Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels

Paul E. McKenney

OGI School of Science & Technology
Computer Science & Engineering
Paul.McKenney@us.ibm.com

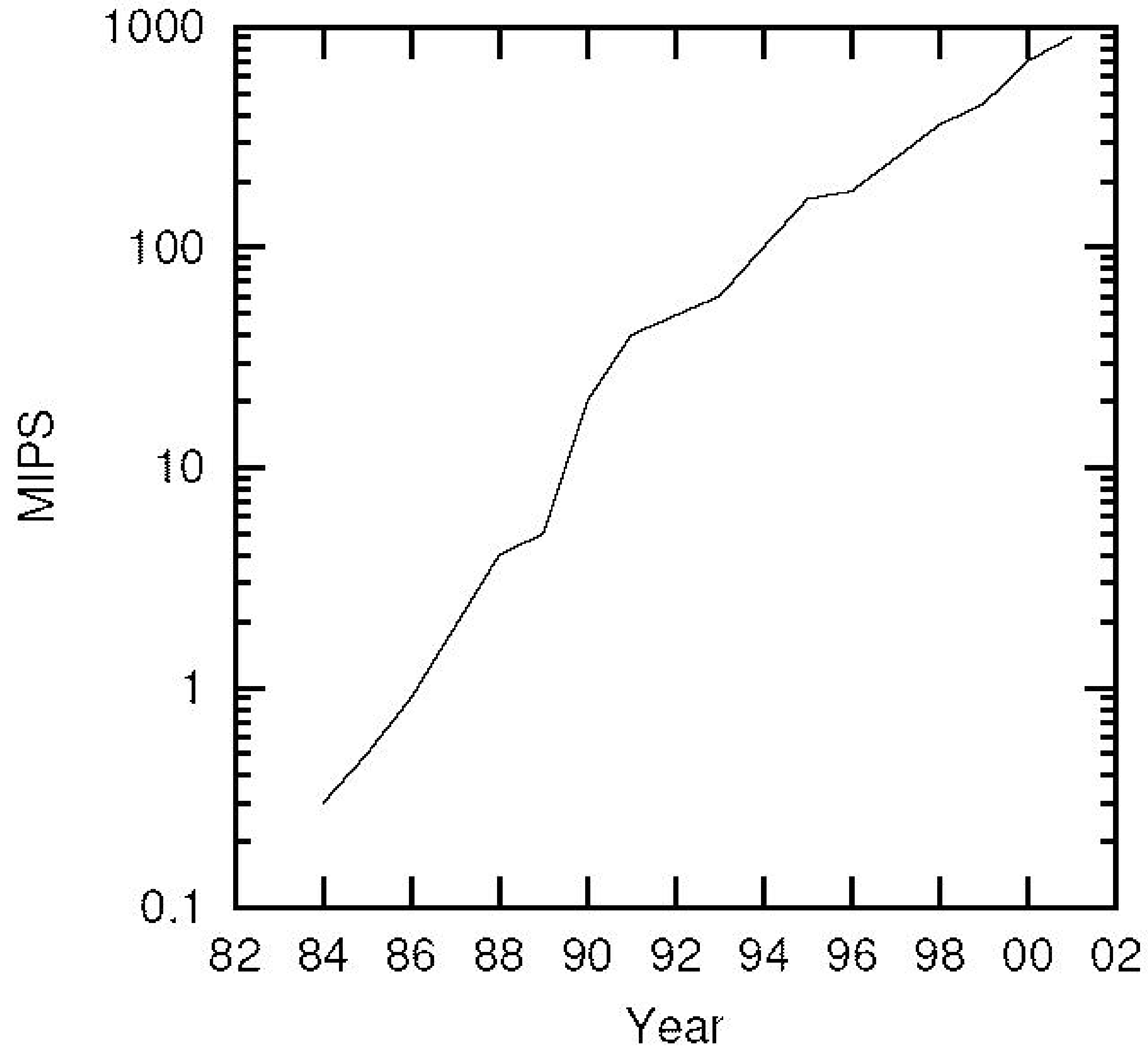
Advisor: Jon Walpole

Overview

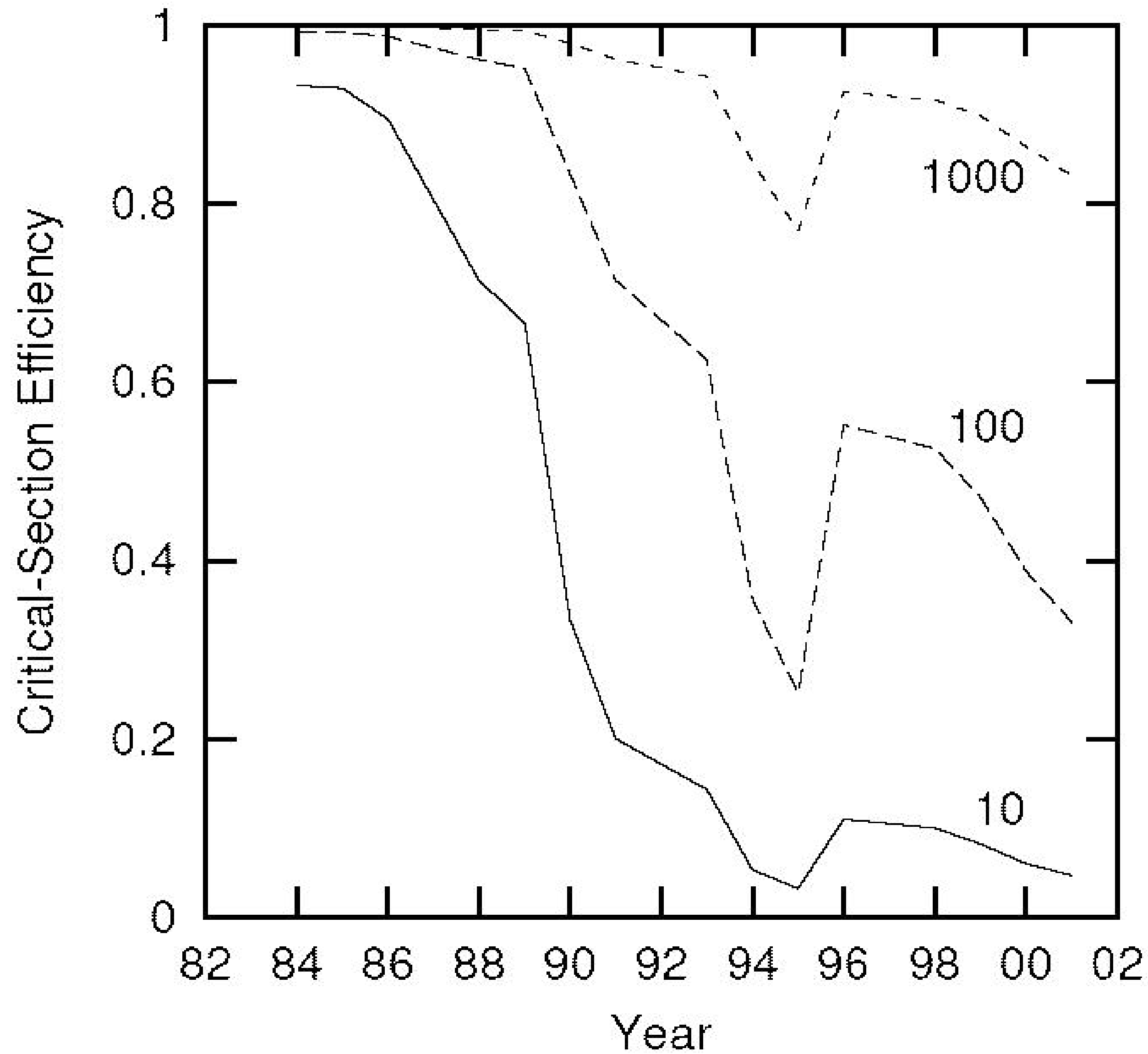
- Moore's Law and Parallel Software
- Analysis of Related Work
- My Thesis – RCU
- Efficient RCU Implementations
- RCU's Performance and Complexity
- RCU Design Patterns
- Summary and Conclusions

Moore's Law and Parallel Software

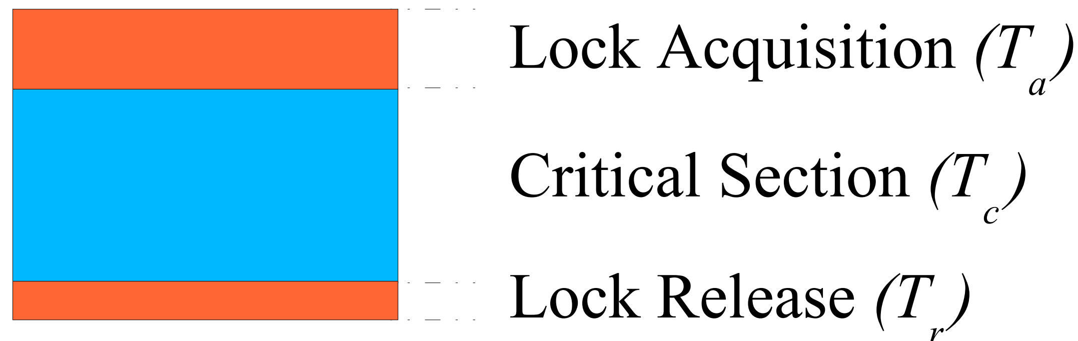
Instruction Speed Increased



Synchronization Speed Decreased



Critical-Section Efficiency



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

Assuming negligible contention and no caching effects in critical section

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
  - 2004: 3-4 level cache hierarchies, since instructions orders of magnitude faster than memory accesses
- Synchronization requires consistent view of data across CPUs, i.e., 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
  - 2004: Many instructions per clock – 20-stage pipelines
- Modern super-scalar CPUs execute instructions out of order in order to keep their pipelines full
  - Can't reorder the critical section before the lock!!!
- Therefore, synchronization operations must stall the pipeline, decreasing performance

What is Going On? (3/3)

- 1984: The main issue was lock contention
- 2004: 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

Analysis of Related Work

Related Work

- Global locking (“code locking”)
- Reader-writer locking
- Data locking/partitioning
- Per-CPU reader-writer lock (brlock)
- Wait-free synchronization

- Need to compare to each other and to ideal

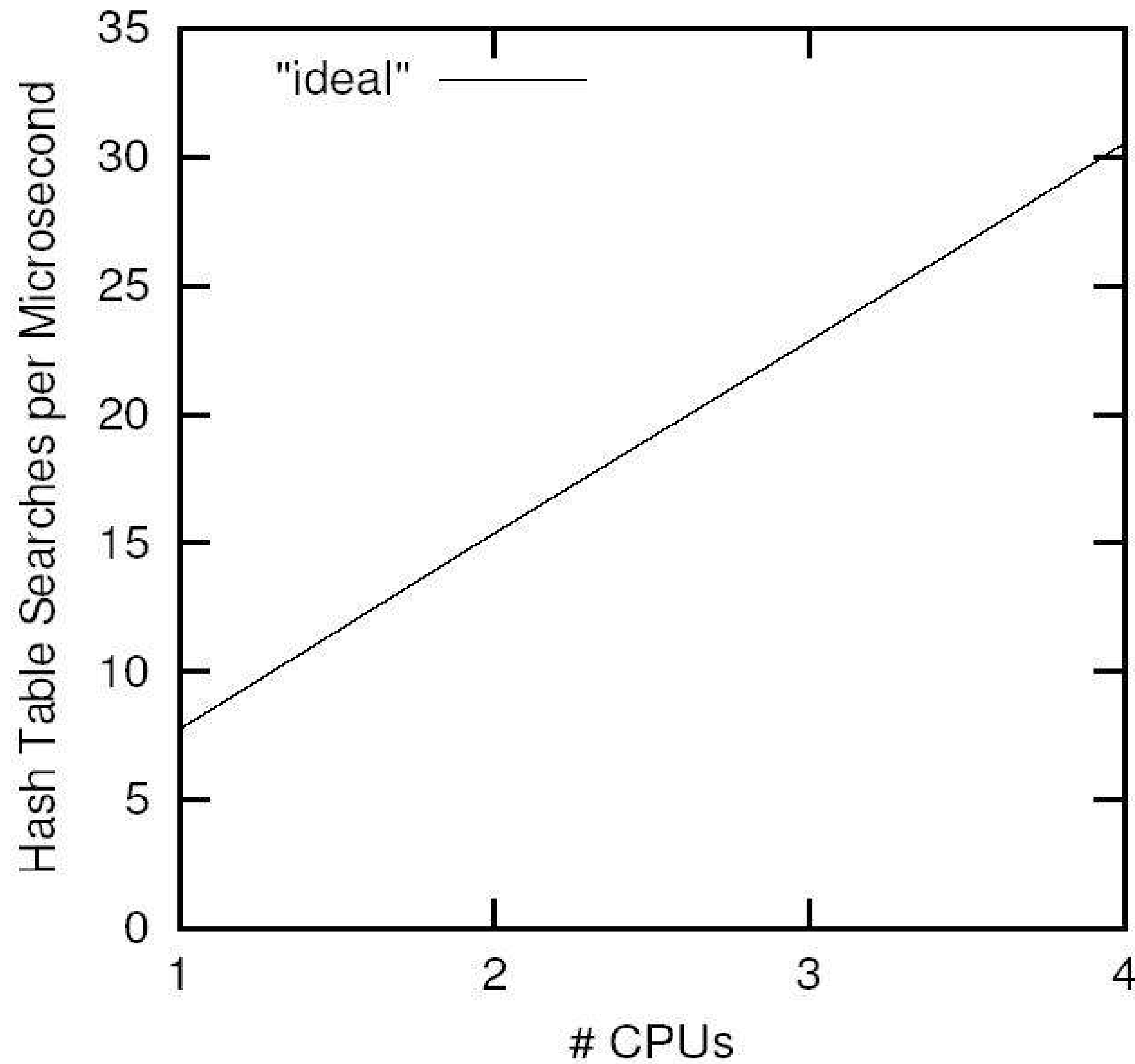
What Benchmark to Use?

- 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 selected for specified mix
    - (See thesis)
- Start with pure search workload (read only)
- Run on 4-CPU 700MHz P-III system
  - Single quad Sequent/IBM NUMA-Q system

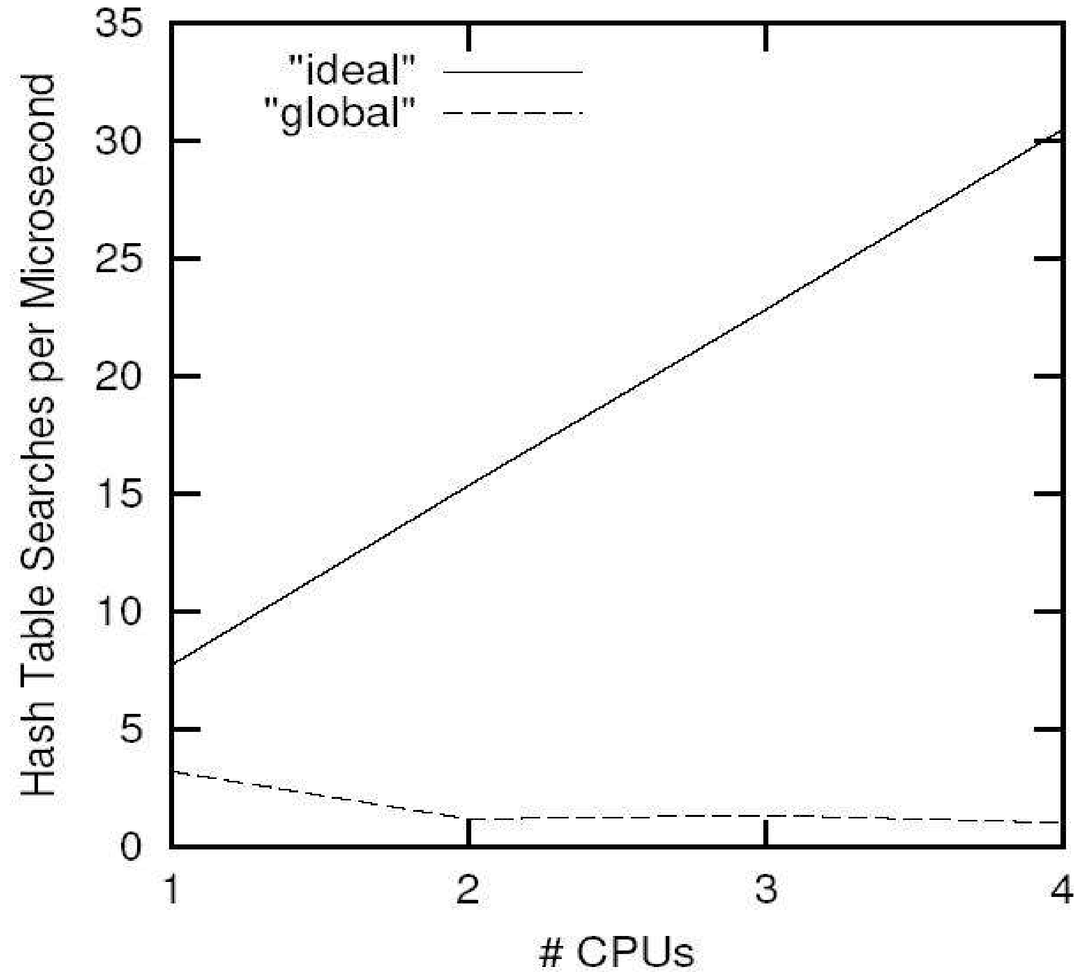
Ideal Performance



What is Global Locking?

- “Textbook” locking
  - Hoare monitor
  - “Code locking”
  - Java “synchronized” on global object
  - Linux spin_lock() on global spinlock_t
- Only one CPU at a time allowed to perform the activity guarded by the global lock

Performance of Global Locking



Global locking performance sucks!!!

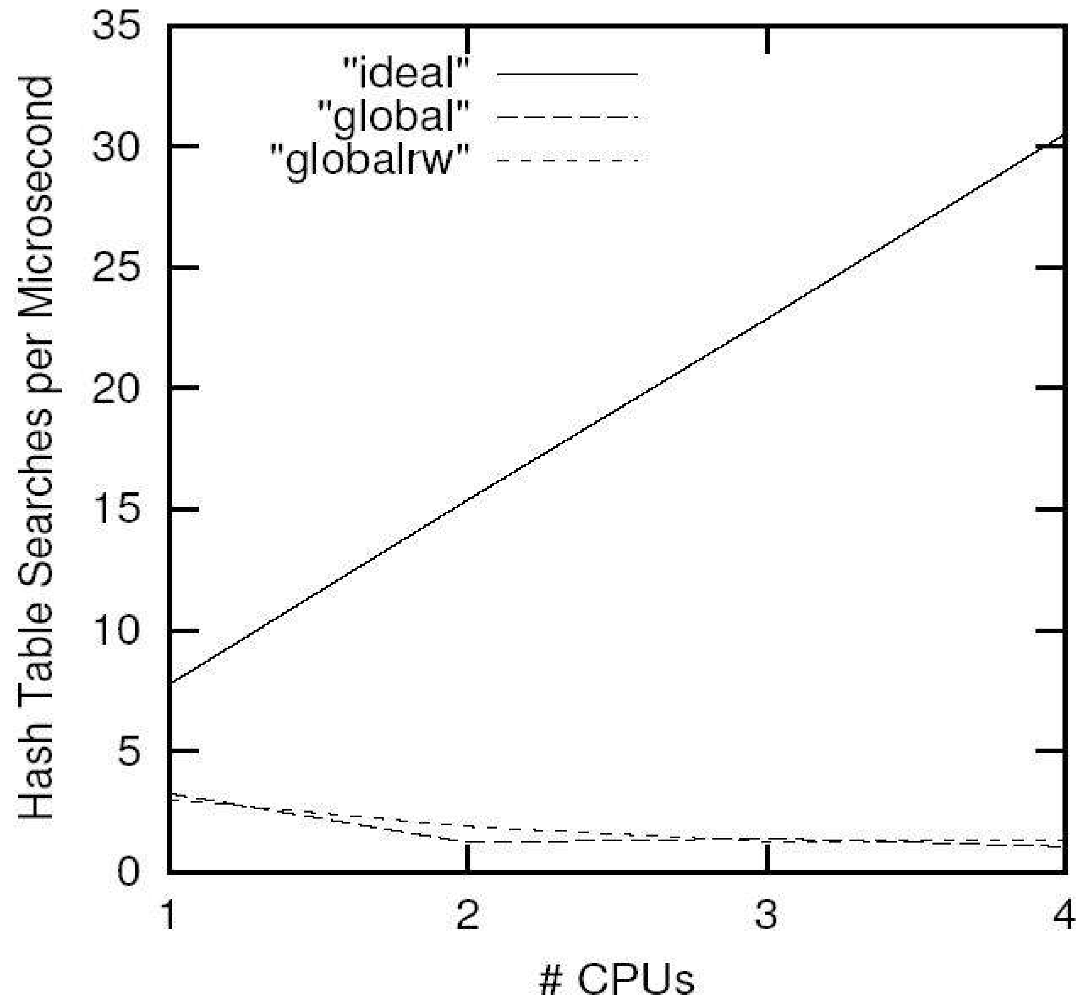
Why So Slow?

- Could it be lock contention?
  - Only one CPU at a time may access the hash table
  - The CPUs are doing nothing but accessing the hash table
- Recipe for severe lock contention

Reader-Writer Locking

- Many readers can concurrently hold the lock
- Writers exclude readers and other writers
- This benchmark does only searches
  - So there are readers but no writers
  - Therefore reader-writer lock eliminates lock contention for this benchmark!!!

Reader-Writer Performance



Lock contention is not the issue!!!

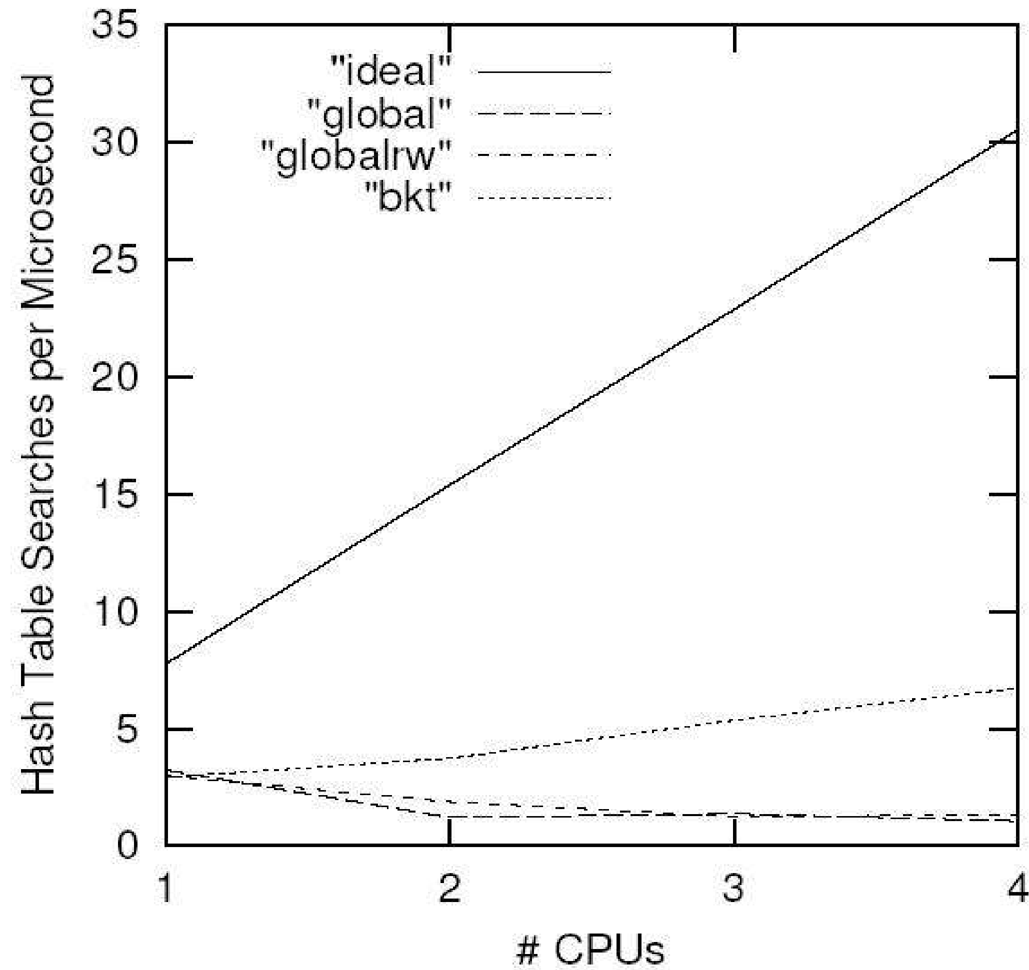
What is Limiting Performance?

- Overheads:
  - ~~Lock contention~~
  - Memory latency?
  - Pipeline stalls?
  - Instruction overhead?
- Reader-writer lock serializes memory latency
  - Can we incur memory latency in parallel?

Data Locking

- Per-hash-bucket locking is a form of data locking for hash tables
- CPUs still acquire locks, but can acquire locks for different hash chains in parallel
  - CPUs incur memory-latency and pipeline-flush overheads in parallel

Per-Bucket Locking Performance



Memory latency parallelized, but performance still sucks!!!

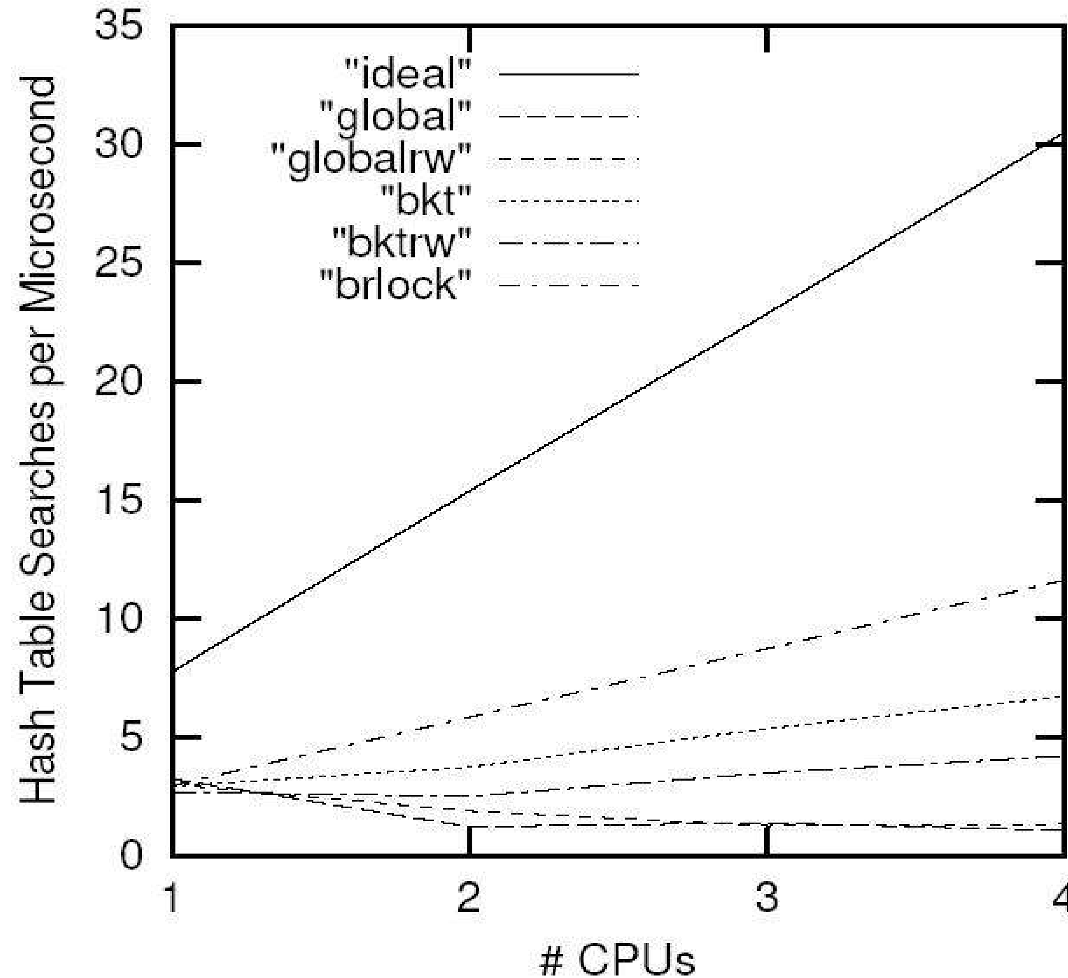
What Else Can Be Done???

- Overheads:
  - ~~Lock contention~~
  - Memory latency
    - Parallelized, but still present
  - Pipeline flushing
    - Parallelized, but still present
  - Instruction overhead
- Can we eliminate memory latency?

Per-CPU Reader-Writer Lock

- One lock per CPU
- Reader acquires own lock, writer all locks
  - In read-only workload, CPUs need not access or modify other CPUs' locks
  - Therefore no memory latency is incurred

Per-CPU Reader-Writer-Lock (brlock) Performance

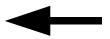


Memory latency eliminated, but performance still poor!!!

Why So Slow???

- Overheads:
  - ~~Lock Contention~~
  - ~~Memory Latency~~
  - Pipeline-Stall Overhead
  - Instruction Overhead

Instruction/Pipeline Costs on a 4-CPU 700MHz i386 P-III

Operation	Nanoseconds	
Instruction	0.7	
Clock Cycle	1.4	
L2 Cache Hit	12.9	
Atomic Increment	58.2	
Cmpxchg Atomic Increment	107.3	
Atomic Incr. Cache Transfer	113.2	
Main Memory	162.4	
CPU-Local Lock	163.7	
Cmpxchg Blind Cache Transfer	170.4	
Cmpxchg Cache Transfer and Invalidate	360.9	

What About Wait-Free Synchronization?

- What is wait-free synchronization?
  - Roll back to resolve conflicting changes instead of spinning or blocking
  - Uses 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

Why Not Wait-Free Synchronization?

Operation	Nanoseconds
Instruction	0.7
Clock Cycle	1.4
L2 Cache Hit	12.9
Atomic Increment	58.2
Cmpxchg Atomic Increment	107.3
Atomic Incr. Cache Transfer	113.2
Main Memory	162.4
CPU-Local Lock	163.7
Cmpxchg Blind Cache Transfer	170.4
Cmpxchg Cache Transfer and Invalidate	360.9

What Have We Learned?

- Contention is bad
- Memory latency is bad
- Pipeline stalls are bad
- Atomic instructions are bad

- But there is hope...

Promising Ideas (1/2)

- Reducing contention
  - Readers vs. writers
  - Partitioning
  - Per-CPU locking
- Hiding complex updates behind atomic commit points [Herlihy]
  - Reduces number of atomic instructions

Promising Ideas (2/2)

- Maintaining multiple versions [Kung, Herlihy]
  - Changes problem from inconsistency to staleness
- Deferring destruction [Kung,Hennessy]
  - Garbage collection for multiple versions reduces complexity
  - Batched destruction amortizes overhead

My Thesis – RCU

What is RCU? (1)

- Reader-writer synchronization mechanism
  - Leverages read-mostly nature of many operating-system data structures
- Writers use atomic commit points create new versions
- Readers can access old versions independently of subsequent writers
  - Old versions garbage-collected by deferring destruction
  - Readers must signal writers when done

What is RCU? (2)

- Writers incur substantial overhead
  - Writers must synchronize with each other
  - Writers must defer destructive actions until readers are done

RCU Performance Challenges

- How can readers signal writers?
  - Atomic instruction?
  - Can reader completion be inferred from existing system state?
- How can destruction be deferred without burdening readers?
  - Can costs of destruction be associated with writers and batched to amortize overhead?
- How can memory consumption be limited?

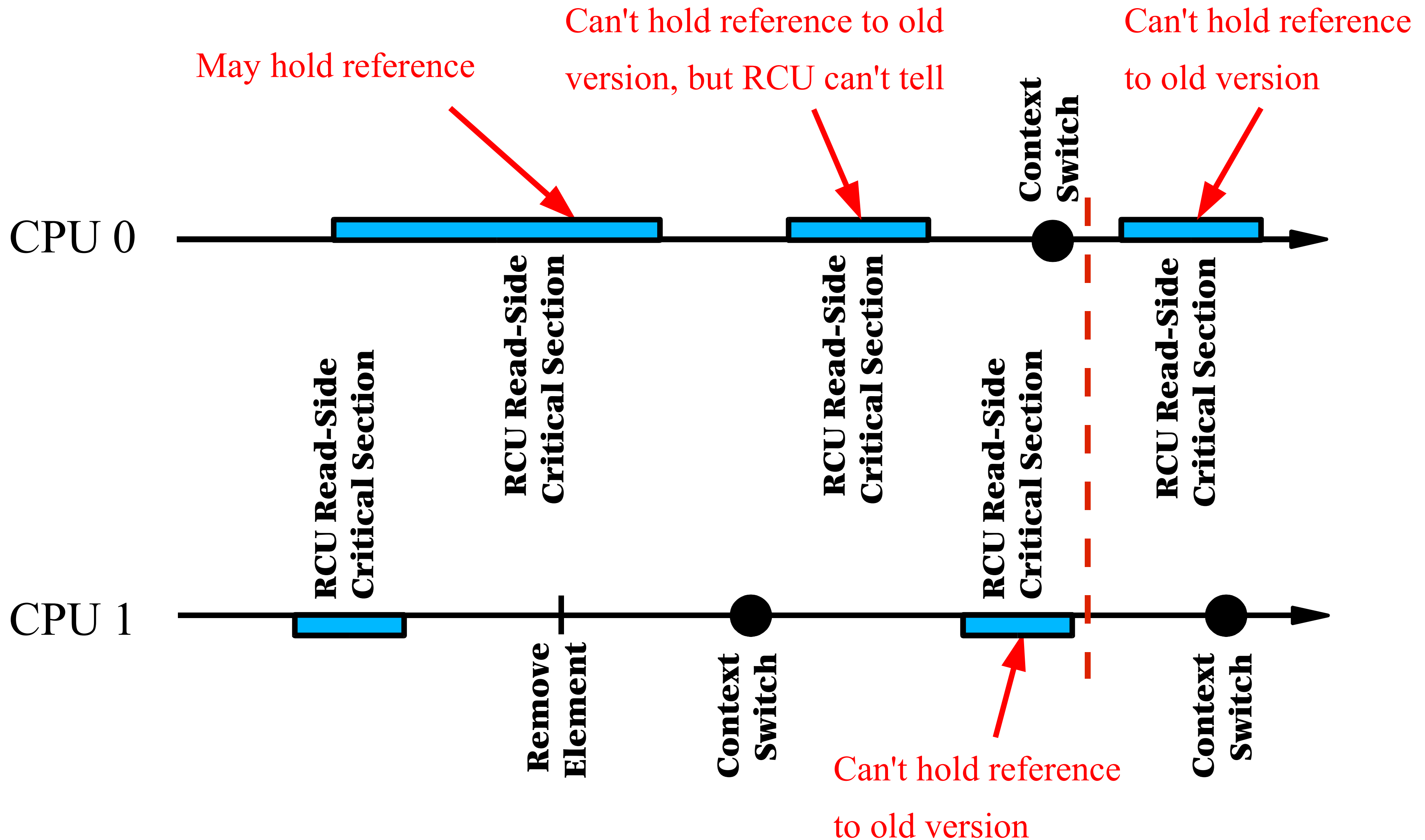
What is RCU's Environment? (1/2)

- Operating system kernels
- Many read-mostly data structures
  - Routing tables, configuration data structures
  - 10% or fewer of accesses are updates
    - For some data structures, only one update per million accesses
- Motivates asymmetric approaches greatly favoring readers

What is RCU's Environment? (2/2)

- Context switch is an example pre-existing event that signifies reader completion
    - Piggyback notification of reader completion on context-switch 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



How Does RCU Address Overheads? (1/2)

- Lock Contention
  - Readers need not acquire locks: no contention!!!
  - Writers can still suffer lock contention
    - But only with each other, and writers are infrequent
    - Very little contention!!!
- Memory Latency
  - Readers do not perform memory writes
  - No need to communicate data among CPUs for cache consistency
    - Memory latency greatly reduced

How Does RCU Address Overheads? (2/2)

- Pipeline-Stall Overhead
  - On most CPUs, readers do not stall pipeline due to update ordering or atomic operations
- Instruction Overhead
  - No atomic instructions required for readers
  - Readers only need to execute fast instructions

RCU Research Challenges (1/2)

- Provide efficient RCU implementations
  - Inferring reader signals from existing operations such as context switch
  - Generalize this across multiple OS kernels
- Evaluate RCU's performance
  - CPU overhead
  - Scalability
  - Latency

RCU Research Challenge (2/2)

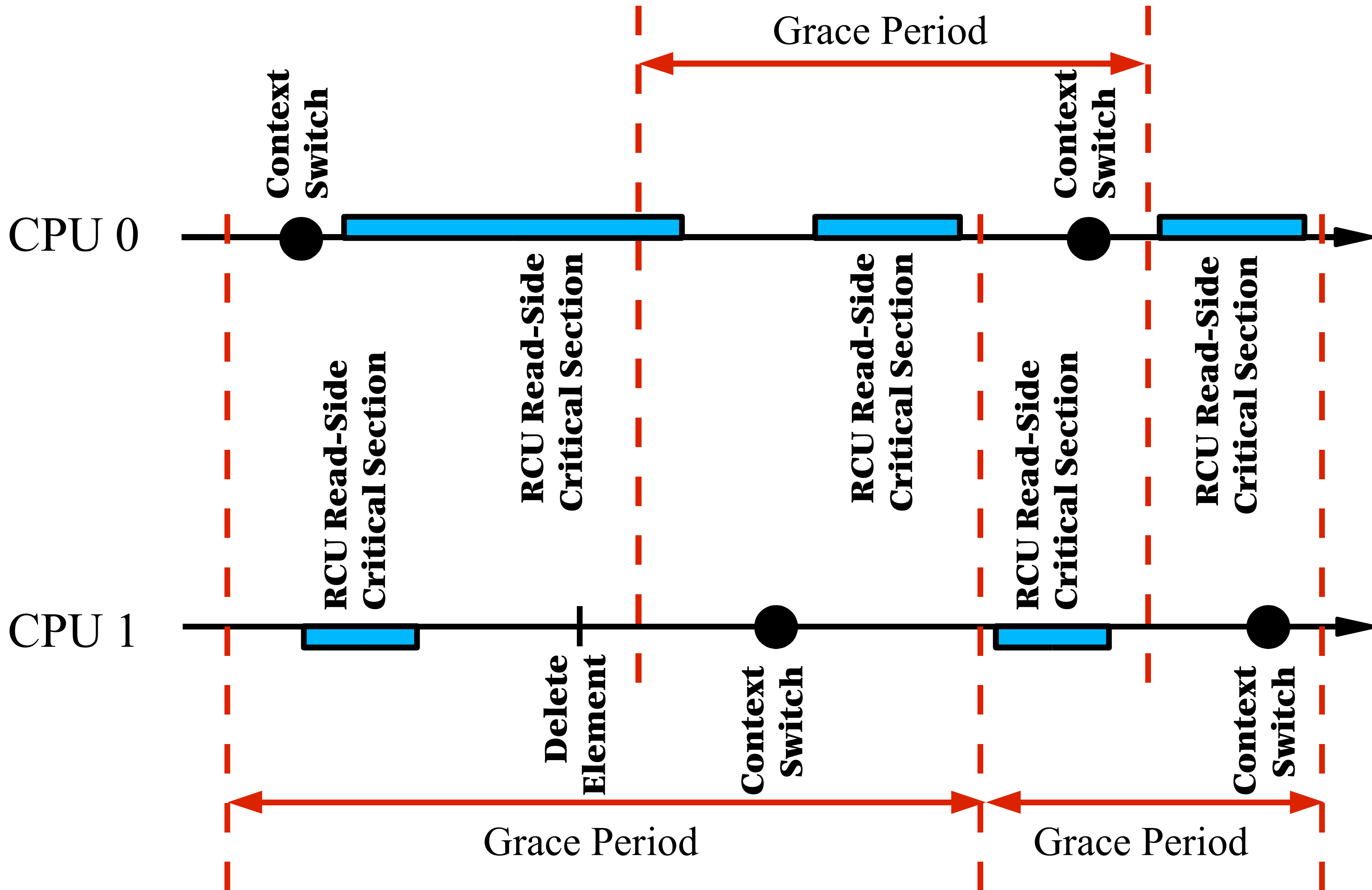
- Evaluate RCU's code complexity
  - Size of code
  - Impact on structure of kernel
- Make RCU generally applicable and easy to use
  - Define RCU design patterns

Efficient RCU Implementation

RCU Implementation Overview

- An RCU implementation must contain mechanisms for:
  - Deferring destructive actions.
  - Detecting when all readers have completed (grace period detection).
  - Carrying out destruction at end of grace period.

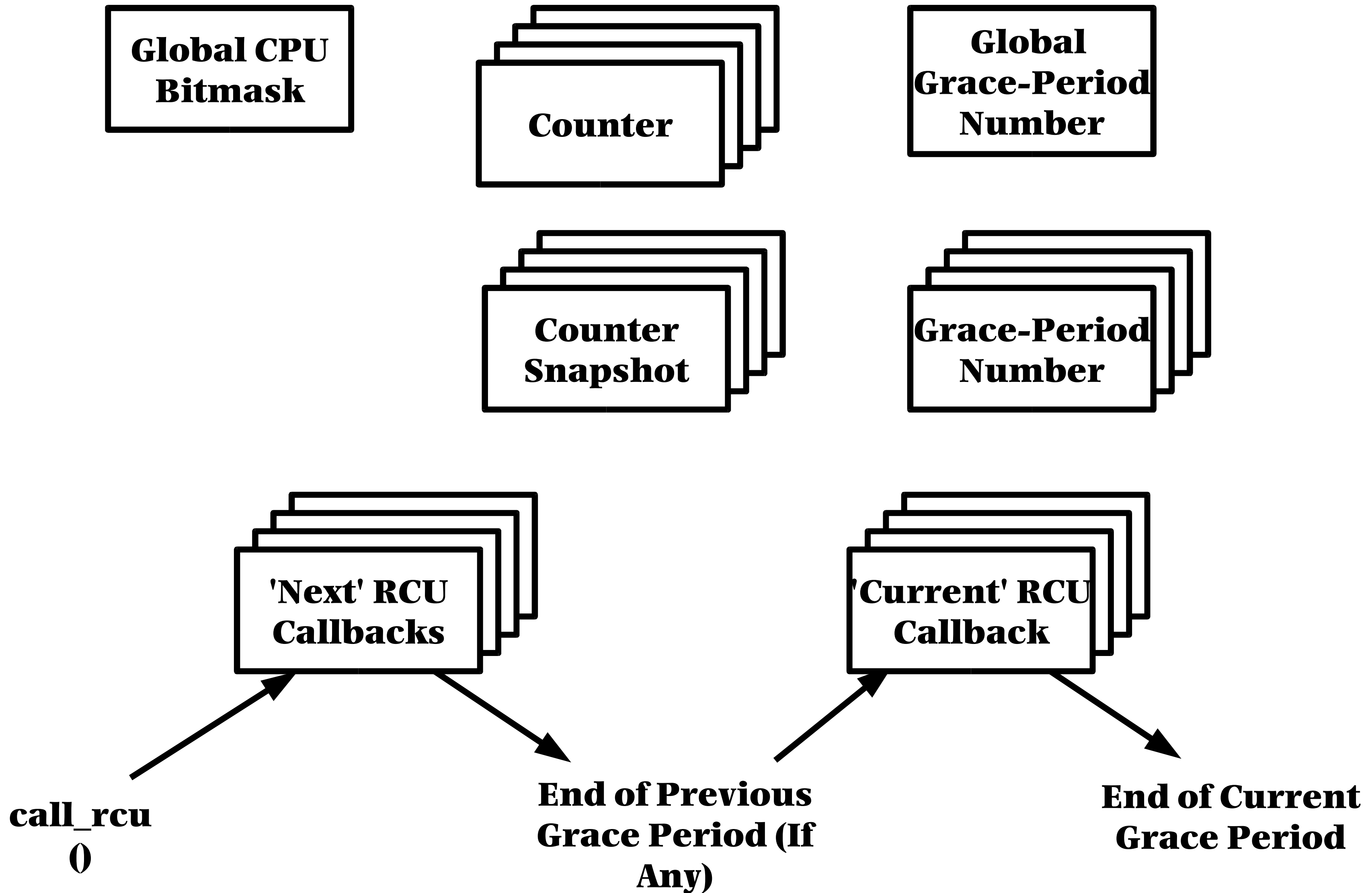
Grace Periods



Deferring Destructive Actions

- Writers enqueue destructive actions
  - Represent action as a data structure containing:
    - Function (destructive operation)
    - Argument (references data to be destroyed)
  - Data structure is called an “RCU callback”
- Maintain per-CPU queues of RCU callbacks
  - Avoids CPU-to-CPU communication
  - Requires two RCU-callback queues per CPU:
    - callbacks waiting for grace-period start (“next”)
    - callbacks waiting for grace period end (“current”)

RCU's Data Structures



List of RCU Implementations

- DYNIX/ptx RCU (data center)
- Linux
  - Multiple implementations (in 2.5 kernel)
  - Preemptible and nonpreemptible
- Tornado/K42 “generations”
  - Preemptive kernel
  - Helped generalize usage

RCU Performance and Complexity

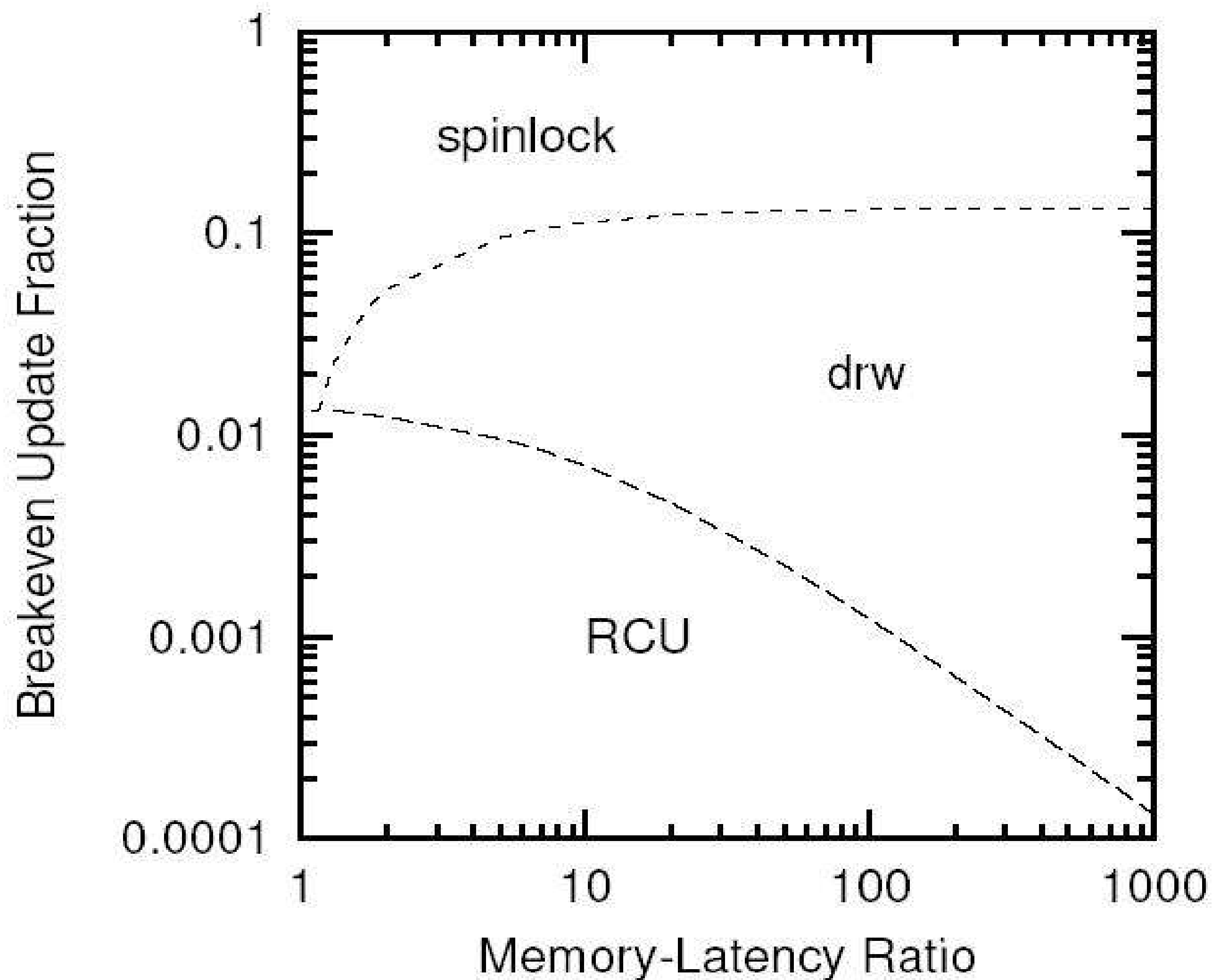
Analytical Evaluation

- When should RCU be used?
  - Instead of simple spinlock?
  - Instead of per-CPU reader-writer lock?
- Under what environmental conditions?
  - Memory-latency ratio
  - Number of CPUs
- Under what workloads?
  - Fraction of accesses that are updates
  - Number of updates per grace period

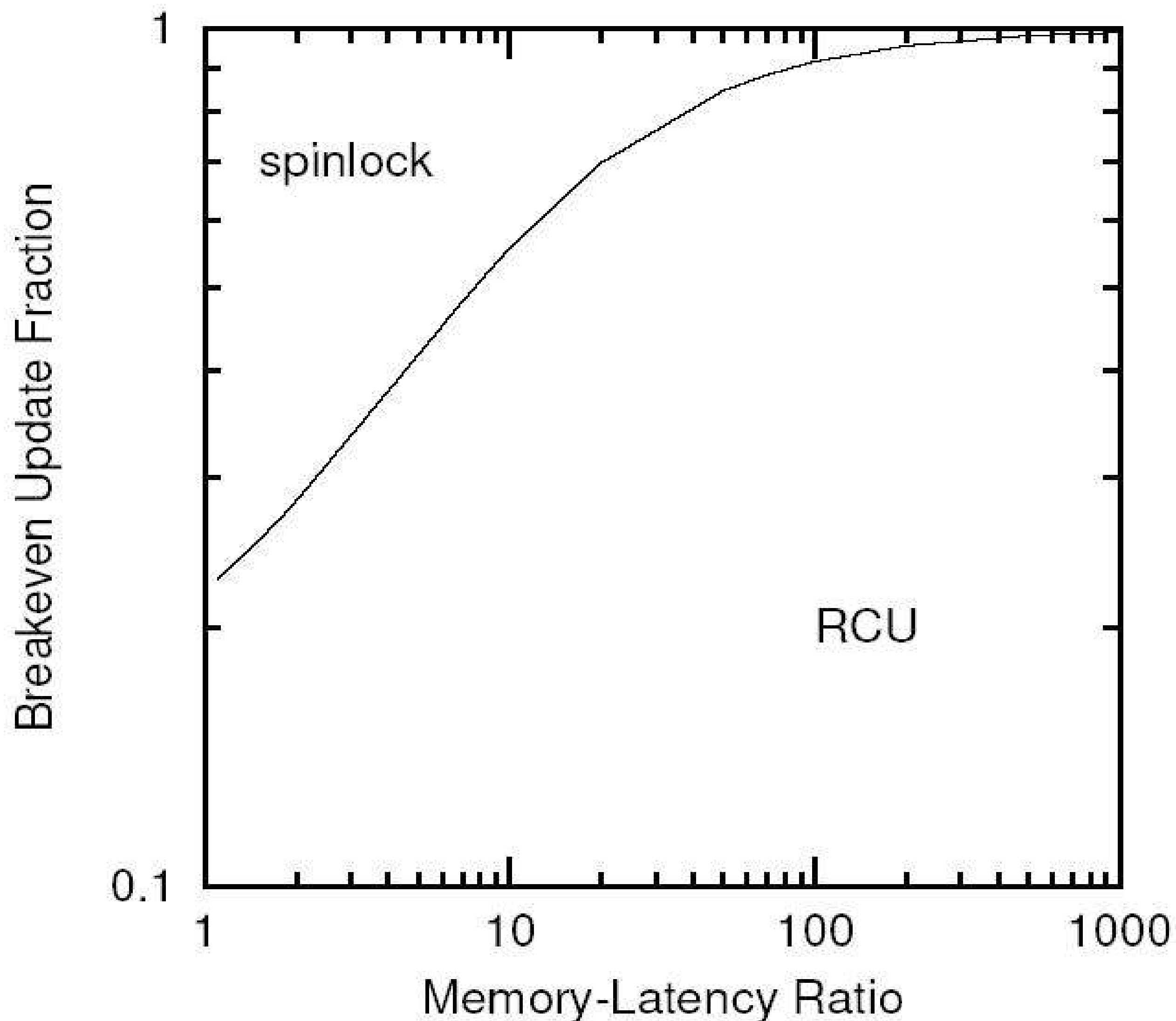
Representative Analytic Results

- Compute breakeven update-fraction contours for RCU vs. locking performance, against:
  - Number of CPUs (n)
  - Updates per grace period (λ)
  - Memory-latency ratio (r)
- Look at computed memory-latency ratio at extreme values of λ for n=4 CPUs
  - Other scenarios covered in thesis chapter 7

Breakevens for RCU Worst Case (f vs. r for Small λ)



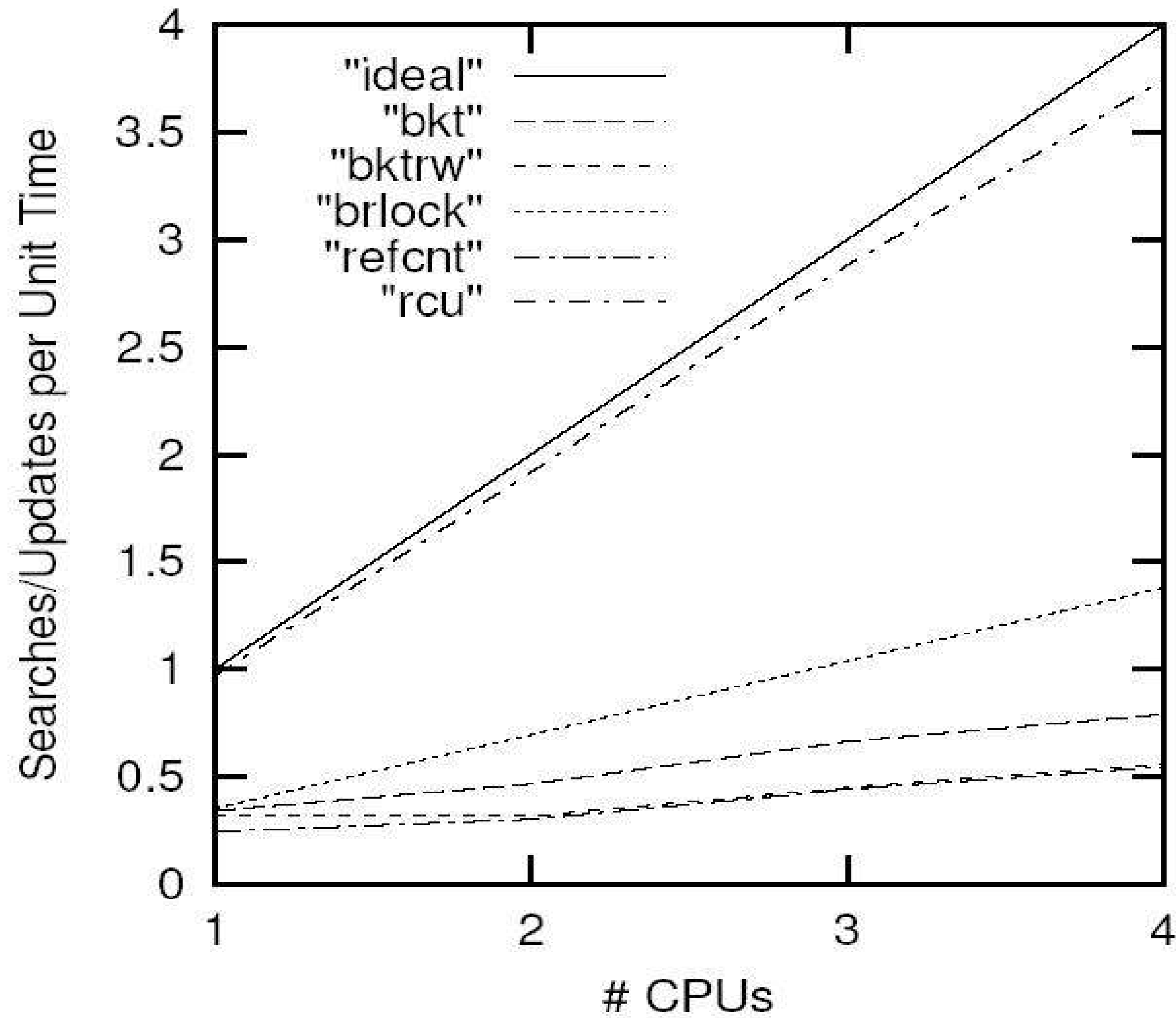
Breakeven for RCU Best Case (f vs. r, Large λ)



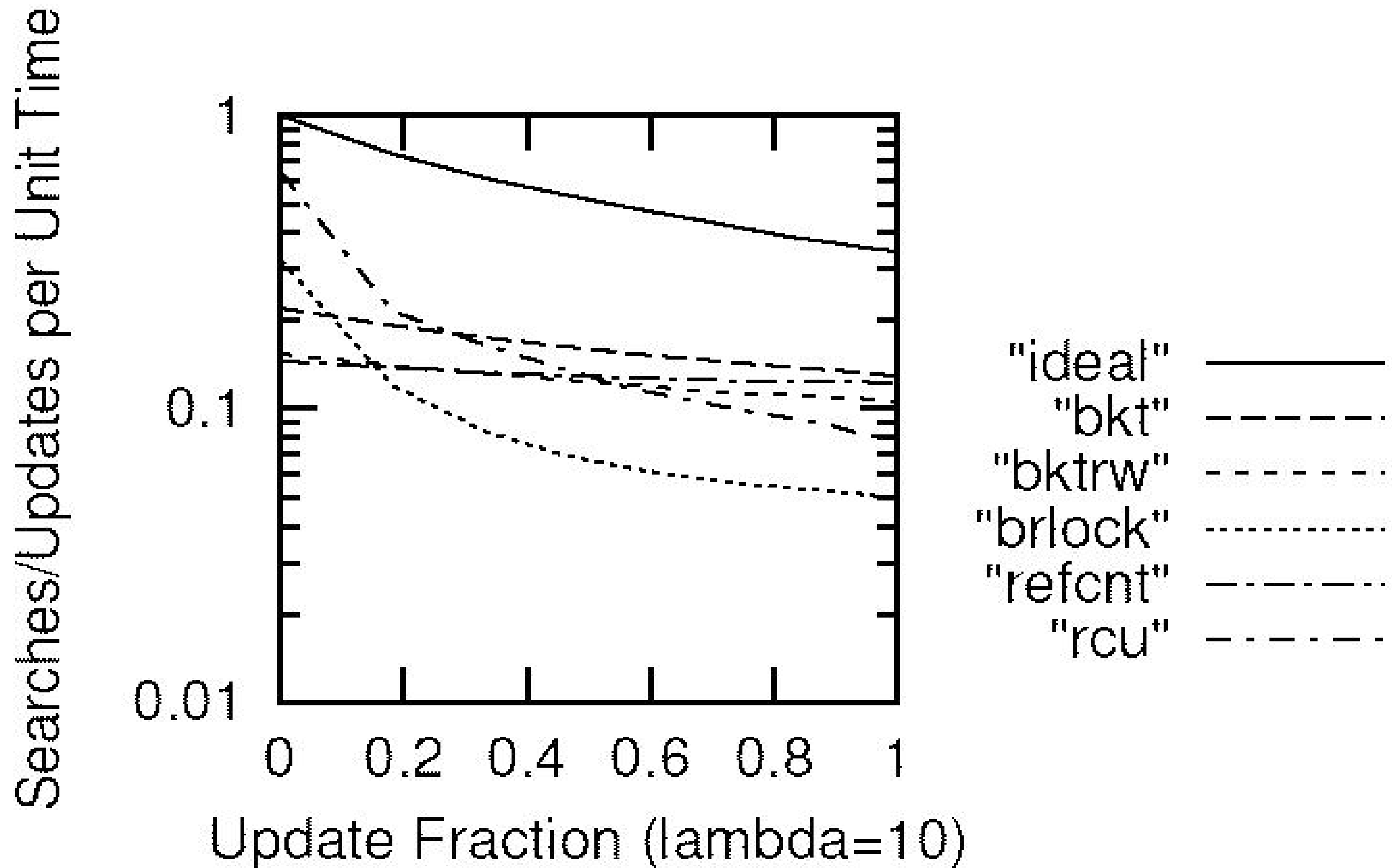
Validate Analytic Results

- 4-CPU 700MHz P-III system (NUMA-Q quad)
- Read-only mini-benchmark
  - For data structures that are almost never modified
    - Routing tables, HW/SW configuration, policies
- Mixed workload mini-benchmark
  - Vary fraction of accesses that are updates
  - See how things change as read-intensity varies
  - Expect breakeven point for RCU and locking

x86 Read-Only Results



x86 Results for Mixed Workload



Real-World RCU Performance and Complexity

- SysV IPC
  - >10x on microbenchmark (8 CPUs)
  - 5% for database benchmark (2 CPUs)
  - 151 net lines added to the kernel
- Directory-Entry Cache
  - +20% in multiuser benchmark (16 CPUs)
  - +12% on SPECweb99 (8 CPUs)
  - -10% time required to build kernel (16 CPUs)
  - 126 net lines added to the kernel

Real-World RCU Performance and Complexity

- Task List
  - +10% in multiuser benchmark (16 CPUs)
  - 6 net lines added to the kernel
    - 13 added
    - 7 deleted

RCU Design Patterns

What is a Design Pattern?

Design patterns capture the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context.

Because they address fundamental challenges in software system development, design patterns are an important technique for improving the quality of software.

Key challenges addressed by design patterns include communication of architectural knowledge among developers, accommodating a new design paradigm or architectural style, and avoiding development traps and pitfalls that are usually learned only by (painful) experience.

Coplien and Schmidt, 1995

Two Types of Design Patterns For RCU

- For situations well-suited to RCU:
  - Patterns that describe direct use of RCU
- For algorithms that do not tolerate RCU's stale- and inconsistent-data properties:
  - Patterns that describe transformations of algorithms into forms that can tolerate stale and/or inconsistent data

Patterns for Direct RCU Use

- Reader/Writer-Lock/RCU Analogy
  - Routing tables, Linux tasklist lock patch, ...
- RCU Readers With WFS Writers
  - K42 hash tables
- RCU Existence Locks
  - Ensure data structure persists as needed
  - Linux SysV IPC, dcache, IP route cache, ...
- Pure RCU
  - Dynamic interrupt handlers...
  - Linux NMI handlers...

Reader/Writer-Lock/RCU Analogy

- read_lock()
- read_unlock()
- write_lock()
- write_unlock()
- list_add()
- list_del()
- free(p)

- rcu_read_lock()
- rcu_read_unlock()
- spin_lock()
- spin_unlock()
- list_add_rcu()
- list_del_rcu()
- call_rcu(free, p)

Patterns for Direct RCU Use

- Reader/Writer-Lock/RCU Analogy (5)
- RCU Readers With WFS Writers (1)
- RCU Existence Locks (7)
- Pure RCU (4)

Stale and Inconsistent Data

- RCU allows concurrent readers and writers
  - RCU allows readers to access old versions
    - Newly arriving readers will get most recent version
    - Existing readers will get old version
  - RCU allows multiple simultaneous versions
    - A given reader can access different versions while traversing an RCU-protected data structure
    - Concurrent readers can be accessing different versions
- Some algorithms tolerate this consistency model, but many do not

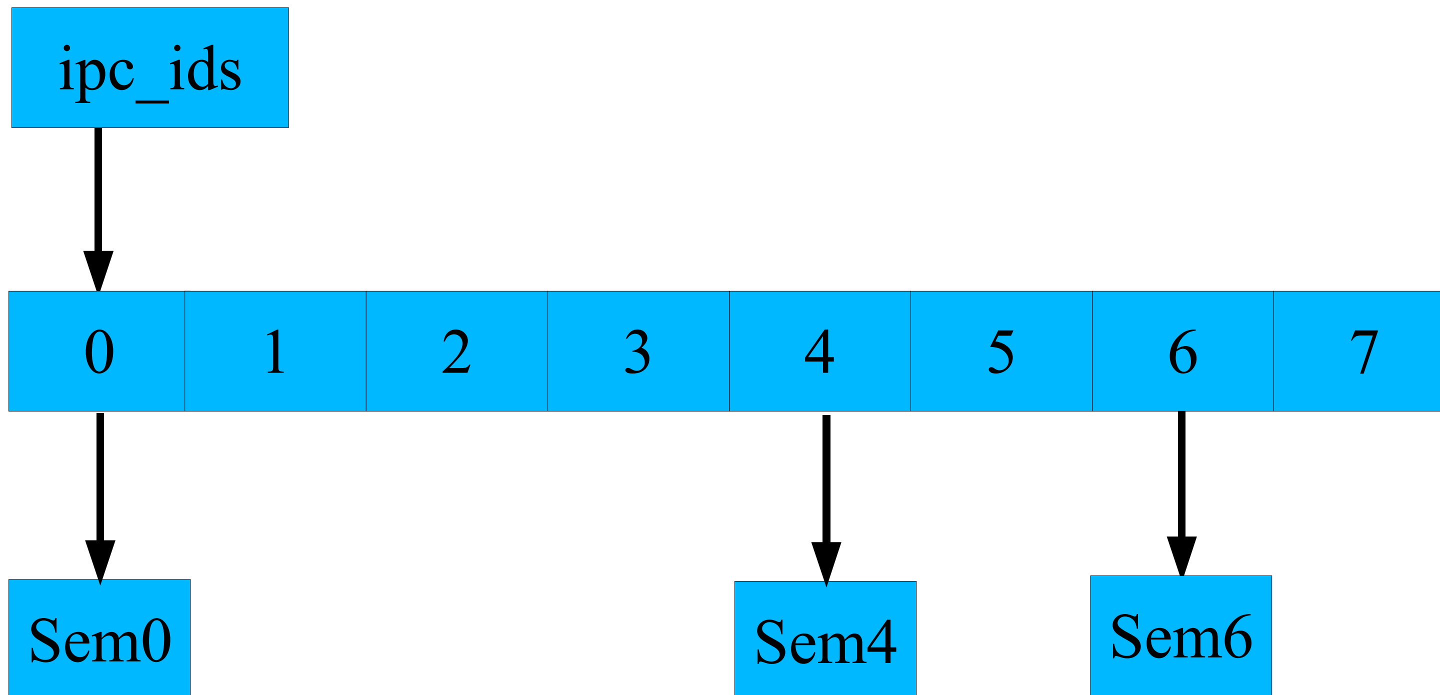
RCU Transformational Patterns

- Substitute Copy for Original *
- Impose Level of Indirection
- Mark Obsolete Objects
- Ordered Update With Ordered Read
- Global Version Number
- Stall Updates

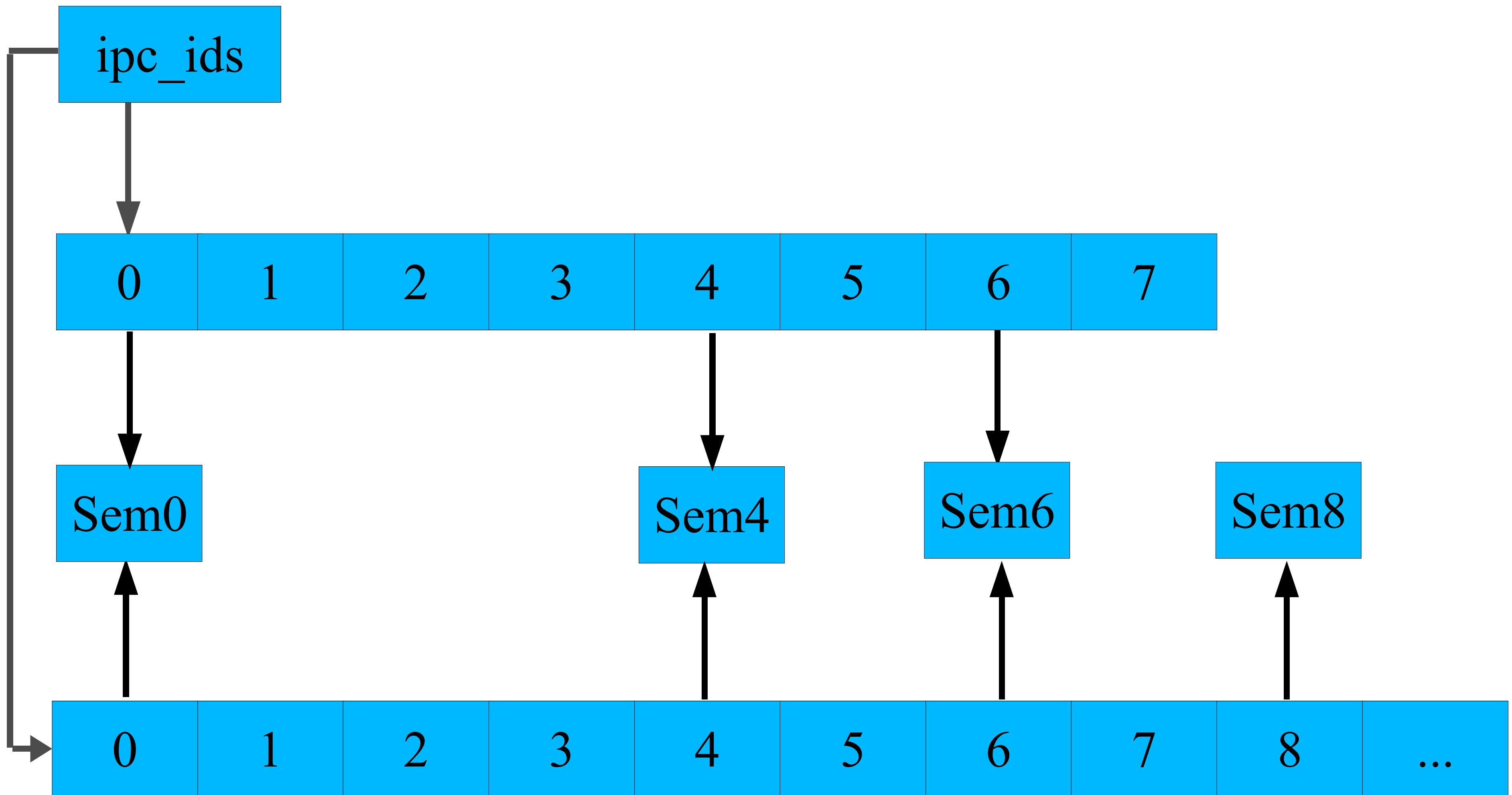
Substitute Copy For Original

- In its pure form, RCU relies on atomic updates of a single value
  - Most CPUs support this
- If data structure requires multiple updates that must appear atomic to readers
  - Must hide updates behind a single atomic operation in order to apply RCU
- To provide atomicity:
  - Make a copy, update the copy, then substitute the copy for the original

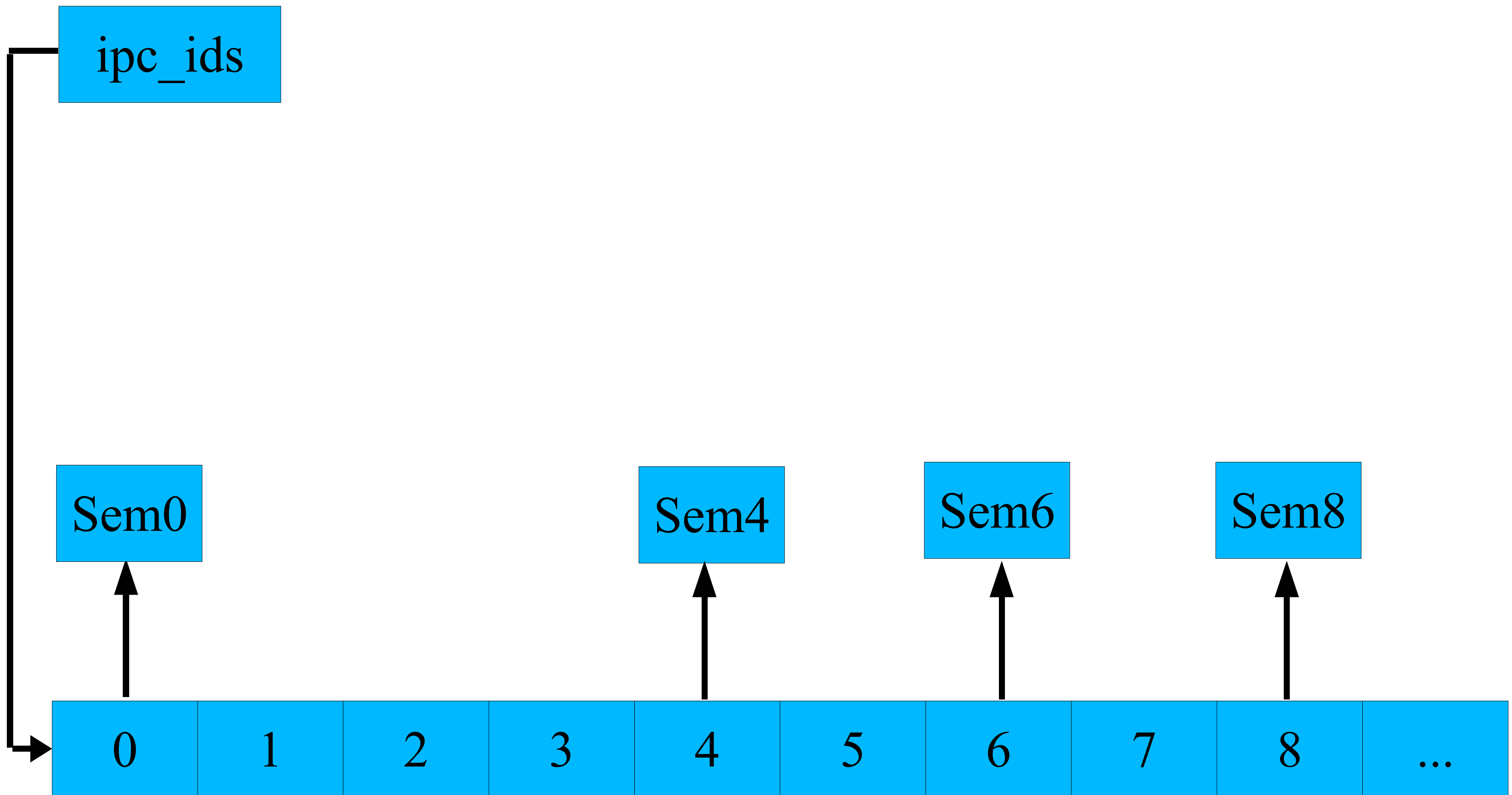
Substitute Copy Animation



Substitute Copy Animation



Substitute Copy Animation



RCU Transformational Patterns

- Substitute Copy for Original (2)
- Impose Level of Indirection (~1)
- Mark Obsolete Objects (2)
- Ordered Update With Ordered Read (3)
- Global Version Number (2)
- Stall Updates (~1)

Summary and Conclusions

Summary and Conclusions (1)

- RCU can provide order-of-magnitude speedups for read-mostly data structures
  - RCU optimal when less than 10% of accesses are updates over wide range of CPUs
  - RCU projected to remain useful in future CPU architectures
- In Linux 2.6 kernel, RCU provided excellent performance with little added complexity

Summary and Conclusions (2)

- RCU best when designed in from the start
  - RCU added late to 2.6 kernel, limited changes feasible after Halloween feature freeze
  - Interesting to see what can be done given the ability to make major changes in 2.7
- Use of design patterns key to RCU
  - RCU consistency semantics require transformation of some algorithms
  - Transformational design patterns can be used

Future Work

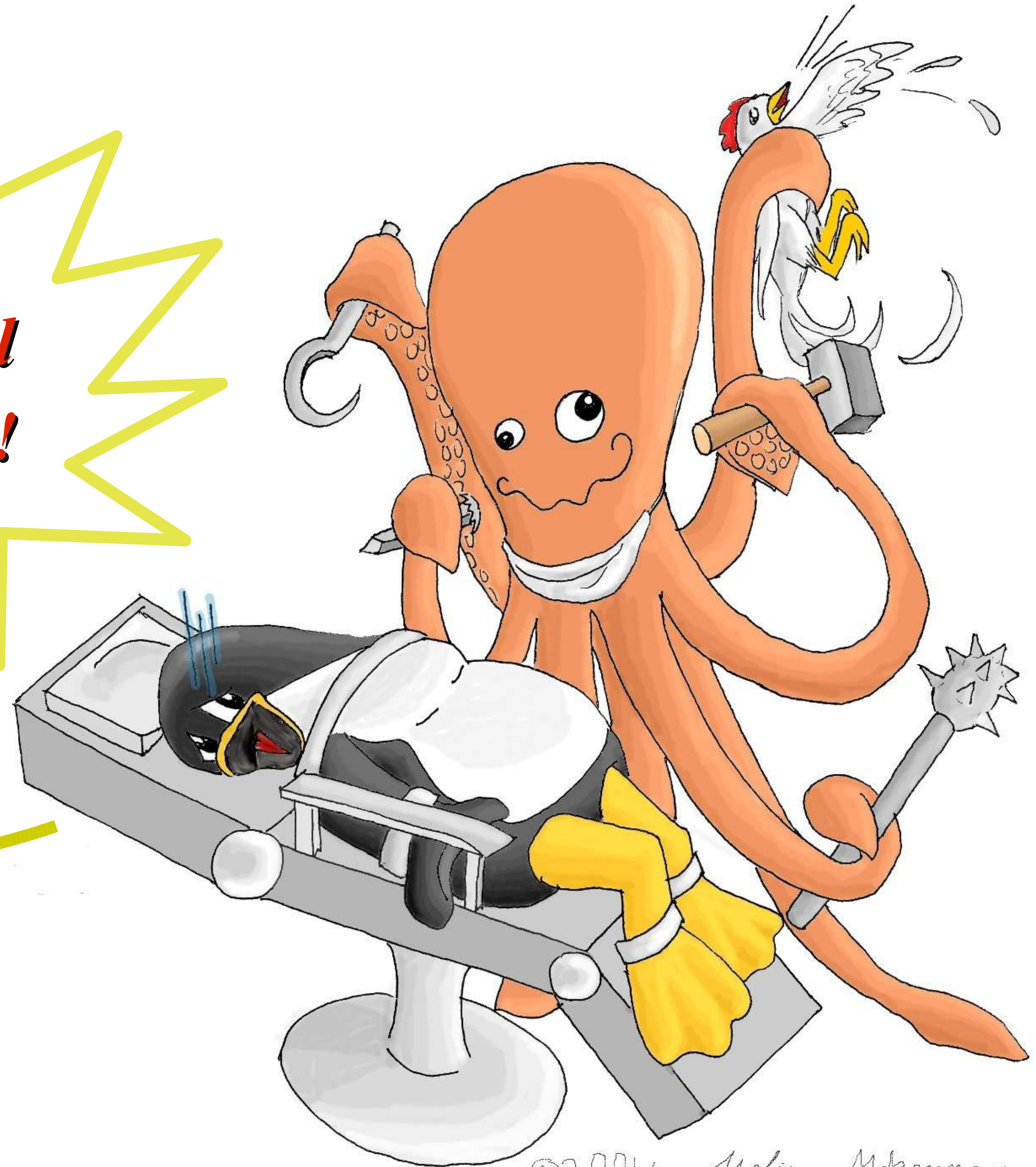
- Formal model of RCU's semantics
- Formal model of consistency semantics for algorithms
- Tools to automatically transform algorithms into a form consistent with RCU's semantics
- Tools to automatically generate RCU from code that uses locking
- Apply RCU to other computational environments

RCU Thesis Publications

1. McKenney et al. "Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications", to appear at USENIX/UseLinux, 6/2004.
2. McKenney, "Locking performance on different CPUs", linux.conf.au, 1/2004.
3. McKenney et al. "Scaling dcache with RCU", Linux Journal, 1/2004.
4. McKenney, "Using RCU in the Linux 2.6 kernel", Linux Journal, 10/2003.
5. Arcangeli et al. "Using read-copy update techniques for System V IPC in the Linux 2.5 kernel", FREENIX, 6/2003.
6. Appavoo et al. "Enabling autonomic behavior in systems software with hot swapping", IBM Systems Journal, 1/2003.
7. McKenney et al. "Read-copy update", Ottawa Linux Symposium, 6/2002.
8. McKenney et al. "Read-copy update", Ottawa Linux Symposium, 7/2001.

- 24 additional publications, 14 patents, 22 patents pending.

Use
the right tool
for the job!!!



BACKUP