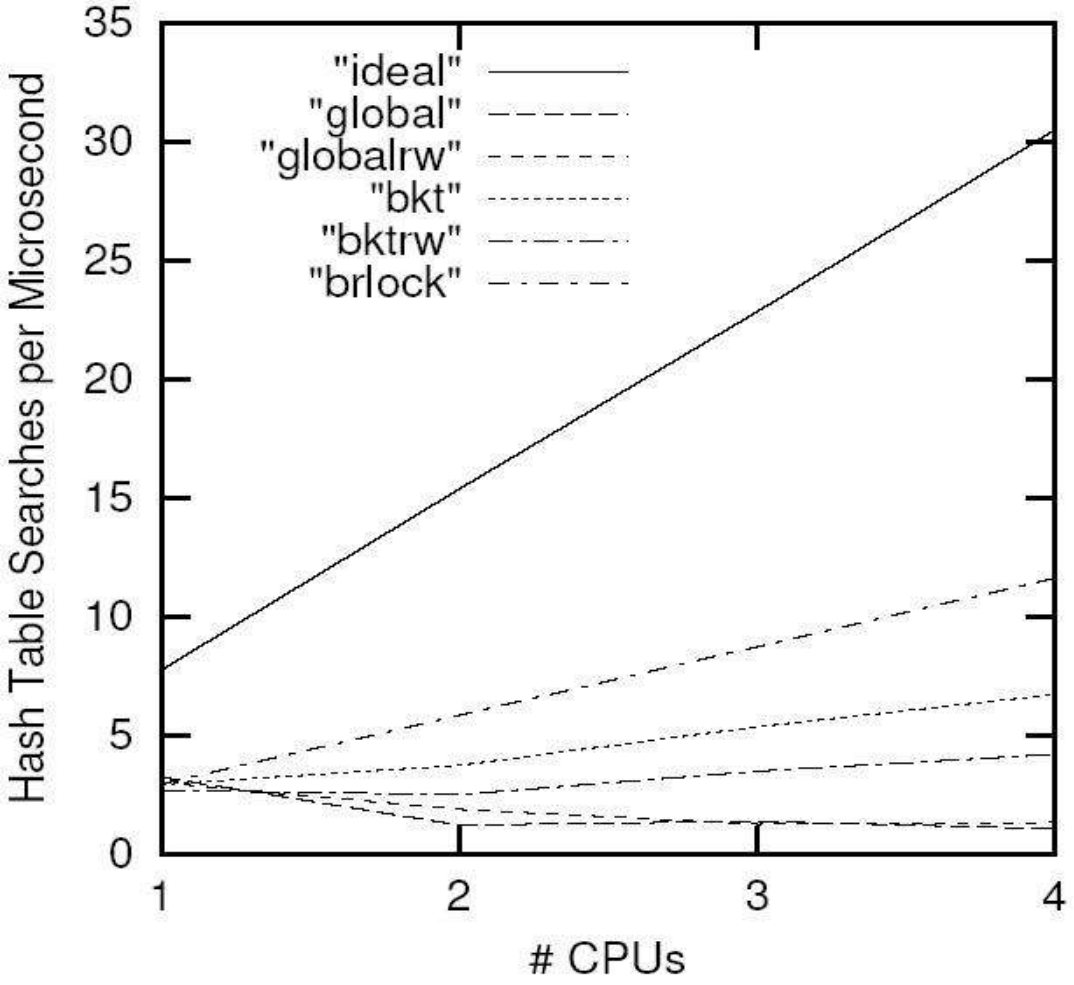
Performance Comparison: What Benchmark to Use?

- Focus on operating-system kernels
  - Many read-mostly hash tables
- Hash-table mini-benchmark
  - Dense array of buckets
  - Doubly-linked hash chains
  - One element per hash chain
    - You do tune your hash tables, don't you???

How to Evaluate Performance?

- Mix of operations:
  - Search
  - Delete followed by reinsertion: maintain loading
  - Random run lengths for specified mix
    - (See thesis)
- Start with pure search workload (read only)
- Run on 8-CPU 1.45GHz PPC system

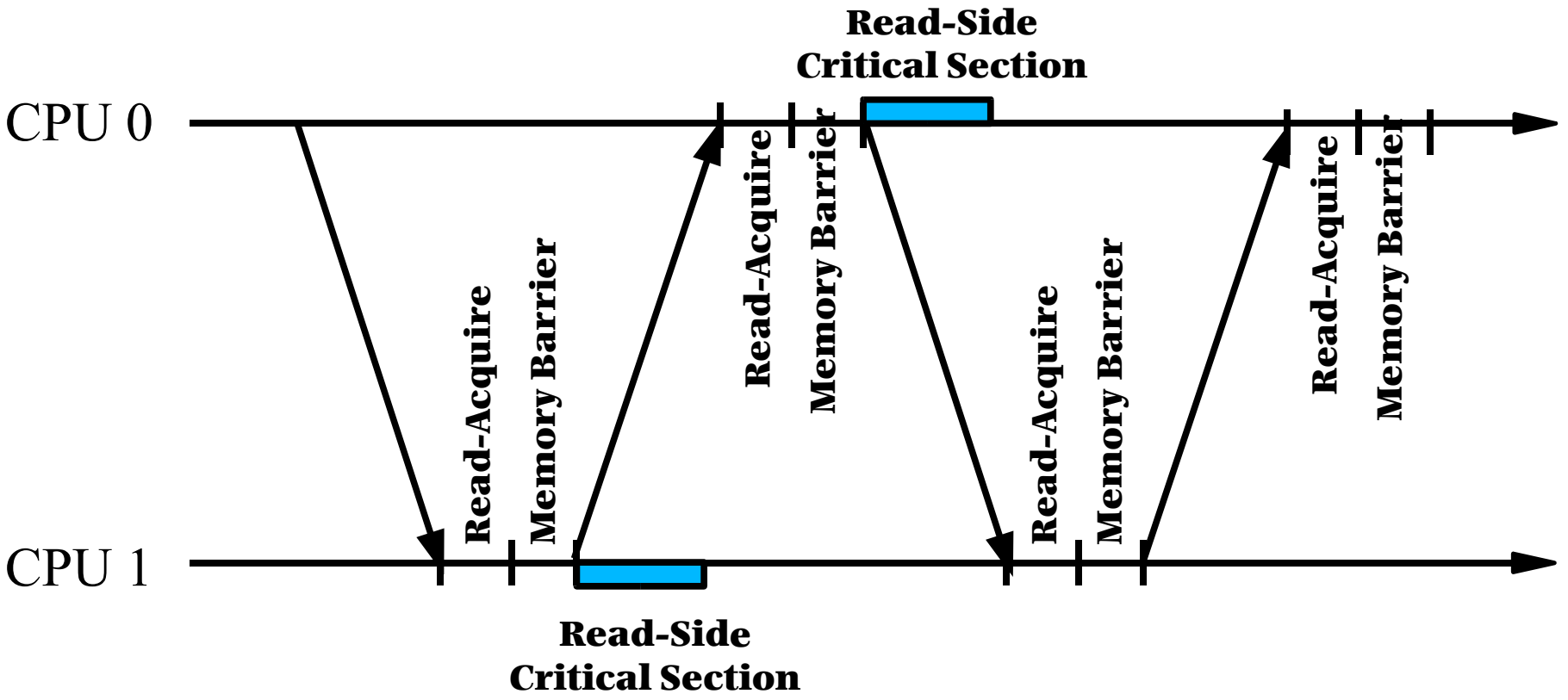
Locking Performance



Reality Check #2:

Extra CPUs not buying much! Note: workload fits in cache.

Do Not Use rwlock_t for Short Read-Side Critical Sections



Reality Check #3: Parallel reader access isn't.

Non-Blocking Synchronization (NBS)

What About Non-Blocking Synchronization?

- What is non-blocking synchronization (NBS)?
  - Roll back to resolve conflicting changes instead of spinning or blocking
  - Use atomic instructions to hide complex updates behind a single commit point
    - Readers and writers use atomic instructions such as compare-and-swap or LL/SC
- Simple "NBS" algorithms in heavy use
  - Atomic-instruction-based algorithms

Why Not NBS All The Time?

Operation	Nanoseconds
Instruction	0.24
Clock Cycle	0.69
Atomic Increment	42.09
Cmpxchg Blind Cache Transfer	56.80
Cmpxchg Cache Transfer and Invalidate	59.10
SMP Memory Barrier (eieio)	75.53
Full Memory Barrier (sync)	92.16

Reality check #4: the 1980s ended a ***long*** time ago...

When to Use NBS?

- Simple NBS algorithm is available
  - Split counters (strictly speaking, only by 1)
    - More on this later...
  - Simple queue/stack management
  - Especially if NBS constraints may be relaxed!
- Workload is update-heavy, but simple
  - NBS's use of atomic instructions and memory barriers not causing gratuitous performance pain
  - Complexity of “Macho NBS” avoided

NBS Constraints

- Progress guarantees in face of task failure
  - Everyone makes progress: wait free
  - Someone makes progress: lock free
  - Someone makes progress in absence of contention: obstruction free
  - ***Some*** progress, but...
- Linearizability
  - Everyone agrees on all intermediate states
- Reality check #5:
  - Both constraints are usually irrelevant!!!

How Can Progress Guarantees ***Possibly*** Be Irrelevant???

- Failure due to software bug
  - What fraction of software bugs are fail-stop?
- "Failure" due to preemption/interrupt
  - Scheduler-conscious synchronization
    - Available in all commercial Unix-like systems
    - Including Linux, AIX, Solaris, HP-UX, DYNIX/ptx, ...
- "Failure" due to page fault
  - It is 2005. Over-provision memory. Get over it.
  - If the page is really nonresident, everyone faults!
- Production FT systems use locking

How Can Linearizability ***Possibly*** Be Irrelevant???

- By design
  - Linearizability implies dependencies
  - Dependencies are expensive in today's systems
  - Why add gratuitous dependencies???
    - Performance optimization ***avoids*** dependencies
- By nature
  - How can you tell which of two unrelated events occurred first?
  - Why would an application care???

Linearizability Example

- Linearizable Add:

```
atomic_add(&ctr, v);
```

- Linearizable Value:

```
return (ctr);
```

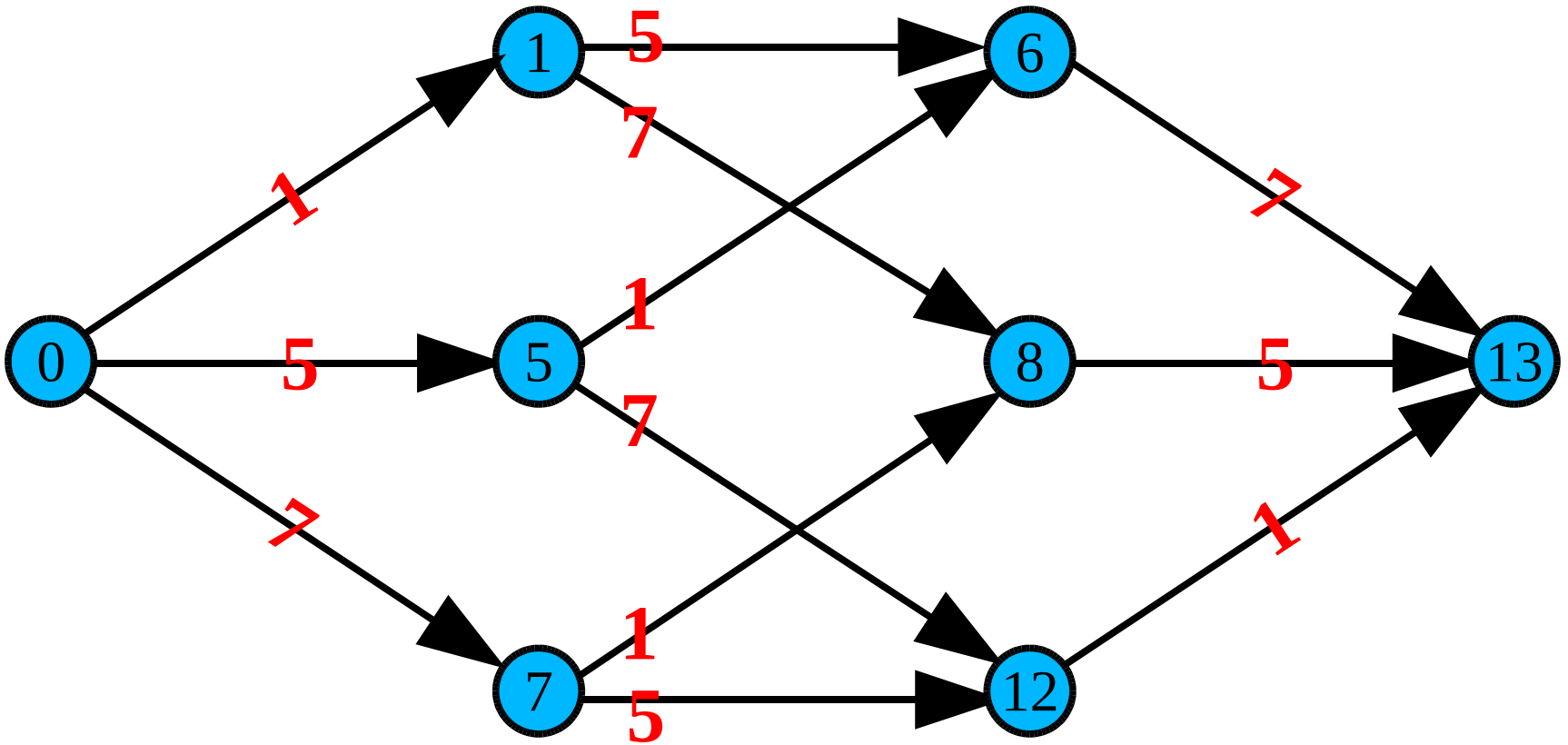
- Laissez-Faire Add:

```
__get_cpu_var(ctr)++;
```

- Laissez-Faire Value:

```
for_each_cpu(cpu) {
    sum += per_cpu(ctr,cpu);
}
return (sum);
```

Friendly Advice: Tolerate Dissent



NBS Summary

- Use it where it makes sense
  - Simple update-heavy data structures
  - Use locking for complex update-heavy data structures: scheduler-conscious synchronization
- Relax NBS forward-progress & linearizability constraints when it makes sense
  - Most of the time...
- Why do hard things the hard way???

Read-Copy Update (RCU)

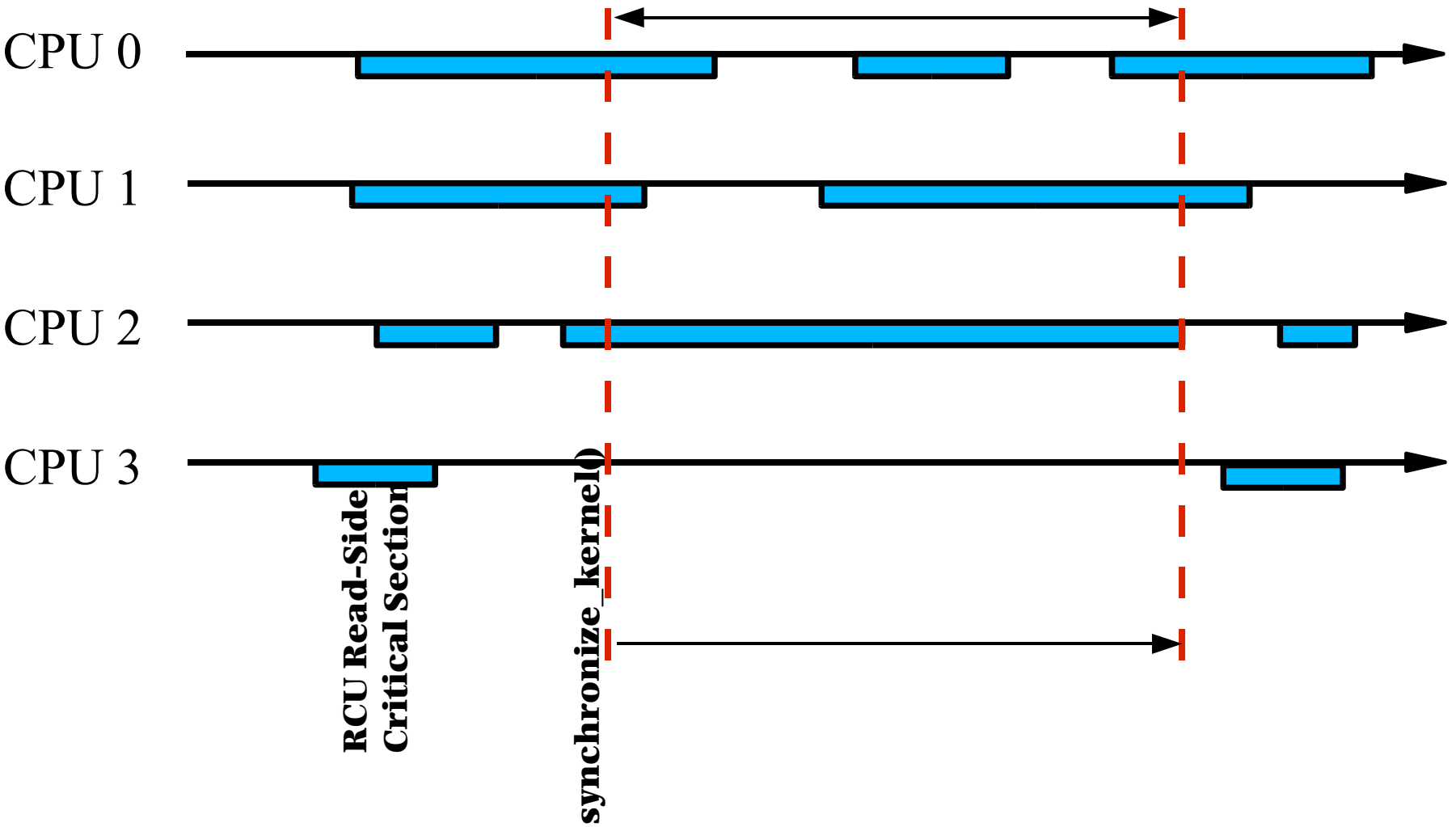
What is Synchronization?

- Mechanism ***plus coding convention***
  - Locking: must hold lock to reference or update
  - NBS: must use carefully crafted sequences of atomic operations to do references and updates
  - RCU coding convention:
    - Must define "quiescent states" (QS)
      - e.g., context switch in non-CONFIG_PREEMPT kernels
    - QSes must not appear in read-side critical sections
    - CPU in QSes are guaranteed to have completed all preceding read-side critical sections
  - RCU mechanism: "lazy barrier" that computes "grace period" given QSes.

RCU Fundamental Primitives

- rcu_read_lock() & rcu_read_unlock()
  - Demark RCU read-side critical section.
  - Zero overhead in non-preemptive environment.
- synchronize_rcu()
  - Wait until all pre-existing RCU read-side critical sections complete.
  - Subsequently started RCU read-side critical sections not waited for.
  - See next slide...
- call_rcu(): callback form of synchronize_rcu()
  - AKA “continuation” or “asynchronous” form.

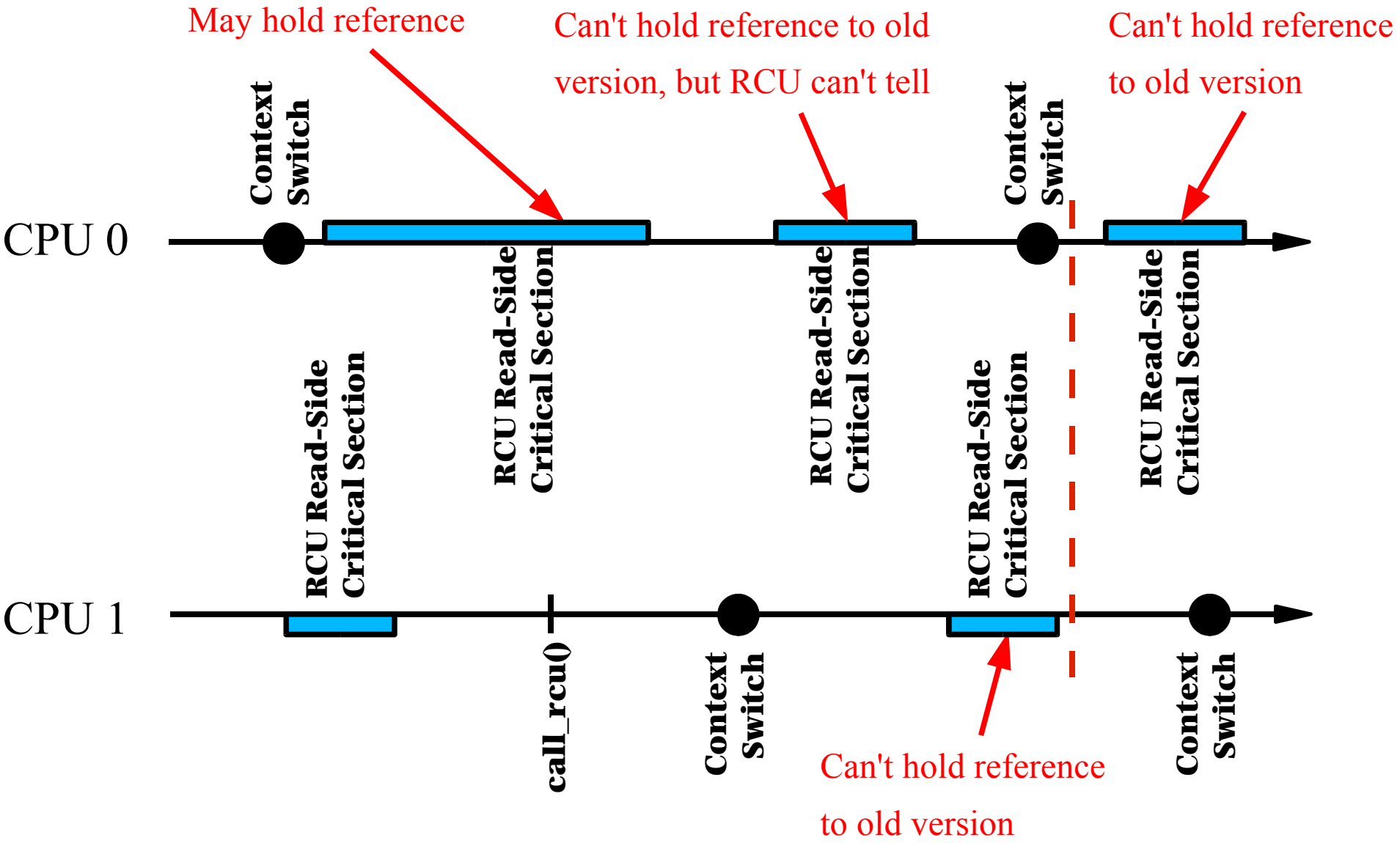
RCU Operation



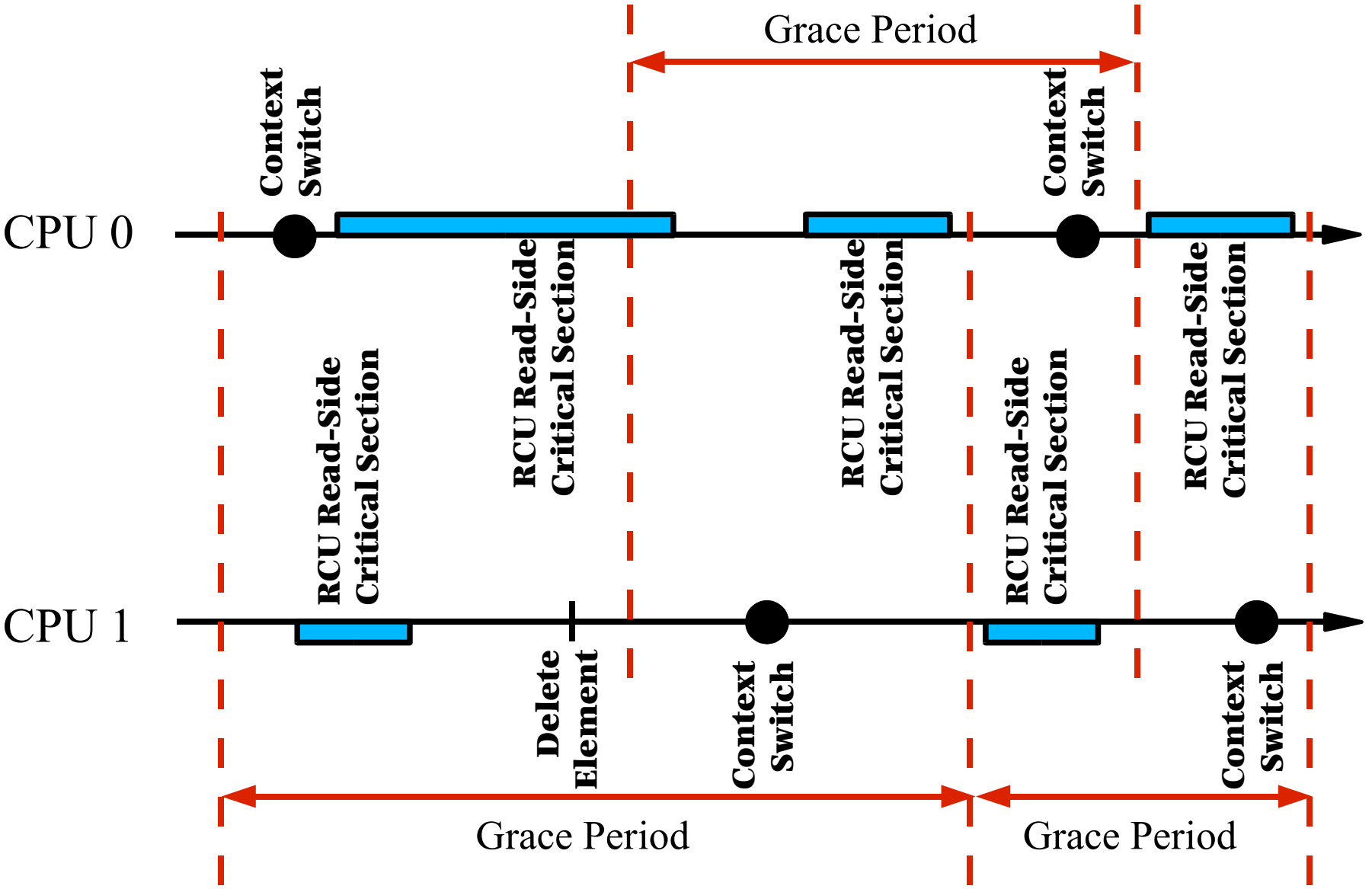
How Can RCU Updates Be Fast?

- Piggyback notification of reader completion on context switch (and similar events)
- Kernels are usually constructed as event-driven systems, with short-duration run-to-completion event handlers
  - Greatly simplifies deferring destruction because readers are short-lived
  - Permits tight bound on memory overhead
    - Limited number of versions waiting to be collected

RCU's Deferred Destruction



Grace Periods



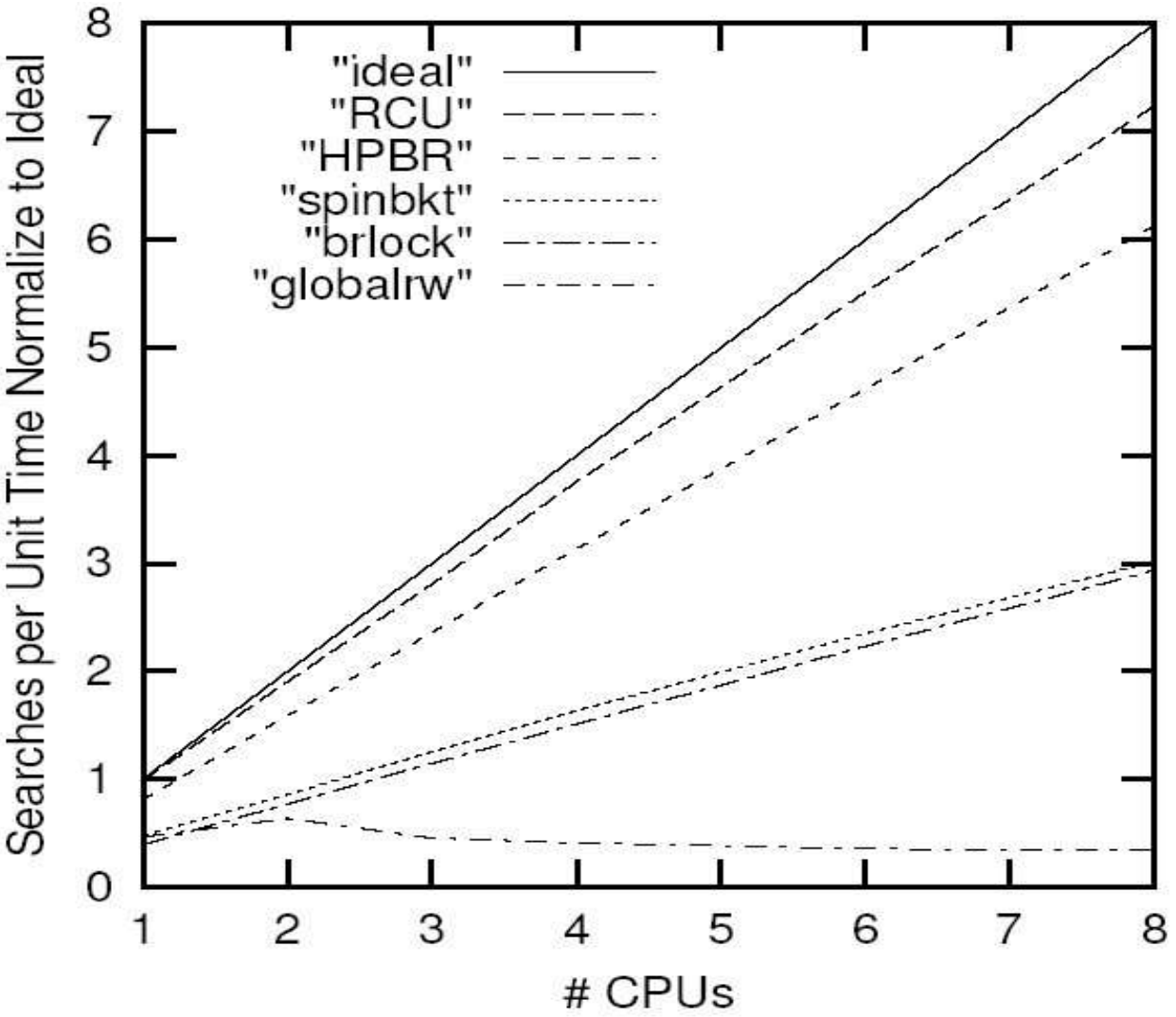
What is RCU? (1)

- Reader-writer synchronization mechanism
  - Best for read-mostly data structures
- Writers create new versions atomically
  - Normally create new and delete old elements
- Readers can access old versions independently of subsequent writers
  - Old versions garbage-collected by “poor man's” GC, deferring destruction
  - Readers must signal “GC” when done

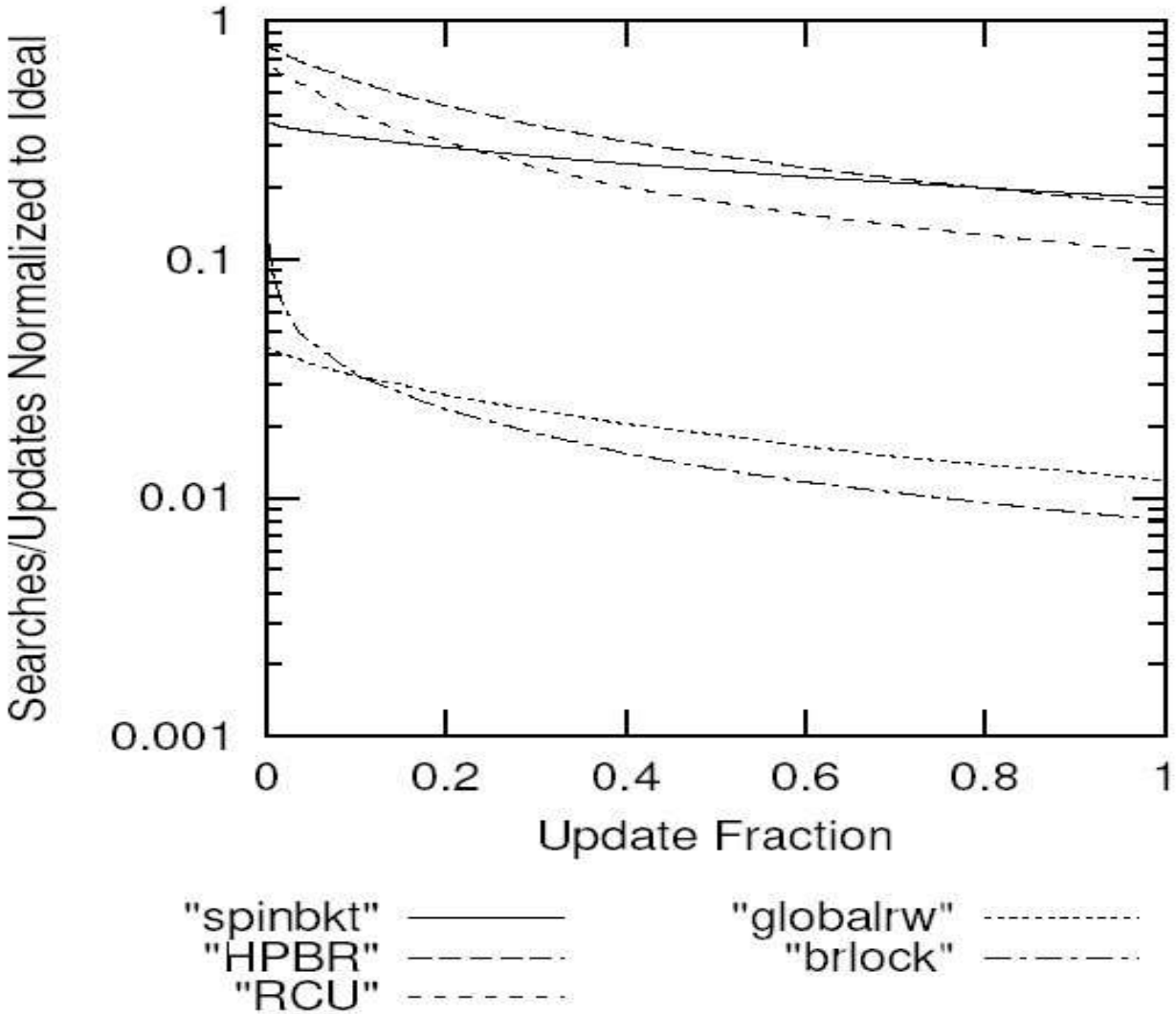
What is RCU? (2)

- Readers incur little or no overhead
- Writers incur substantial overhead
  - Writers must synchronize with each other
  - Writers must defer destructive actions until readers are done
  - The “poor man's” GC also incurs some overhead

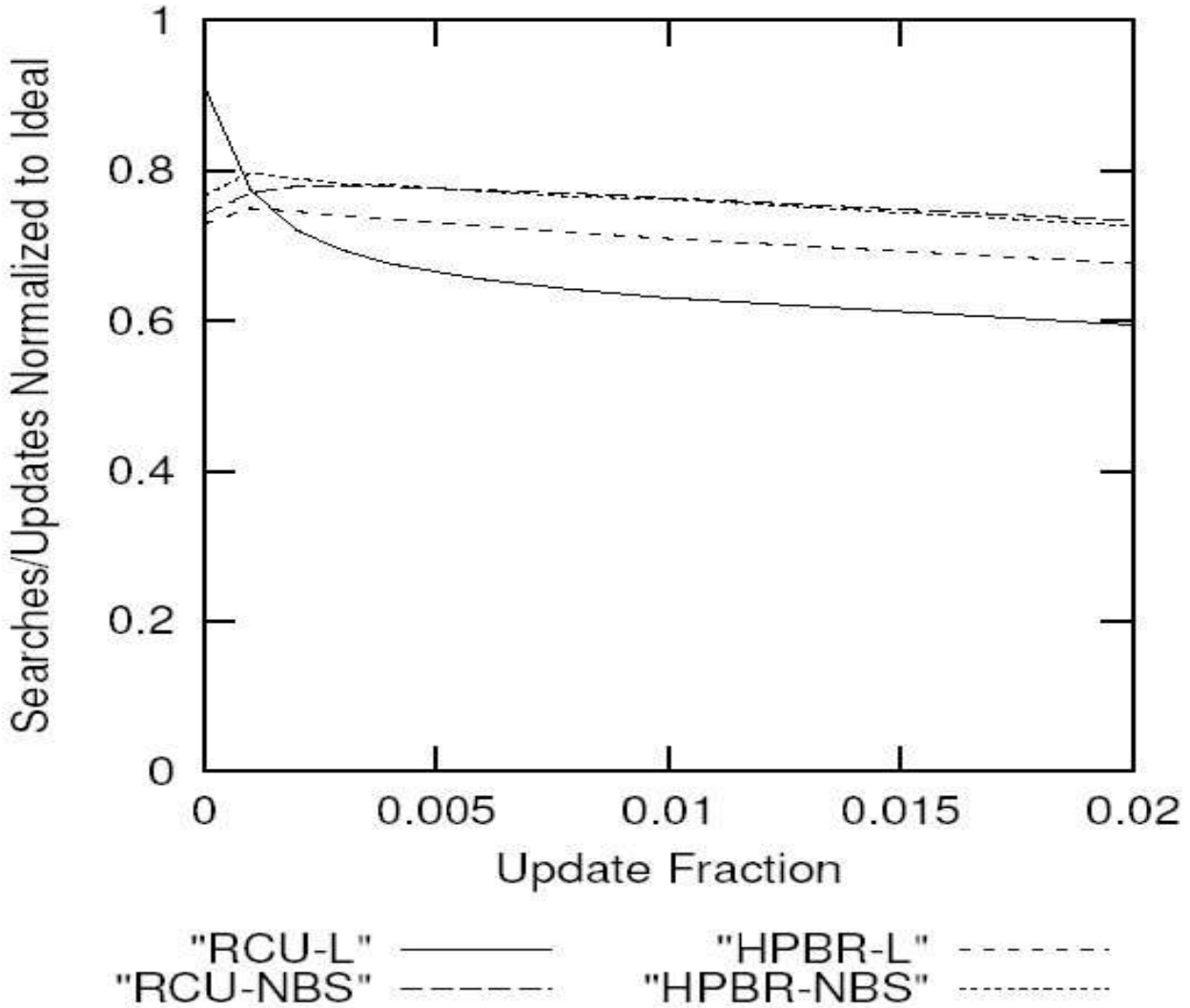
PPC Read-Only Results



PPC Mixed Workload



PPC Read-Mostly Mixed Workload



But We Cut HPBR a Break

- We assumed that the hazard pointers can be statically allocated
- Invalid assumption in production software, as many important data structures require unbounded numbers of hazard pointers:
  - tree traversal, graph traversal, nested data structures, recursive traversal of data structures
- Reality Check #6:
  - Hazard pointers must be dynamically allocated
  - Which will increase HPBR overhead

So Who Cares About 99.9% Reads???

- Networking routing table
  - 1,000 packets per second (moderate webserver)
  - Internet routing protocols limited to one update per few minutes (avoid route thrashing)
  - 99.999% reads!
- Hardware configuration tracking
  - Used on every I/O, almost ***never*** changes!
  - Essentially 100% reads
- Security policies, netfilter setup, dcache, ...
- Reality Check #7:
  - Read-mostly scenarios ***extremely*** important!!!

RCU Sem Micro-Benchmark

Kernel	Run 1	Run 2	Avg
2.5.42-mm2	515.1	515.4	515.3
2.5.42-mm2+ipc-rcu	46.7	46.7	46.7

Numbers are test duration, smaller is better.

8-CPU 700MHz Intel PIII System

RCU Sem DBT1 Performance

Kernel	Average	Standard Deviation
2.5.42-mm2	85.0	7.5
2.5.42-mm2+ipc-rcu	89.8	1.0

Numbers are transaction rate, larger is better.

2-CPU 900MHz PIII

When to Use RCU

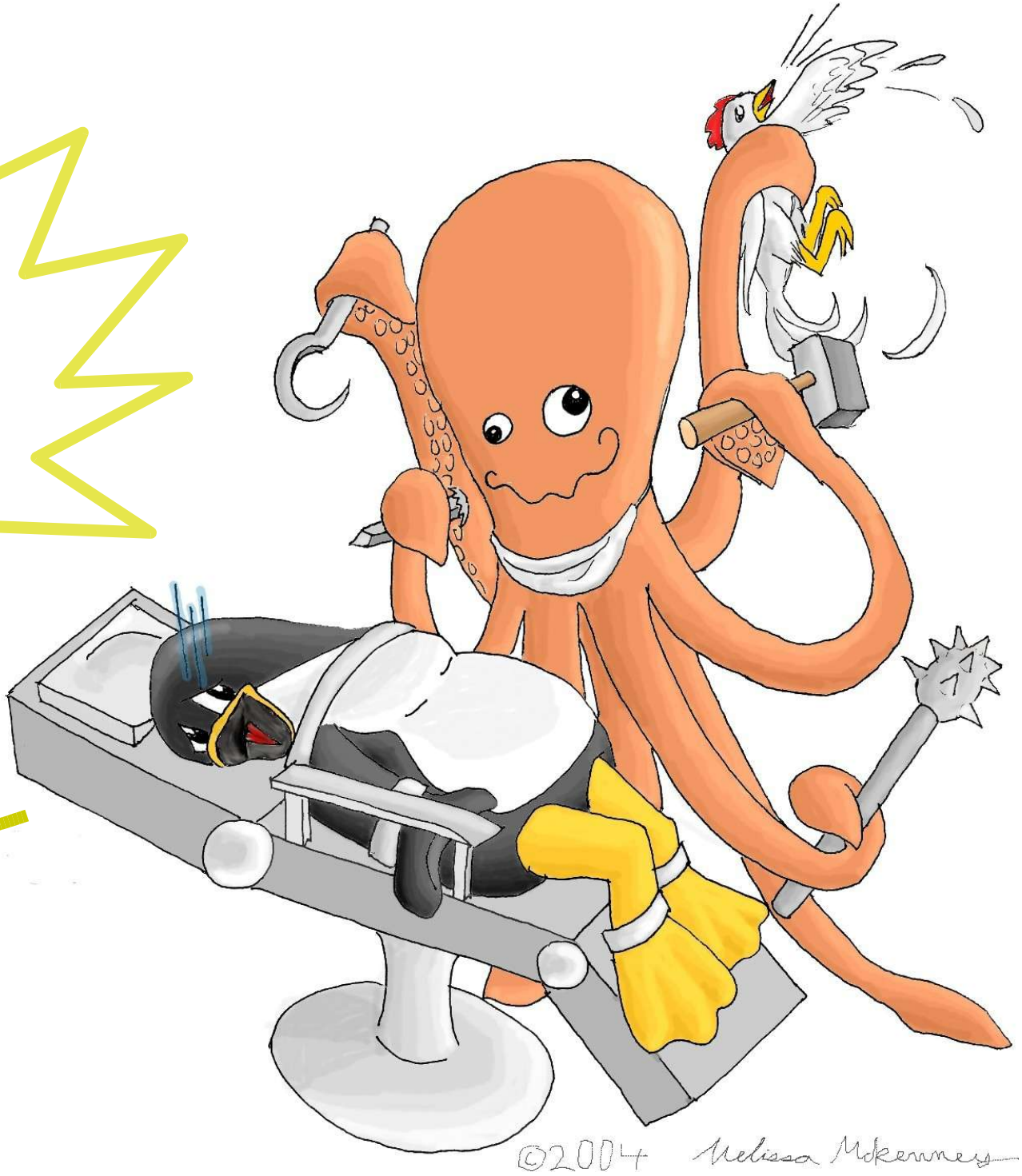
- Read-mostly data structures
- Algorithms that can tolerate concurrent accesses and updates
  - There are ways to transform algorithms into a form that can tolerate concurrent accesses and updates

Summary and Conclusions

What to Use Where (Short Form)

- Read-mostly situations: RCU
- Update-heavy situations:
  - Simple data structures and algorithms: NBS
    - Most likely in conjunction with hazard pointers
  - Complex data structures and algorithms: locking
    - Most likely in conjunction with some form of scheduler-conscious synchronization
- And for the final reality check...

Use the right tool for the job!!!



Legal Statement

- This work represents the view of the author, and does not necessarily represent the view of IBM.
- IBM, NUMA-Q, and Sequent are registered trademarks of International Business Machines in the United States, other countries, or both.
- Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
- Linux is a registered trademark of The Open Group in the United States and other countries.
- Other company, product, and service names may be trademarks or service marks of others.

BACKUP