Linux Kernel Scalability: Using the Right Tool for the Job

Paul E. McKenney
IBM Beaverton

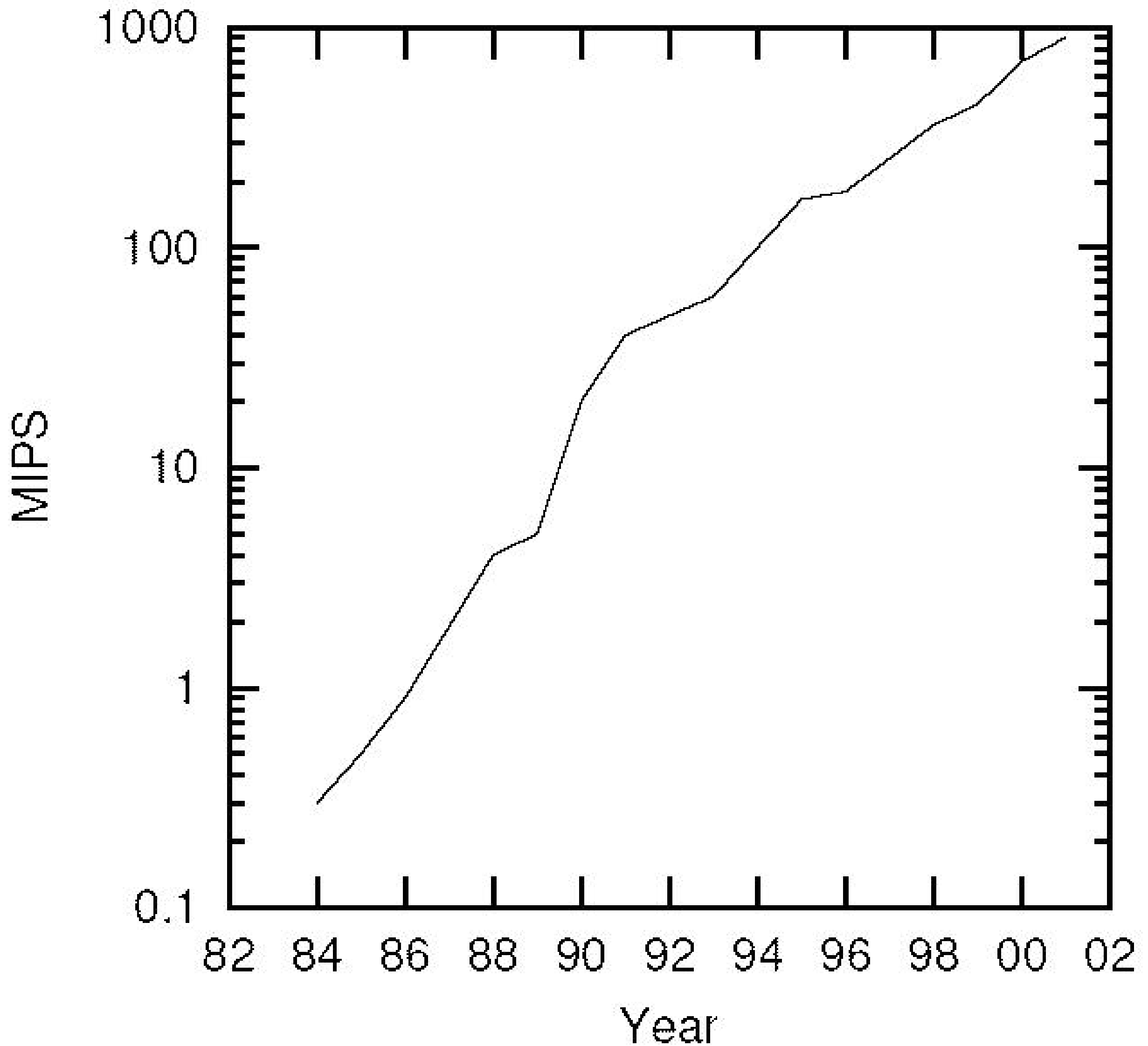
2004 Ottawa Linux Symposium

Overview

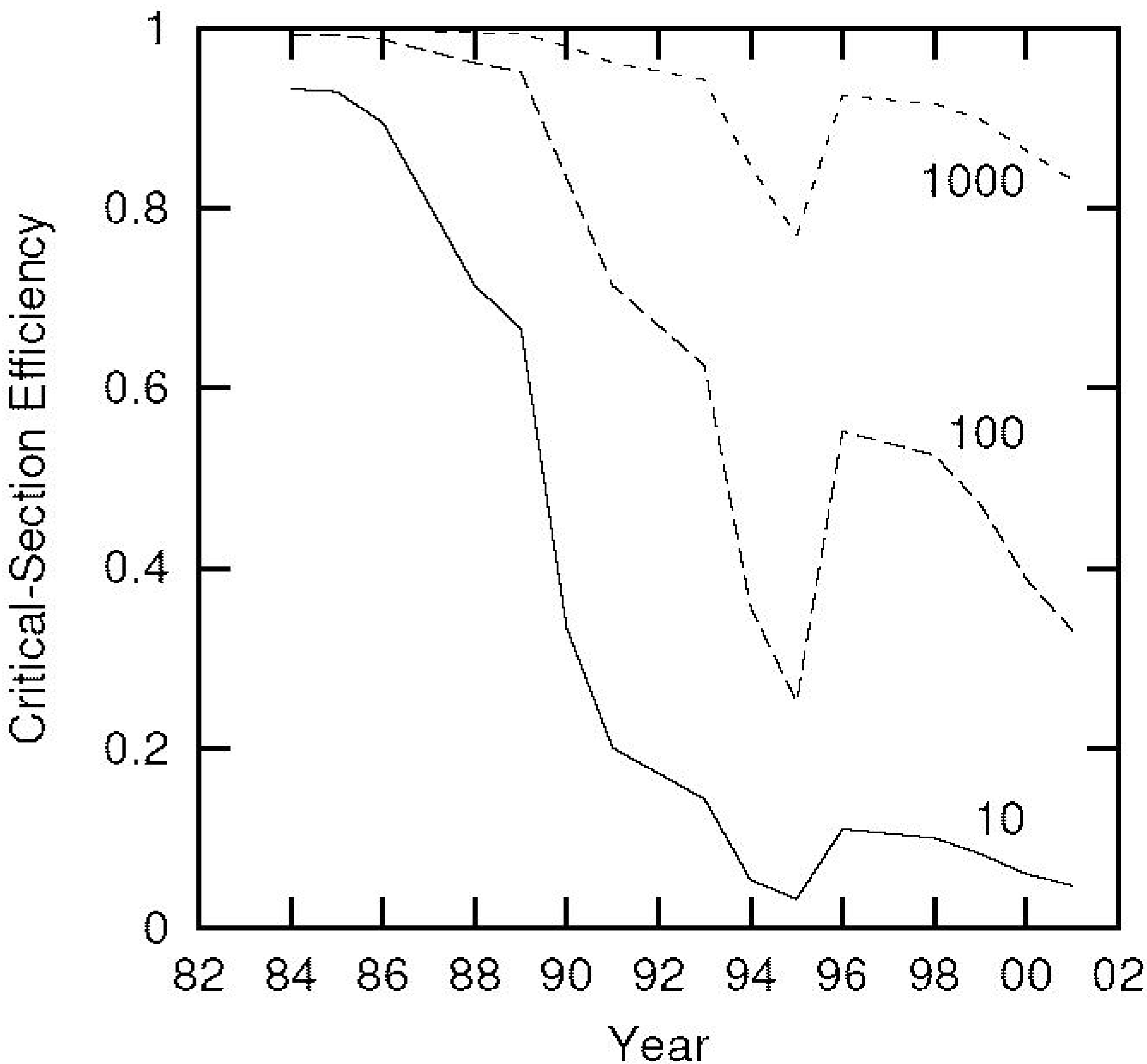
- Moore's Law and SMP Software
- Synchronization Usage
  - Locking, Counting, NBS, and RCU
  - Putting it All Together
- The Road Ahead
- Summary

Moore's Law and SMP Software

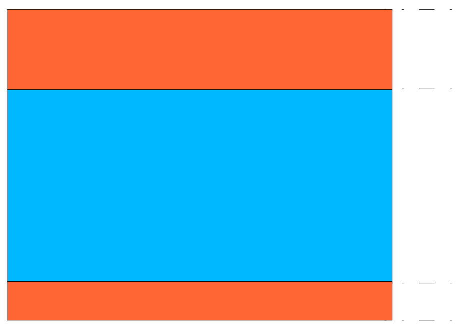
Instruction Speed Increased



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

Instruction/Pipeline Costs on a 4-CPU 700MHz Pentium®-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

Visual Demonstration of Latency

cmpxchg transfer & invalidate: 360.9ns

Each pair of nanoseconds represents
up to about three 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
  - 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

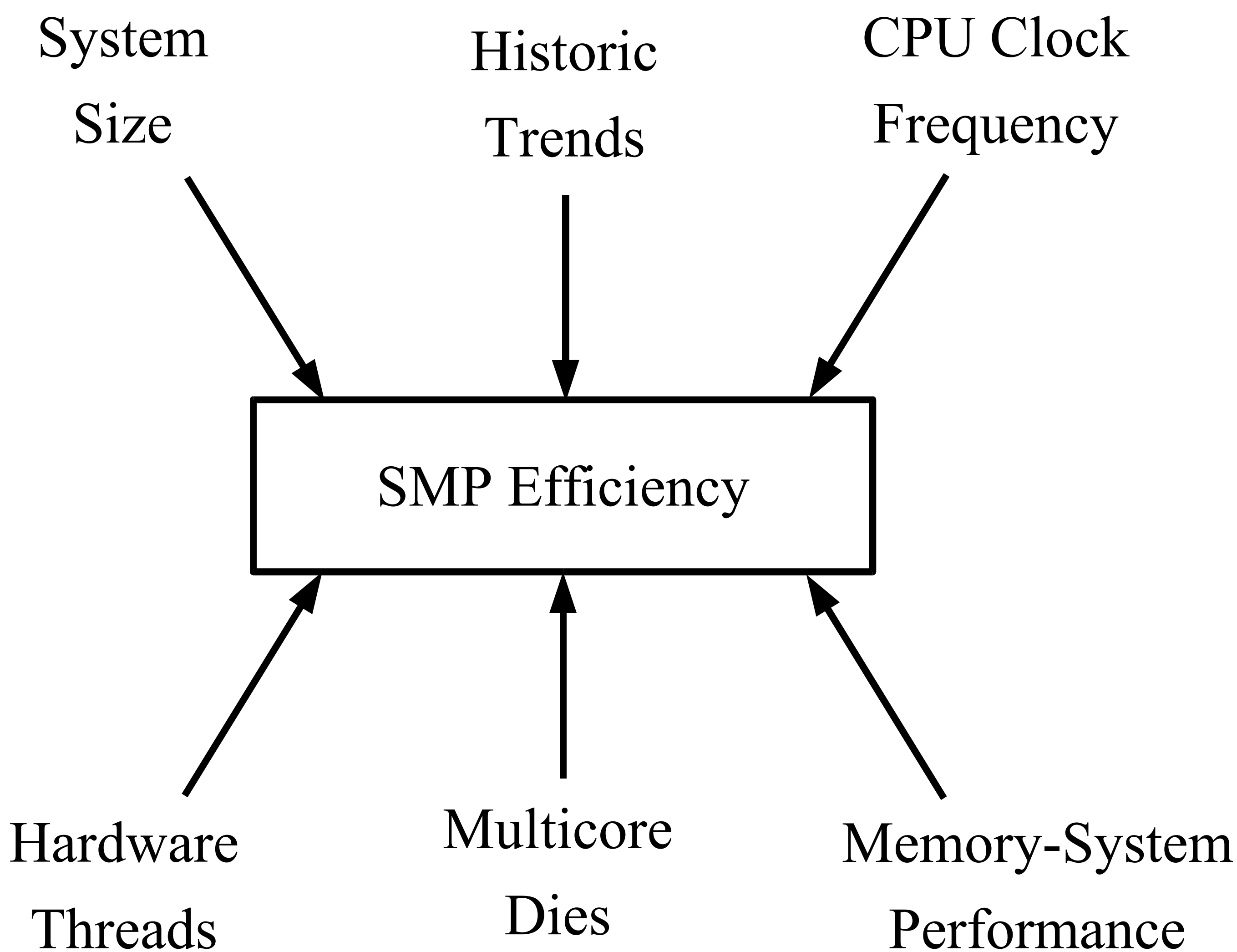
What is Going On? (2/3)

- 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)

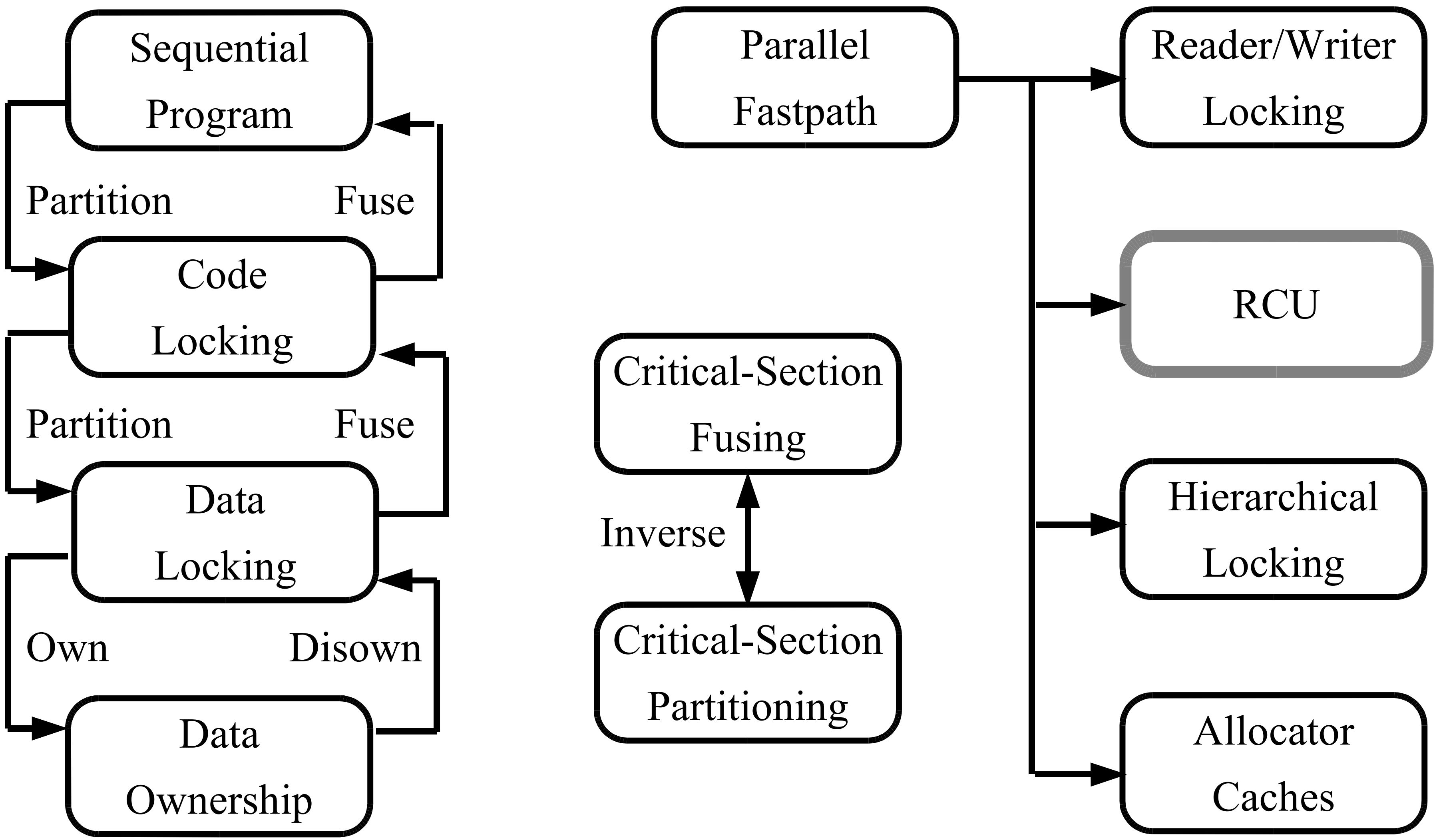
- 1984: The main issue was lock contention
- 2004: Even if lock contention is eliminated, critical-section efficiency must be addressed!!!
  - Even if the lock is *always* free when acquired, performance is seriously degraded

Forces Acting on SMP Efficiency



Locking

Locking Designs



Sequential Program

- If a single CPU can do the job you need, why are you messing with SMP and locking???
  - Not enough challenge in your life???
  - You like slowing things down by including SMP primitives?

Code Locking

- AKA “global locking”
  - Only one CPU at a time in given code path
- Very simple, but no scaling
- Examples:
  - 2.4 runqueue_lock
  - dcache_lock
    - Guards all dcache in 2.4, dcache updates in 2.6
  - rcu_ctrlblk.mutex

Data Locking

- But isn't it *all* data locking?
  - Yes, but... Data locking associates locks with individual data items rather than code paths
    - 2.4: “spin_lock_irq(&runqueue_lock);”
    - 2.6: “spin_lock_irq(&rq->lock)”
  - CPUs process different data items in parallel
- Examples:
  - 2.6 O(1) scheduler (per-runqueue locking)
  - 2.6 d_lock (per-dentry locking for path walking)
  - Manfred Spraul RCU_HUGE patch

Data Locking Implications (1)

- How to handle common global structure?
  - Retain global lock for this purpose
    - dcache_lock retained when per-dentry d_lock added
    - Need both locks on many code paths
  - Restructure to eliminate common structure
  - Apply more aggressive locking model
- What if every CPU hits the same data item?
  - mm_lock is great – unless everyone is faulting on the same shared-memory segment...

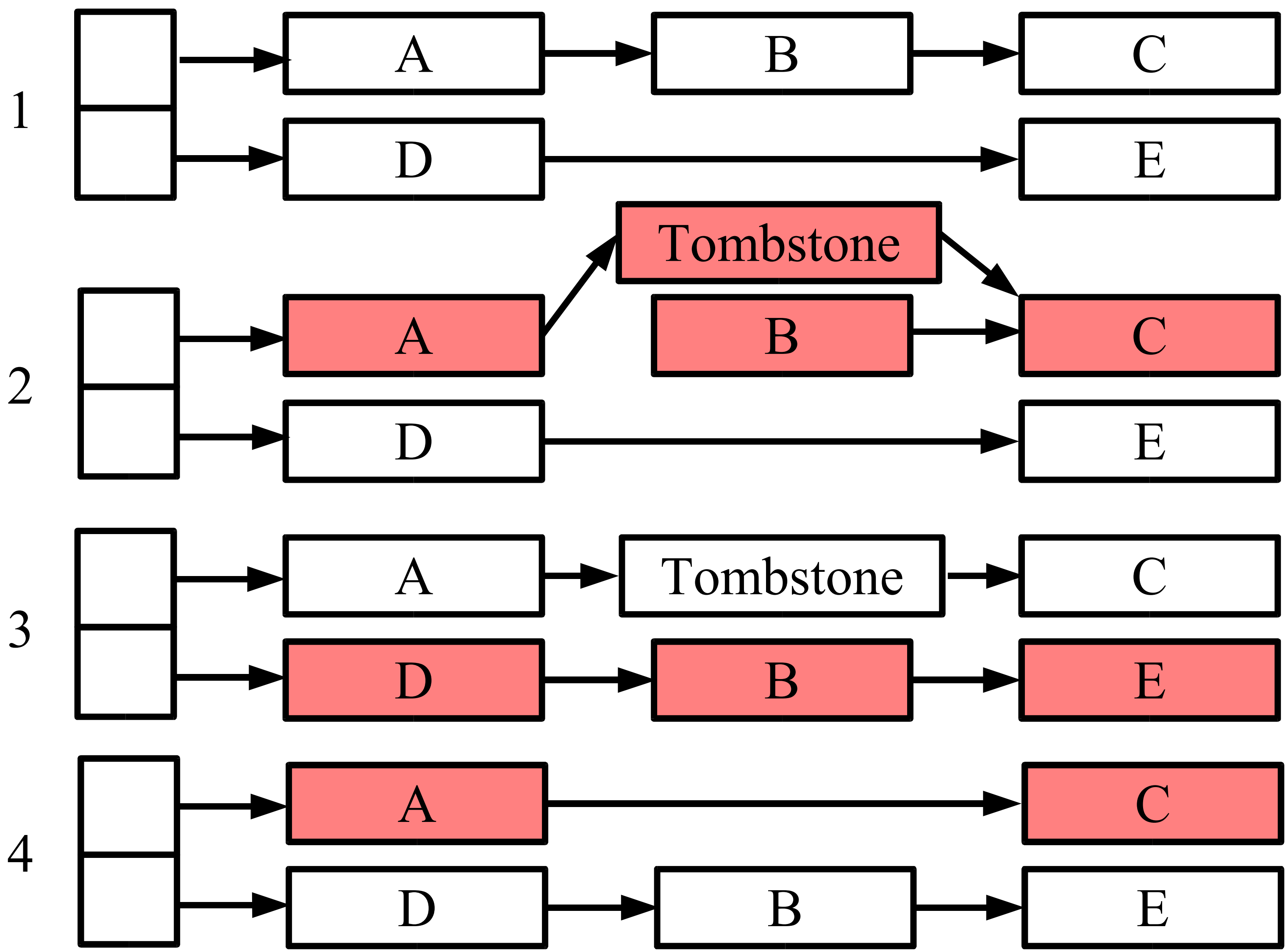
Data Locking Implications (2)

- How to handle two data items concurrently?
  - Acquire locks in order: d_move() in dcache:

```
if (target < dentry) {
        spin_lock(&target->d_lock);
        spin_lock(&dentry->d_lock);
} else {
        spin_lock(&dentry->d_lock);
        spin_lock(&target->d_lock);
}
```

  - Acquire multiple locks only if holding global lock
    - Careful!!! The use of a global lock can easily wipe out any data-locking performance gains!
  - Figure out a way to handle one item at a time

Data Locking: One at a Time



Data Ownership

- DEFINE_PER_CPU(type, name)
  - But it is possible to access others' variables via per_cpu(var, cpu)
  - Used during initialization
  - Also for reading out performance statistics
    - IA64 pfm_proc_show()
    - PPC64 proc_eeh_show()
  - And for coordinating CPUs
    - IA64 wrap_mmu_context()

Data Ownership Implications

- Data completely private to owning CPU
  - Used pervasively throughout Linux kernel
- Incomplete privacy:
  - Owning CPU updates, others read
    - Statistics (next slide)
  - Other CPUs update only if owning CPU offline
    - Didn't see any, may have missed some...
  - Owning CPU reads, others update (via sysfs)
    - store_smt_snooze_delay()

Owning CPU Updates

- TCP stats gathered via IP_INC_STATS_BH
- TCP stats readout

```
static unsigned long
__fold_field(void *mib[], int offt)
{
        unsigned long res = 0;
        int i;
        for (i = 0; i < NR_CPUS; i++) {
        if (!cpu_possible(i))
                continue;
        res += *((unsigned long *)(((void *)per_cpu_ptr(mib[0], i)) +
                                                offt));
        res += *((unsigned long *)(((void *)per_cpu_ptr(mib[1], i)) +
                                                offt));

        }
return res;
}
```

Owning CPU Reads

- PPC64 idle-loop control of hardware threads

```
unsigned long start_snooze;
unsigned long *smt_snooze_delay = &__get_cpu_var(smt_snooze_delay);
while (1) {
        oldval = test_and_clear_thread_flag(TIF_NEED_RESCHED);
        if (!oldval) {
                set_thread_flag(TIF_POLLING_NRFLAG);
                start_snooze = __get_tb() +
                        *smt_snooze_delay * tb_ticks_per_usec;
                while (!need_resched()) {
                        if (*smt_snooze_delay == 0 ||
                            __get_tb() < start_snooze) {
                              HMT_low(); /* Low thread priority */
                              continue;
                        }
                        HMT_very_low(); /* Low power mode */

                        . . .
```

Data Ownership: Function Shipping

- mm/slab.c

```
static void do_drain(void *arg)
{
        kmem_cache_t *cachep = (kmem_cache_t*)arg;
        struct array_cache *ac;
        check_irq_off();
        ac = ac_data(cachep); /* Returns ptr to per-CPU element. */
        spin_lock(&cachep->spinlock);
        free_block(cachep, &ac_entry(ac)[0], ac->avail);
        spin_unlock(&cachep->spinlock);
        ac->avail = 0;
}
static void drain_cpu_caches(kmem_cache_t *cachep)
{
        smp_call_function_all_cpus(do_drain, cachep);
        . . .
}
```

Parallel Fastpath

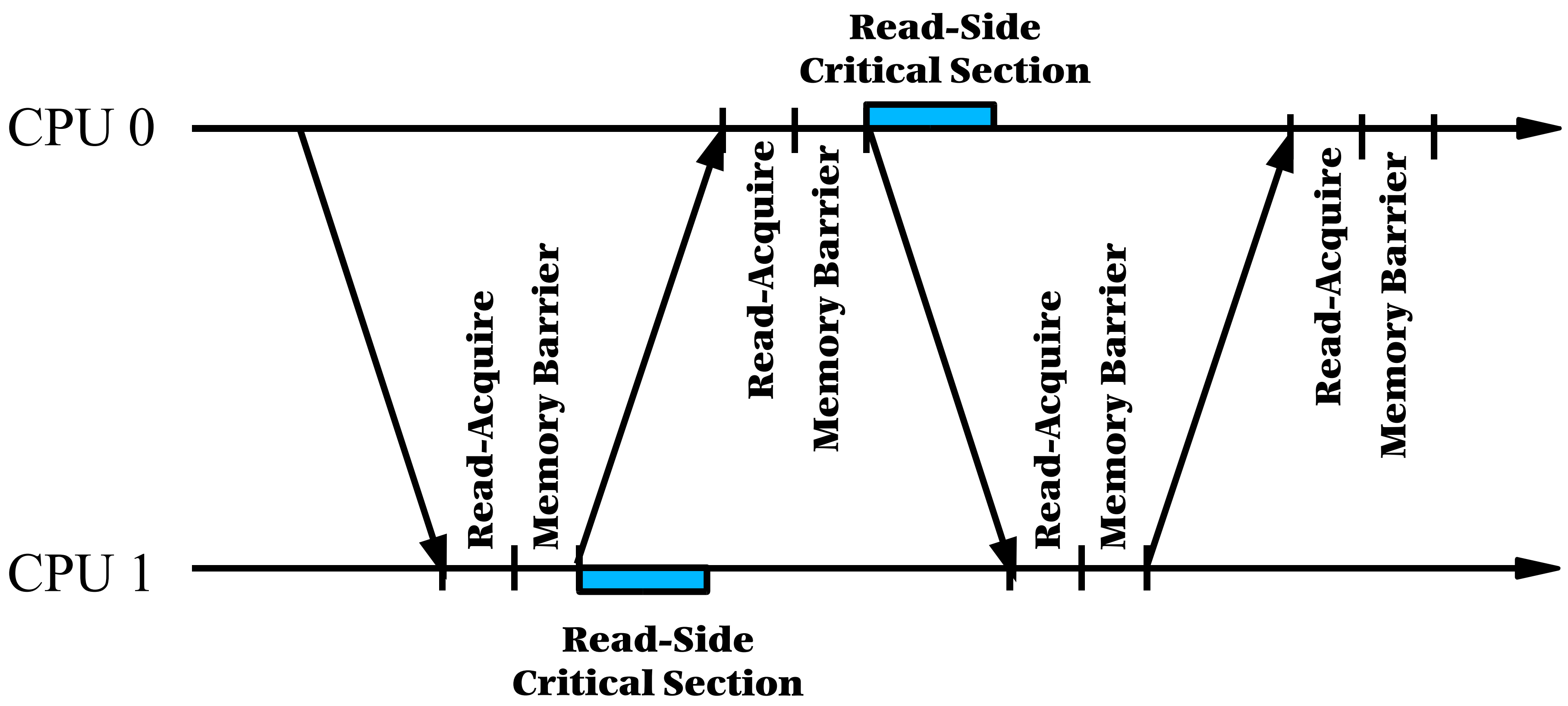
- Make the common case fast, the uncommon case as simple as possible
  - Reader-writer locking
  - RCU (more on this later...)
  - Hierarchical locking
  - Allocator caches

Reader-Writer Locking

- Use for large read-side critical sections.
- get_task() is an example of good usage
  - Might have 1000s of processes
  - Releases lock before returning pointer...

```
read_lock(&tasklist_lock);
for_each_process(task){
    if(task->pid == pid){
        ret = task;
        break;
    }
}
read_unlock(&tasklist_lock);
```

Do Not Use rwlock_t for Short Read-Side Critical Sections



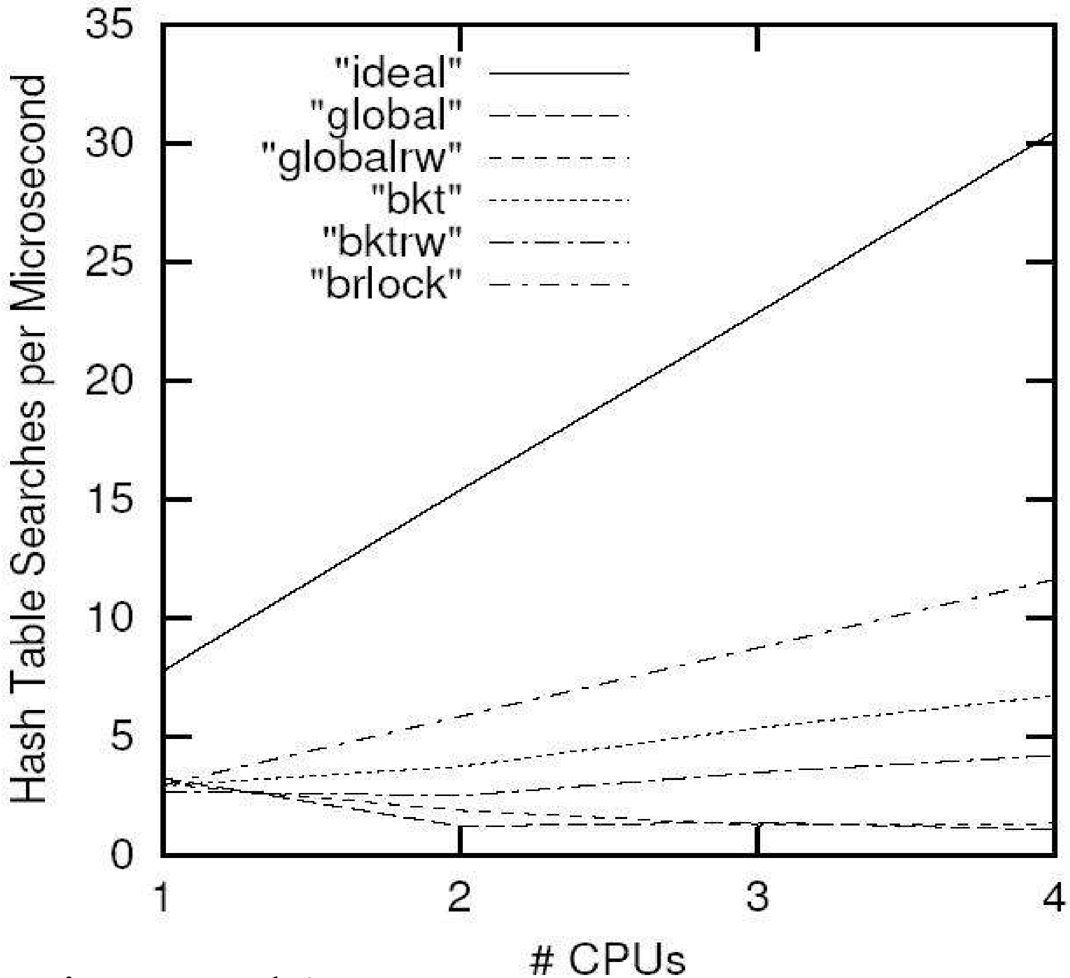
Performance Comparison: What Benchmark to Use?

- 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 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

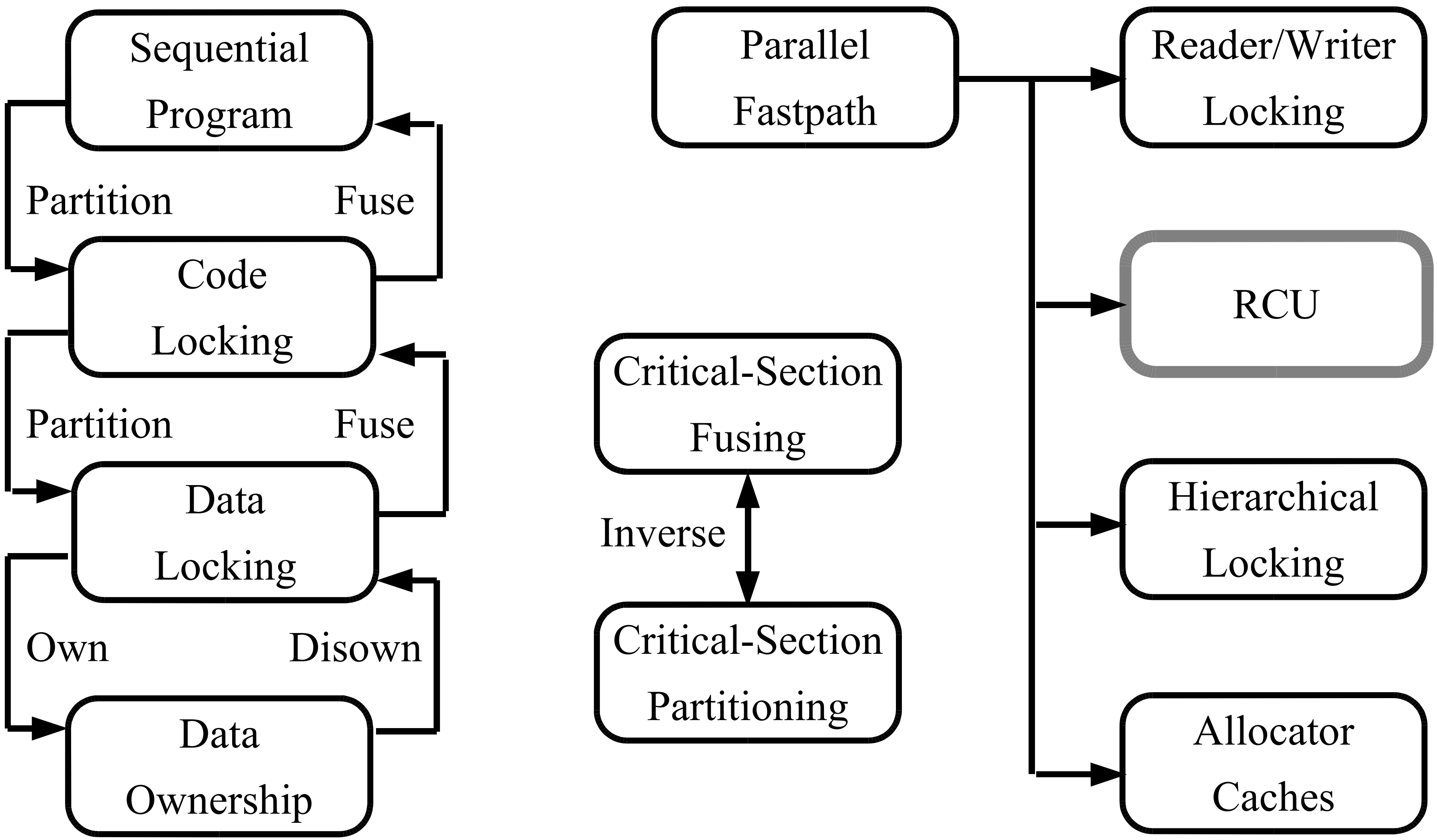
Locking Performance



Extra CPUs not buying much!

Note: workload fits in cache.

Locking Designs



Counting

Counters: Workload Dependent

- No blocking while holding or releasing count
- Updates rare (just use a global counter!!!)
- Updates common:
  - References rare:
    - "Fuzzy" readout permissible
    - Exact readout required
  - References frequent:
    - Just use seqlock_t!!!
    - Memory-barrier/atomic overhead too much and large value
      - "Fuzzy" readout permissible
      - References are checks for rarely exceeded range
- Otherwise, innovation required

Updates Common, References Rare (1)

- Statistical counters!!! Per-CPU counters...
- Fuzzy readout: just need to manage value
  - Reference released on same CPU as acquired (or monotonic counters)
    - Simple per-CPU counters, sum them without lock
    - See previous data-ownership example
  - CPUs can release other CPUs' references
    - Need to migrate counts in some cases
      - For example, if it is important to detect zero crossings
      - Rusty has been working on a prototype, crude version here

Updates Common, References Rare (2)

- Exact readout at arbitrary time and value?
- Must stall readers... And add complexity...
  - br_read_lock() to update counter, br_write_lock() to read counter (can use per_cpu() in 2.6)
    - Moderate latency for readout
    - Moderate overhead for read
  - RCU and flags, readers block if flag set
    - Untried, not clear this is a good approach
- Friendly advice: tolerate uncertainty!!!

brlock Counter

```
/* Increment counter. */
br_read_lock(BR_MY_LOCK);
__get_cpu_var(my_count)++;
br_read_unlock(BR_MY_LOCK);

/* Read out counter. */
br_write_lock(BR_MY_LOCK);
for_each_cpu(i)
        sum += per_cpu(my_count, i);
br_write_unlock(BR_MY_LOCK);
```

- Yes, you do read-acquire the lock to write the variable and vice versa!!!
- We are really using (abusing!) the brlock as a local-global rather than a reader-writer lock

2.6 Implementation of brlock Counter

```
/* Increment counter. */
spin_lock(__get_cpu_var(mylock));
__get_cpu_var(mycount)++;
spin_unlock(__get_cpu_var(mylock));

/* Read out counter. */
for_each_cpu(i) {
        spin_lock(per_cpu(mylock, i));
        sum += per_cpu(mycount, i);
}
for_each_cpu(i) {
        spin_unlock(per_cpu(mylock, i));
}
```

- A few more lines of on the read-out side, but two rather than three loops
- Inline functions helpful if frequently used

“Big Reference Count”

- Maintain per-CPU counters
- But also provide a global counter
  - Value is sum of all counters
  - Ship counts between per-CPU and global count
  - Apply a large bias to the count
- Use the per-CPU counters in fastpath
- When checking for zero, remove the bias
  - Force use of only global counter

Big Reference Count Data

- Per-CPU component

```
struct brefcnt_percpu {
        int      brcp_count;     /* Per-CPU ctr.  Should interlace */
}
```

- Global component

```
struct brefcnt {
        spinlock_t brc_mutex;   /* Guards all but brc_percpu. */
        long     brc_global;     /* Global portion of count. */
        void     (*brc_zero)(struct brefcnt *r, void *arg);
                                 /* Function to call zero count. */
        void     *brc_arg;       /* 2nd argument for brc_zero. */
        struct brefcnt_percpu *brc_percpu ____cacheline_aligned;
        int      brc_local;      /* 1=use local counts, 0=use gbl. */
};
```

- Converging with krefcnt would be challenge!!!

Big Reference Count Increment

```
void brefcnt_inc(struct brefcnt *r)
{
	int val;

	if (likely(r->brc_local)) {
		val = r->brc_percpu[smp_processor_id()].brcp_count++;
		if (unlikely(val > 2 * BREFCNT_PER_CPU_TARGET)) {
			r->brc_percpu[smp_processor_id()].brcp_count
				-= BREFCNT_PER_CPU_TARGET;
			spin_lock(&r->brc_mutex);
			r->brc_global += BREFCNT_PER_CPU_TARGET;
			spin_unlock(&r->brc_mutex);
		}
		return;
	}
	spin_lock(&r->brc_mutex);
	r->brc_global++;
	spin_unlock(&r->brc_mutex);
}
```

Big Reference Count Decrement

```
void brefcnt_dec(struct brefcnt *r)
{
        long val;
        int *pcp = &r->brc_percpu[smp_processor_id()].brcp_count;
        if (likely(r->brc_local)) {
                if (*pcp > 1) {
                        (*pcp)--;
                        return;
                }
                spin_lock(&r->brc_mutex);
                r->brc_global -= BREFCNT_PER_CPU_TARGET;
                spin_unlock(&r->brc_mutex);
                *pcp += BREFCNT_PER_CPU_TARGET - 1;
                return;
        }
        spin_lock(&r->brc_mutex);
        val = --r->brc_global;
        spin_unlock(&r->brc_mutex);
        if ((val == 0) && (r->brc_zero != NULL)) {
                r->brc_zero(r, r->brc_arg);
        }
}
```

Big Refcount Remove Bias

```
void brefcnt_remove_bias(struct brefcnt *r)
{
        int i;
        long val;

        spin_lock(&r->brc_mutex);
        r->brc_local = 0;
        spin_unlock(&r->brc_mutex);

        synchronize_kernel();  /* wait for racing incs/decs. */

        spin_lock(&r->brc_mutex);
        for_each_cpu(i) {
                r->brc_global += r->brc_percpu[i].brcp_count;
                r->brc_percpu[i].brcp_count = 0;
        }
        val = (r->brc_global -= BREFCNT_BIAS);
        spin_unlock(&r->brc_mutex);
        if ((val == 0) && (r->brc_zero != NULL))
                r->brc_zero(r, r->brc_arg);
}
```

Updates Rare, References Common

- Just use seqlock_t!
- Unless you cannot afford the atomic-instruction and memory-barrier overhead
  - If you really believe you cannot afford the atomic-instruction and memory-barrier overhead, do the measurements again, and *carefully* analyze the results!!!
  - If you really cannot afford this, you can use big reference count in some special cases

seqlock_t Timer Handling

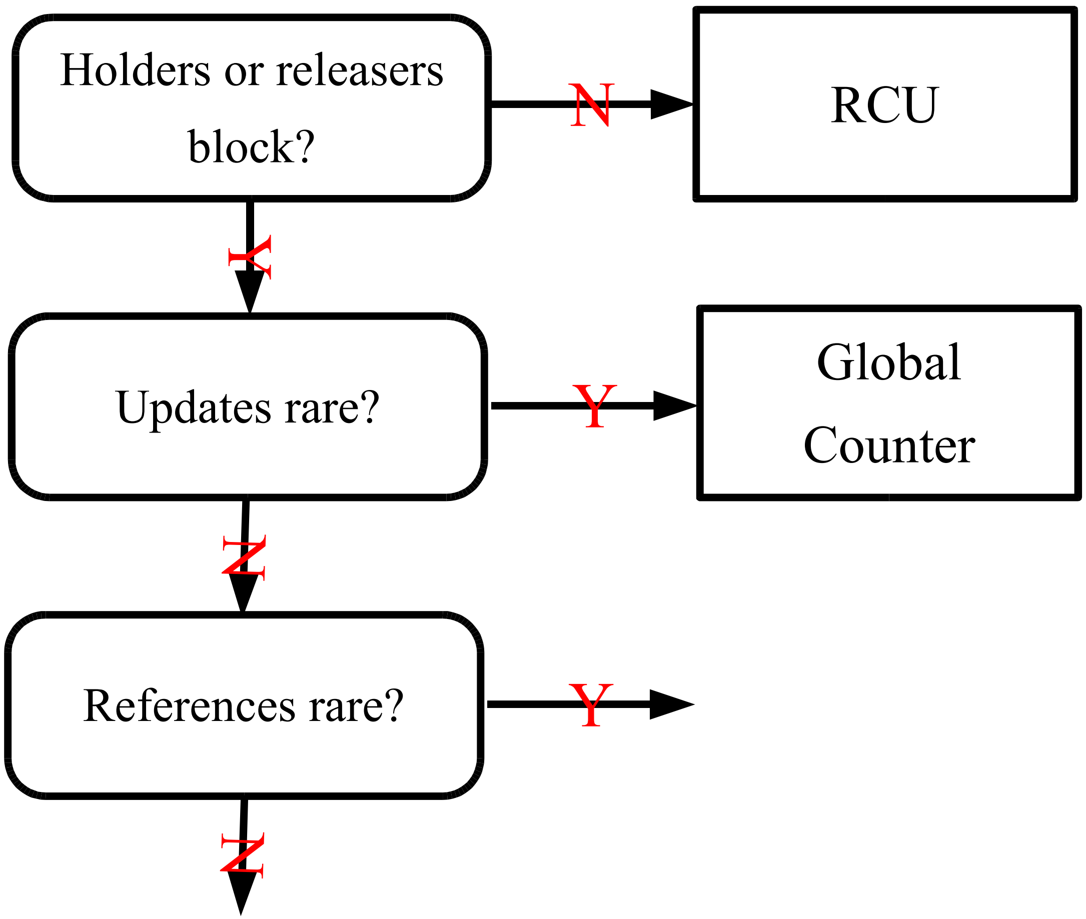
- Timer update

```
write_seqlock(&xtime_lock);
cur_timer->mark_offset();
do_timer_interrupt(irq, NULL, regs);
write_sequnlock(&xtime_lock);
```

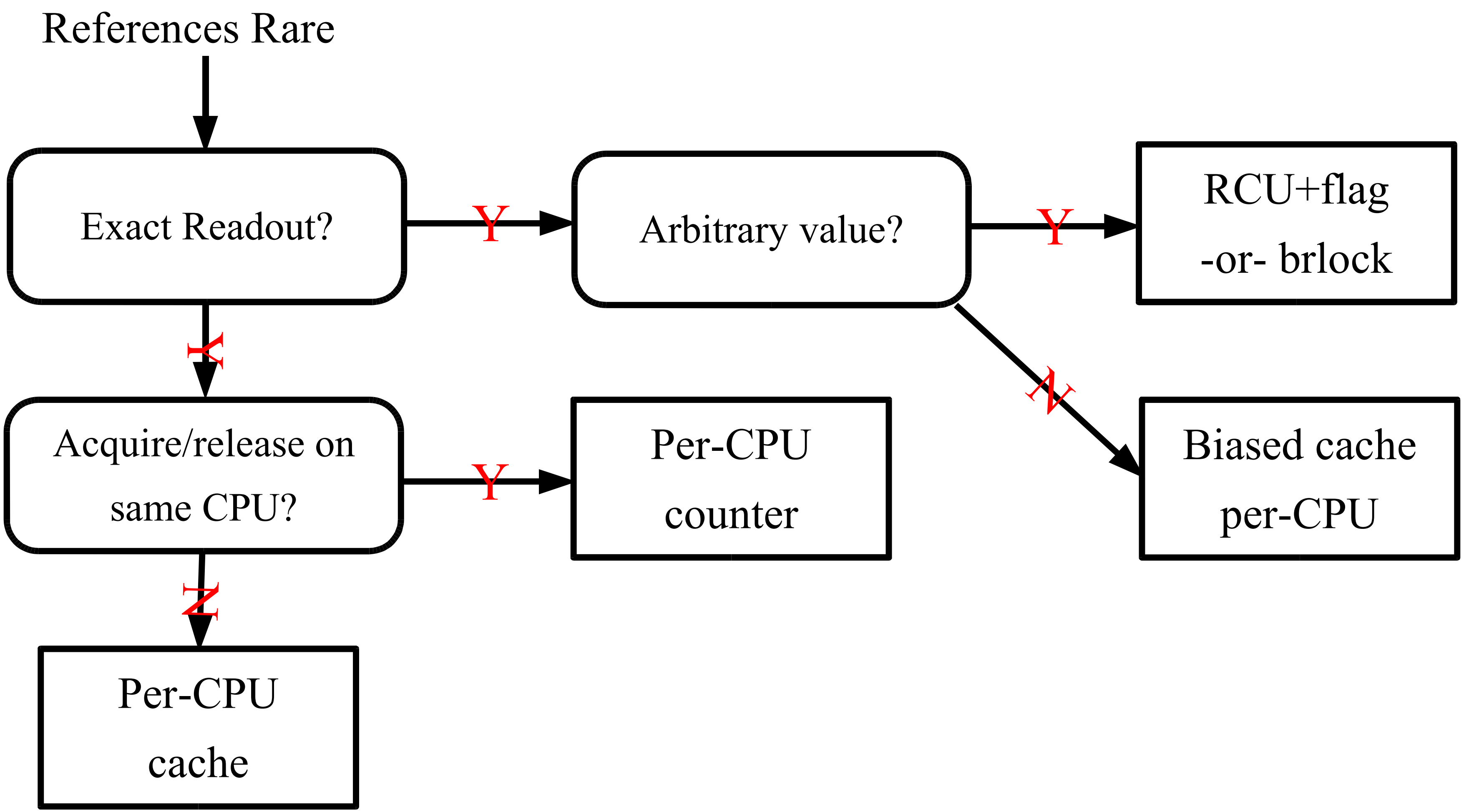
- Timer readout

```
do {
        seq = read_seqbegin_irqsave(&xtime_lock, flags);
        delta_cycles = rpcc() - state.last_time;
        sec = xtime.tv_sec;
        usec = (xtime.tv_nsec / 1000);
        partial_tick = state.partial_tick;
        lost = jiffies - wall_jiffies;
} while (read_seqretry_irqrestore(&xtime_lock, seq, flags));
```

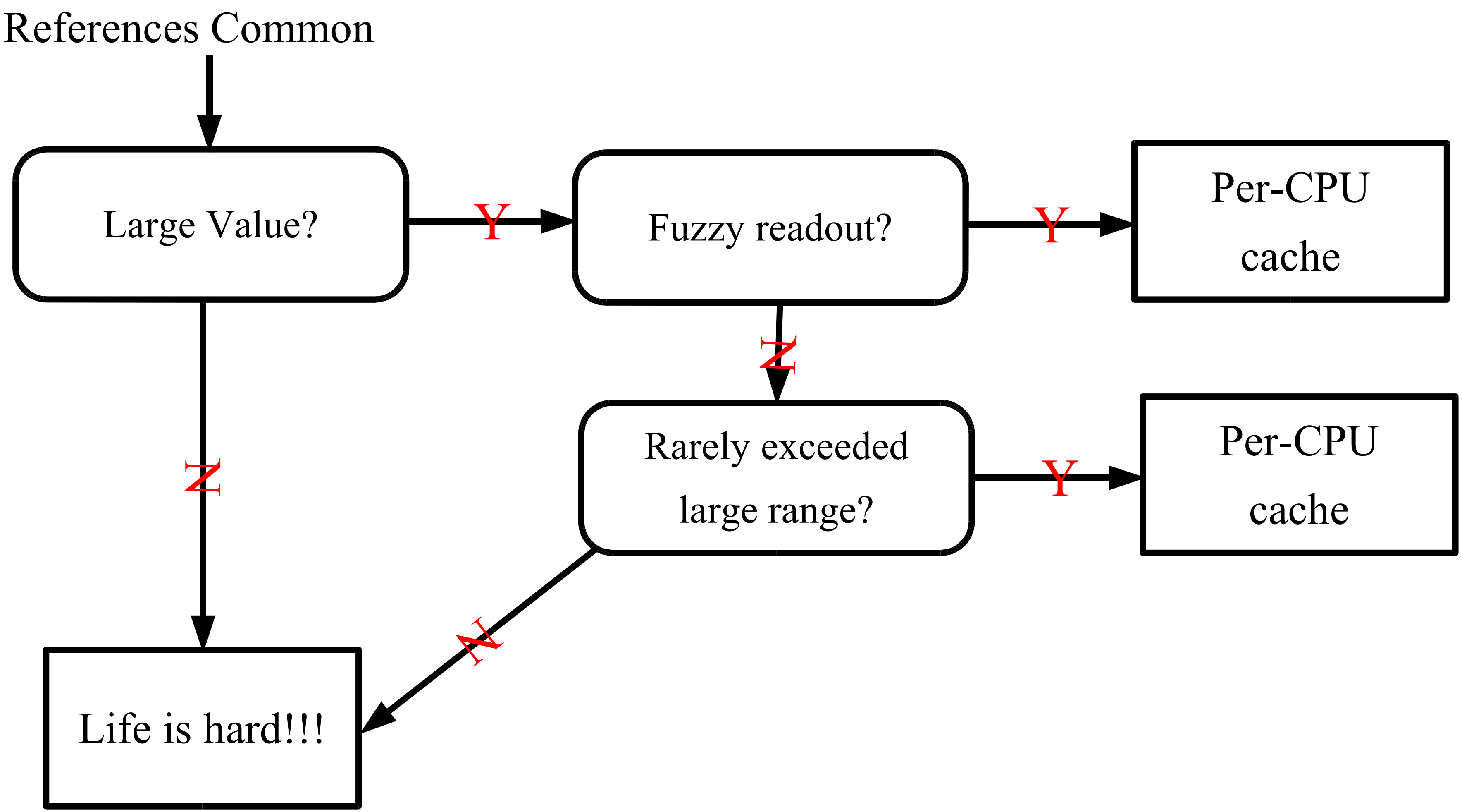
Counter Decision Tree



Counter Decision Tree (Rare Ref)



Counter Decision Tree (Many Ref)



Other Counter Complications

- 64-bit counters on 32-bit machine
- Access from both irq and process context
  - Preemption can have similar effects...
- Need to update other CPUs' counters
- Need agreement on sequence of values
  - Parallel increments of 1, 5, and 7
  - 1, 6, 13? 5, 12, 13? 7, 8, 13?
  - Friendly advice: tolerate dissent!!!

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
  - 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
- Simple "NBS" algorithms in heavy use
  - Atomic-instruction-based algorithms

Why Not NBS All The Time?

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 ←

When to Use NBS?

- Simple NBS algorithm is available
  - Counting (strictly speaking, only by 1)
    - See example from previous section
  - Simple queue/stack management
  - Especially if NBS constraints may be relaxed!
- Workload is update-heavy
  - So that NBS's use of atomic instructions and memory barriers is not causing gratuitous pain

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
- “Linearizability”
  - All CPUs agree on all intermediate states
- Both constraints mostly irrelevant to Linux

RCU

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, deferring destruction
  - Readers must signal GC when done

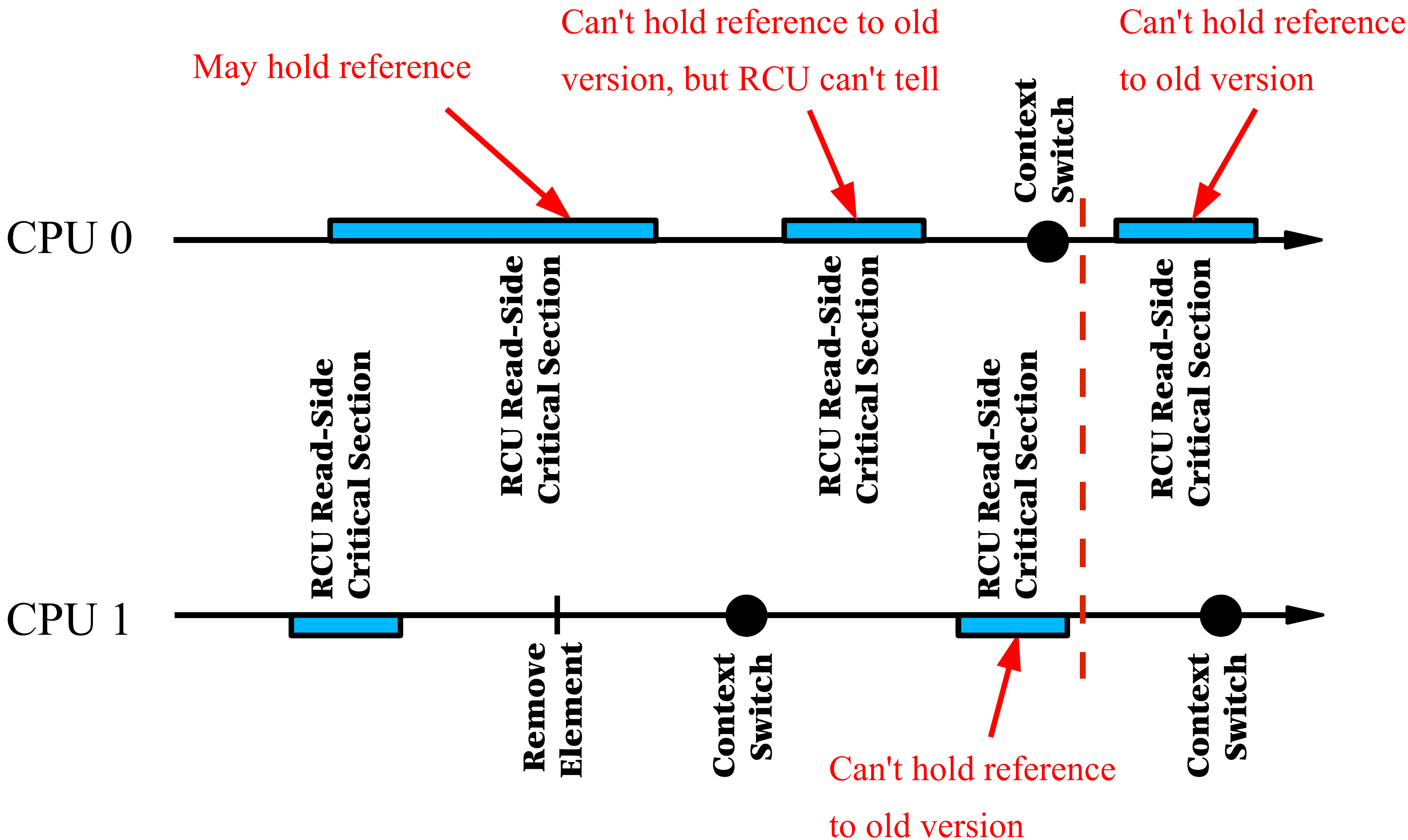
What is RCU? (2)

- 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” garbage collector also incurs some overhead

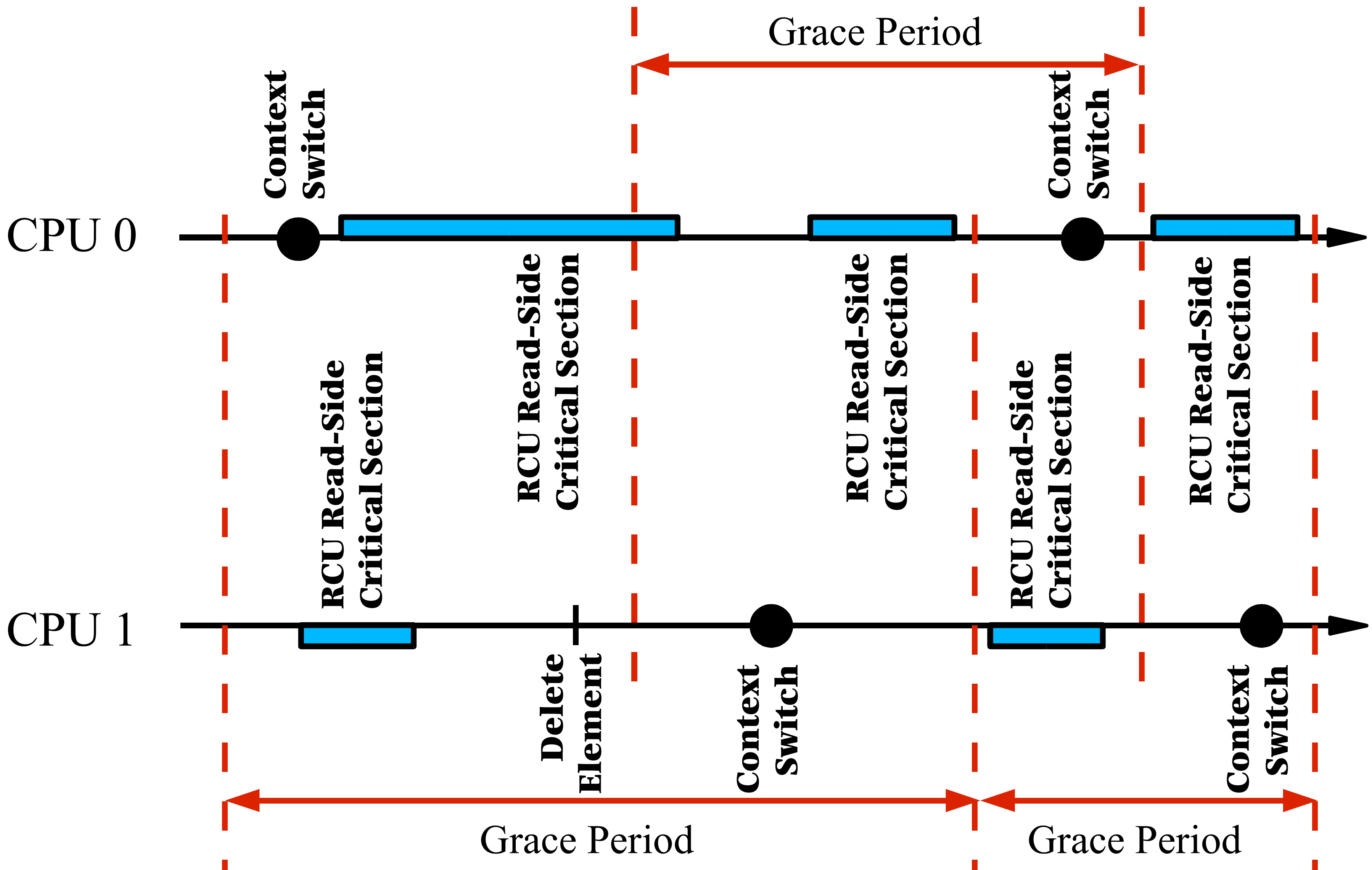
How Can RCU be Fast?

- 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

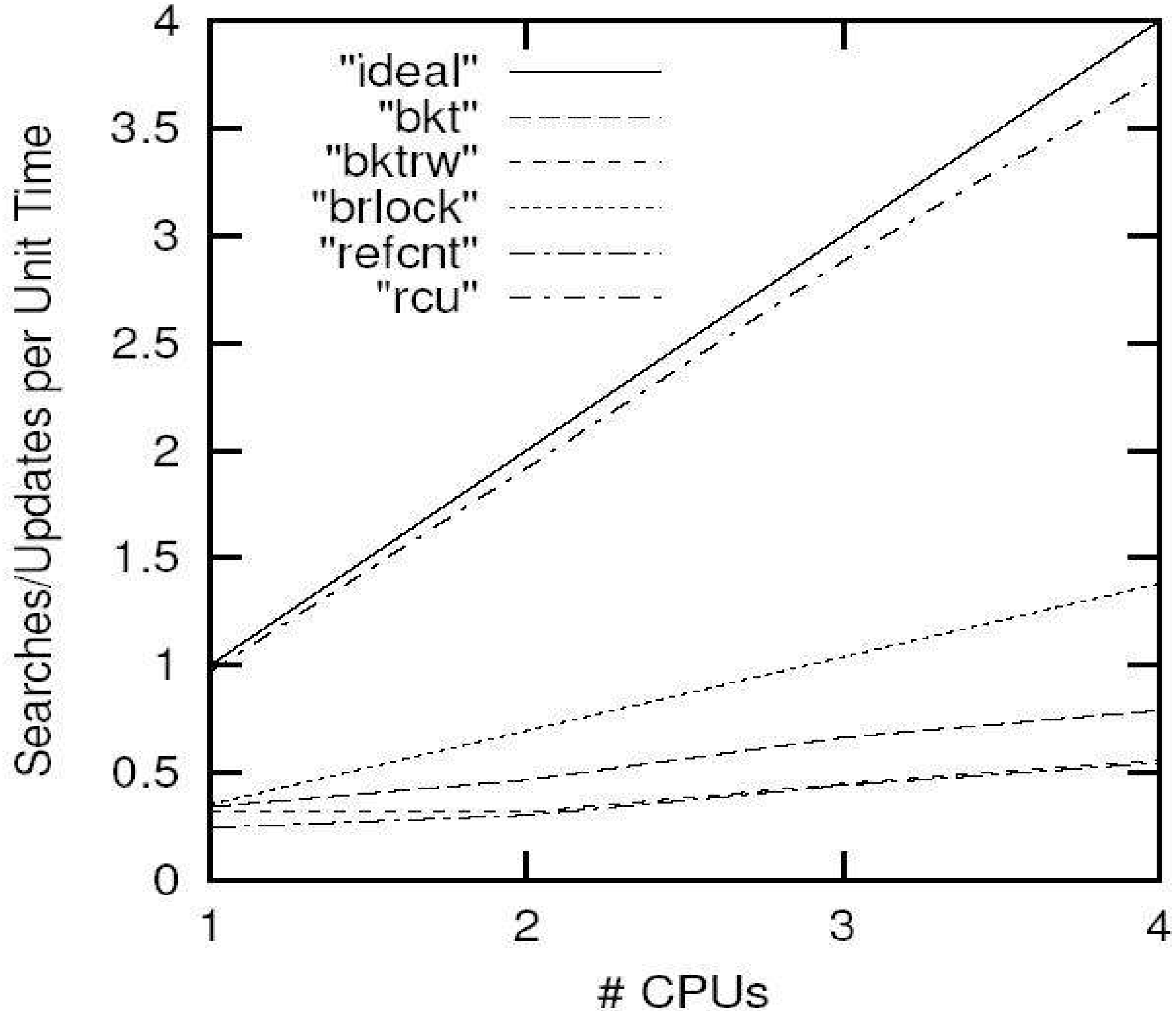
RCU's Deferred Destruction



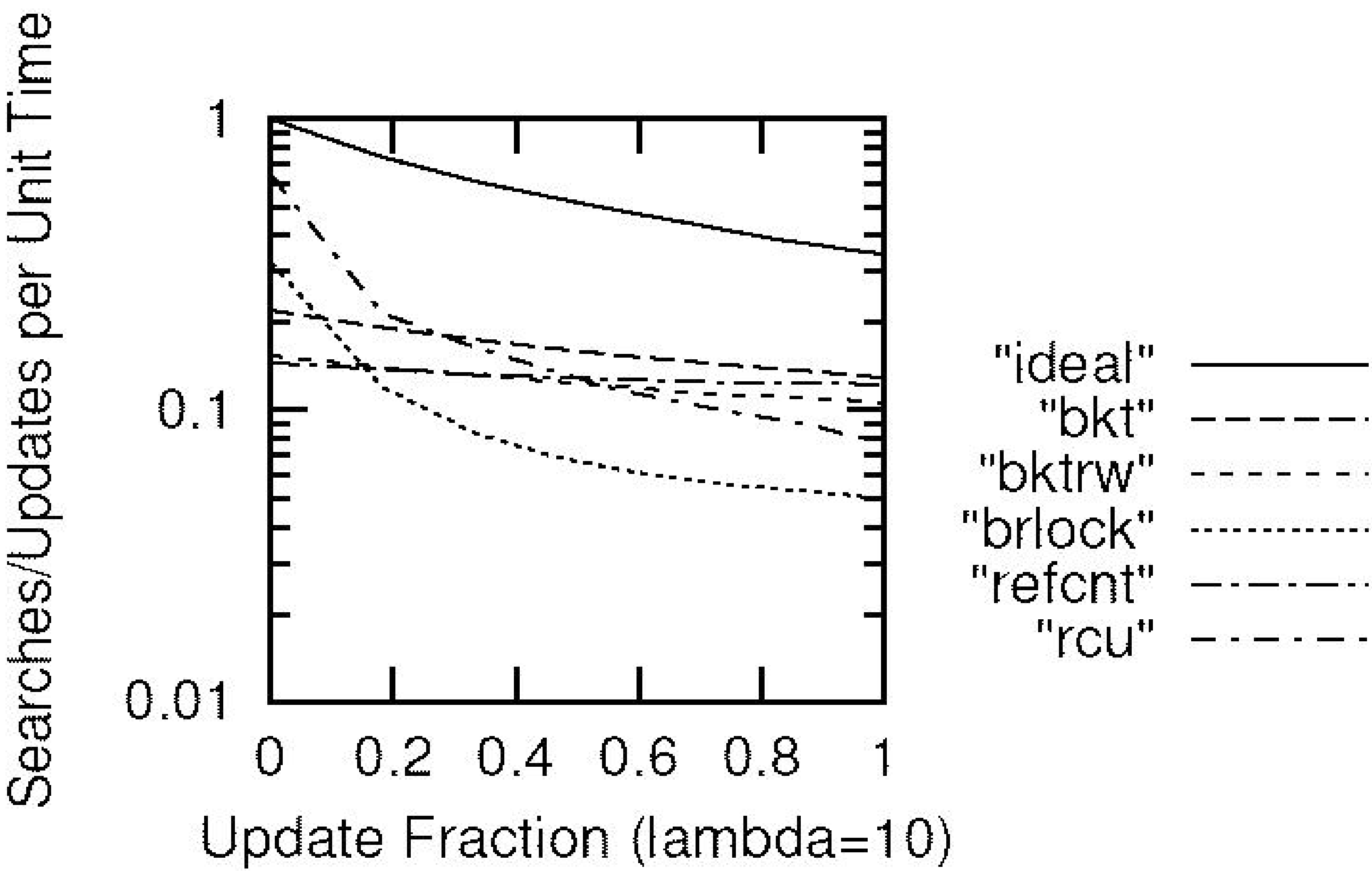
Grace Periods



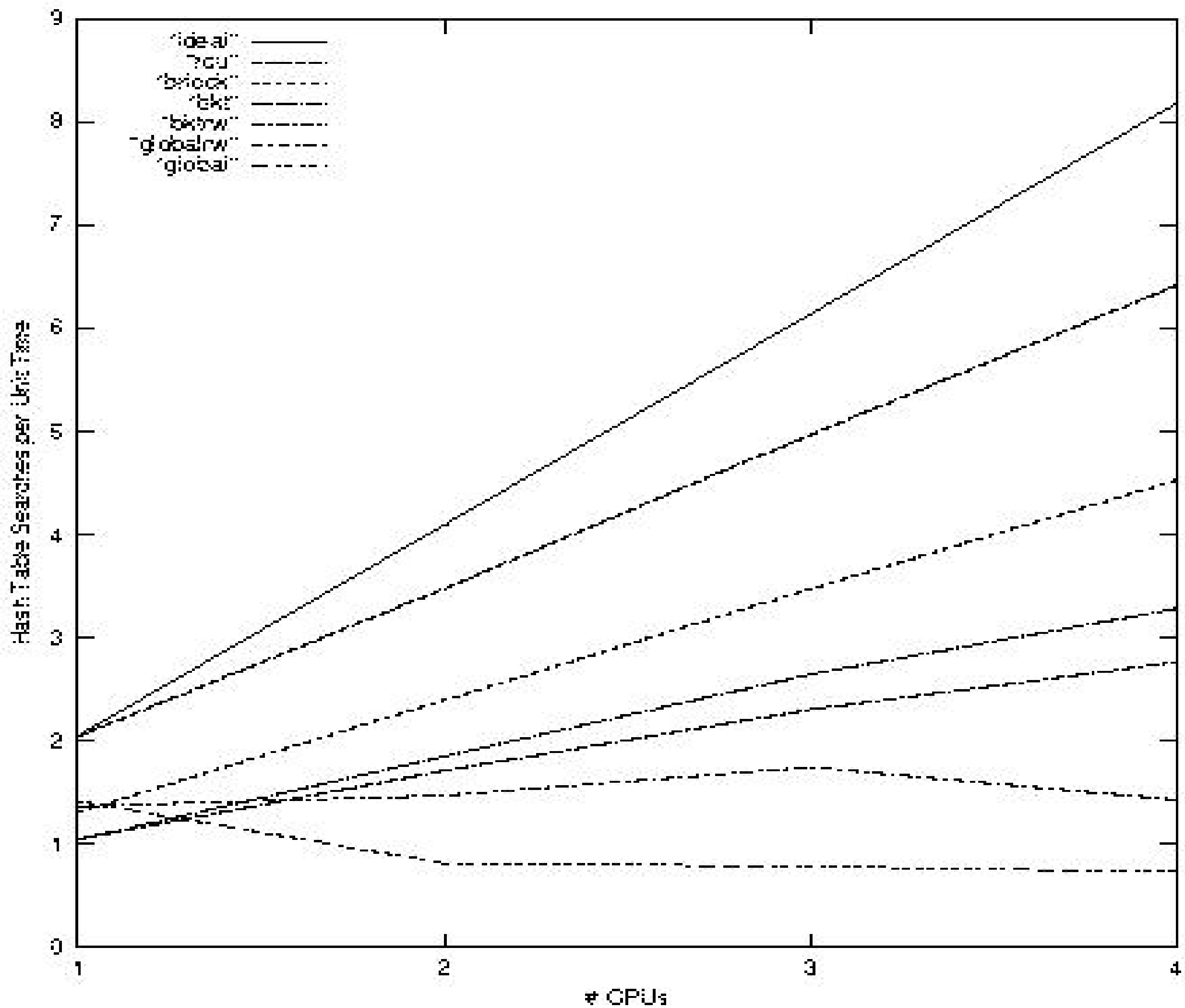
x86 Read-Only Results



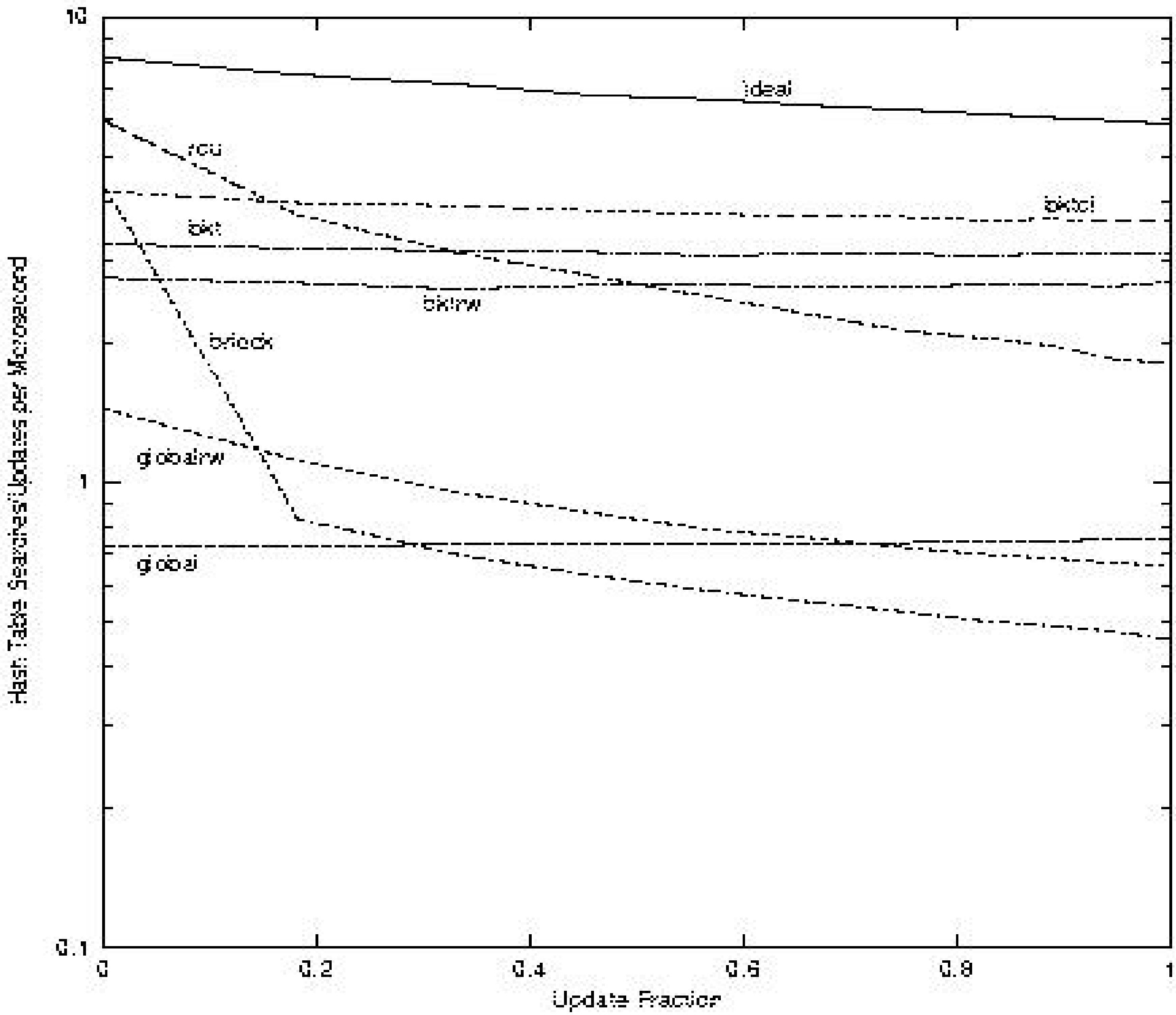
x86 Results for Mixed Workload



x86 Read-Only Results (Large)



x86 Mixed Results (Large)



Two Types of Designs For RCU

- For situations well-suited to RCU:
  - Designs that make direct use of RCU
- For algorithms that do not tolerate RCU's stale- and inconsistent-data properties:
  - Design templates that transform algorithms so as to tolerate stale and/or inconsistent data

Designs for Direct RCU Use

- Reader/Writer-Lock/RCU Analogy (5)
  - Routing tables, Linux tasklist lock patch, ...
- Pure RCU (4)
  - Dynamic interrupt handlers...
  - Linux NMI handlers...
- RCU Existence Locks (7)
  - Ensure data structure persists as needed (K42)
  - Linux SysV IPC, dcache, IP route cache, ...
- RCU Readers With WFS Writers (1)
  - K42 hash tables

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)

Reader-Writer Lock and RCU

```
int search(long key, int result)
{
    struct el *p;
    read_lock(&rw);
    list_for_each_entry(h, p, lst)
        if (p->key == key) {
            *result = p->data;
            read_unlock(&rw);
            return (1);
        }
    read_unlock(&rw);
    return (0);
}
```

```
int search(long key, int result)
{
    struct el *p;
    rcu_read_lock();
    list_for_each_entry_rcu(h, p, lst)
        if (p->key == key) {
            *result = p->data;
            rcu_read_unlock();
            return (1);
        }
    rcu_read_unlock();
    return (0);
}
```

Reader-Writer Lock and RCU

```
int delete(long key)
{
    struct el *p;
    write_lock(&rw);
    list_for_each_entry(h, p, lst)
        if (p->key == key) {
            list_del(&p->lst);
            write_unlock(&rw);
            return (1);
        }
    write_unlock(&rw);
    return (0);
}
```

```
int delete(long key)
{
    struct el *p;
    spin_lock(&lck);
    list_for_each_entry(h, p, lst)
        if (p->key == key) {
            list_del_rcu(&p->lst);
            spin_unlock(&lck);
            return (1);
        }
    spin_unlock(&lck);
    return (0);
}
```

Reader-Writer Lock and RCU

```
void insert(struct el *p)
{
        write_lock(&rw);
        list_add(p, h);
        write_unlock(&rw);
}
```

```
void insert(struct el *p)
{
        spin_lock(&lck);
        list_add_rcu(p, h);
        spin_unlock(&lck);
}
```

RCU/Reader-Writer-Lock Caveats

- Searches race with updates
  - Some algorithms tolerate such nonsense
  - Others need to be transformed – see later slides
- Updaters still can see significant contention
  - See earlier locking designs
- There is no way to block readers
  - Which is the whole point...
  - See later slides for ways to deal with this

Pure RCU

- Delay execution of update until all existing readers are done
  - See prior “big reference counter” example
  - The dynamic NMI/SMI/IPMI handlers are another example

Pure RCU: Timeouts and Interrupts

```
spin_lock_irqsave(&(to_clean->si_lock), flags);
spin_lock(&(to_clean->msg_lock));
to_clean->stop_operation = 1;
to_clean->irq_cleanup(to_clean);
spin_unlock(&(to_clean->msg_lock));
spin_unlock_irqrestore(&(to_clean->si_lock), flags);

synchronize_kernel();
while (!to_clean->timer_stopped) {
        set_current_state(TASK_UNINTERRUPTIBLE);
        schedule_timeout(1);
}
rv = ipmi_unregister_smi(to_clean->intf);
if (rv)
        printk(KERN_ERR "Can't unregister device: errno=%d\n", rv);

to_clean->handlers->cleanup(to_clean->si_sm);
kfree(to_clean->si_sm);
to_clean->io_cleanup(to_clean);
```

RCU Existence Locks

- Normal existence-guarantee schemes use global locks or per-element reference counts
  - Subject to contention and cache thrashing
  - But reference counts are OK if you need to write to the element anyway!
- RCU provides existence guarantees

```
list_del_rcu(p);
synchronize_kernel();
kfree(p);
```

Designs for Direct RCU Use

- Reader/Writer-Lock/RCU Analogy (5)
- Pure RCU (4)
- RCU Existence Locks (7)
- RCU Readers With WFS Writers (1)
  - Only one use thus far, ask me again later!
- But what about algorithms that don't like stale data???

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 Templates

- 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

- RCU uses atomic updates of single value
  - Most CPUs support this
- If multiple updates must appear atomic:
  - 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
- Example in next section

Impose Level of Indirection

- Difficult to ensure consistent view of multiple independent data elements
  - Requires lots and lots of memory barriers
- Solution: place the independent data elements in one structure referenced by a pointer
- Then can atomically switch the pointer
  - And get rid of most of the memory barriers!!!
- Example in next section

Mark Obsolete Object

- RCU search structure w/data-locked items

```
rcu_read_lock();
p = search(key);
if (p != NULL)
	spin_lock(&p->mutex);
rcu_read_unlock();
```

- Place a “deleted” flag in each element

```
rcu_read_lock();
p = search(key);
if (p != NULL) {
	spin_lock(&p->mutex);
	if (p->deleted) {
		spin_unlock(&p->mutex);
		p = NULL;
	}
}
rcu_read_unlock();
return (p);
```

Ordered Update with Ordered Read

- Expanding array

```
/ * update */
new_array = kmalloc(new_size * sizeof(*newarray));
copy_and_init(new_array, array);
smp_wmb();
array = new_array;
smp_wmb();
size = new_size;

/ * read */
if (i >= size)
	return - ENOENT;
smp_rmb();
p = array;
smp_read_barrier_depends();
return p[i];
```

- Usually better to impose level of indirection...

Global Version Number

- In Linux, combine seqlock_t with RCU
- For example, in dcache lookup:

```
do {
        seq = read_seqbegin(&rename_lock);
        dentry = __d_lookup(parent, name);
        if (dentry)
                break;
} while (read_seqretry(&rename_lock, seq));
```

- RCU protects against cache prune and “rm”
- seqlock_t protects against “mv”
- Could also place sequence number in dentry to allow “mass invalidate” of dentries

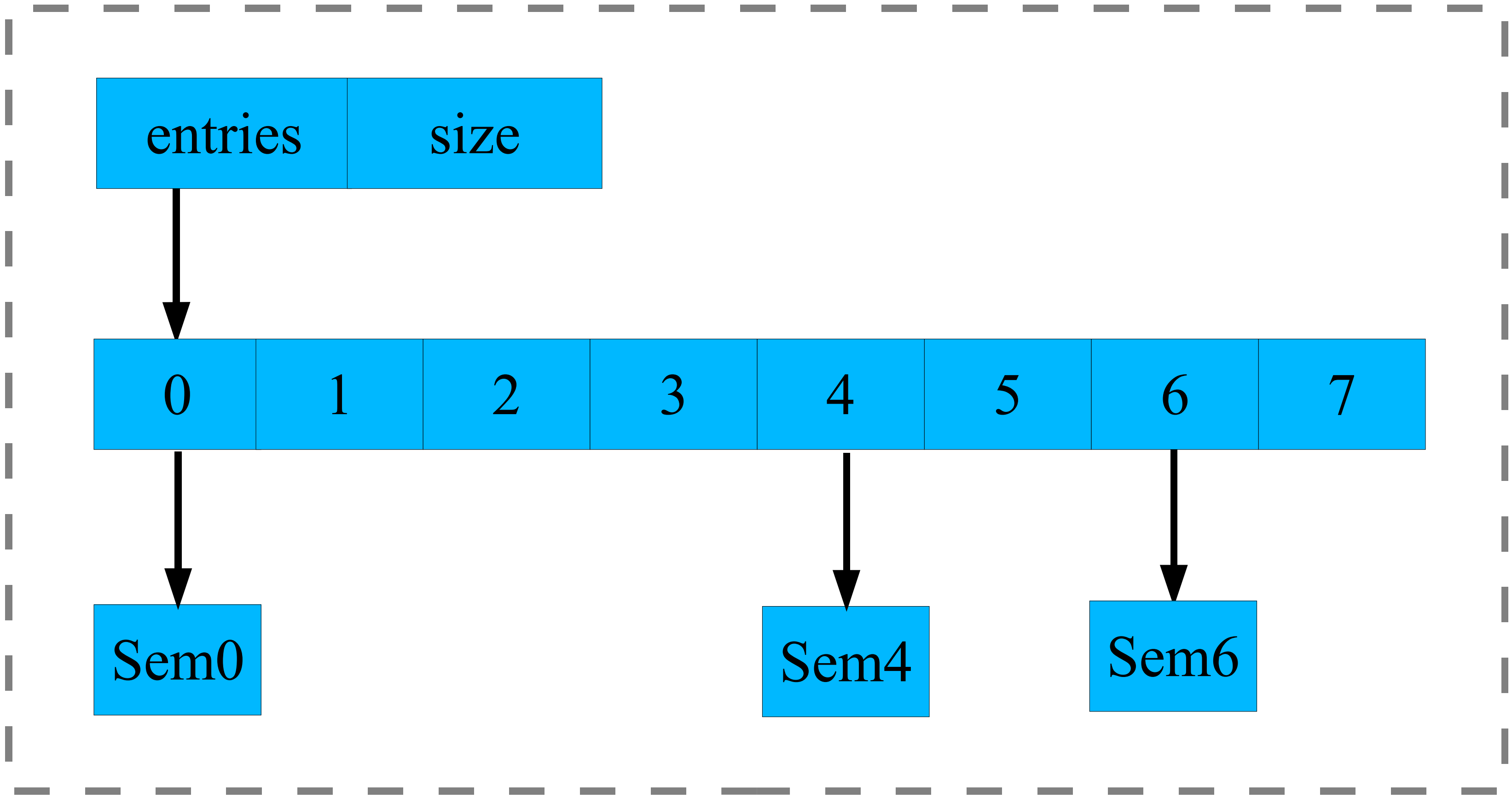
RCU Transformational Patterns

- 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)

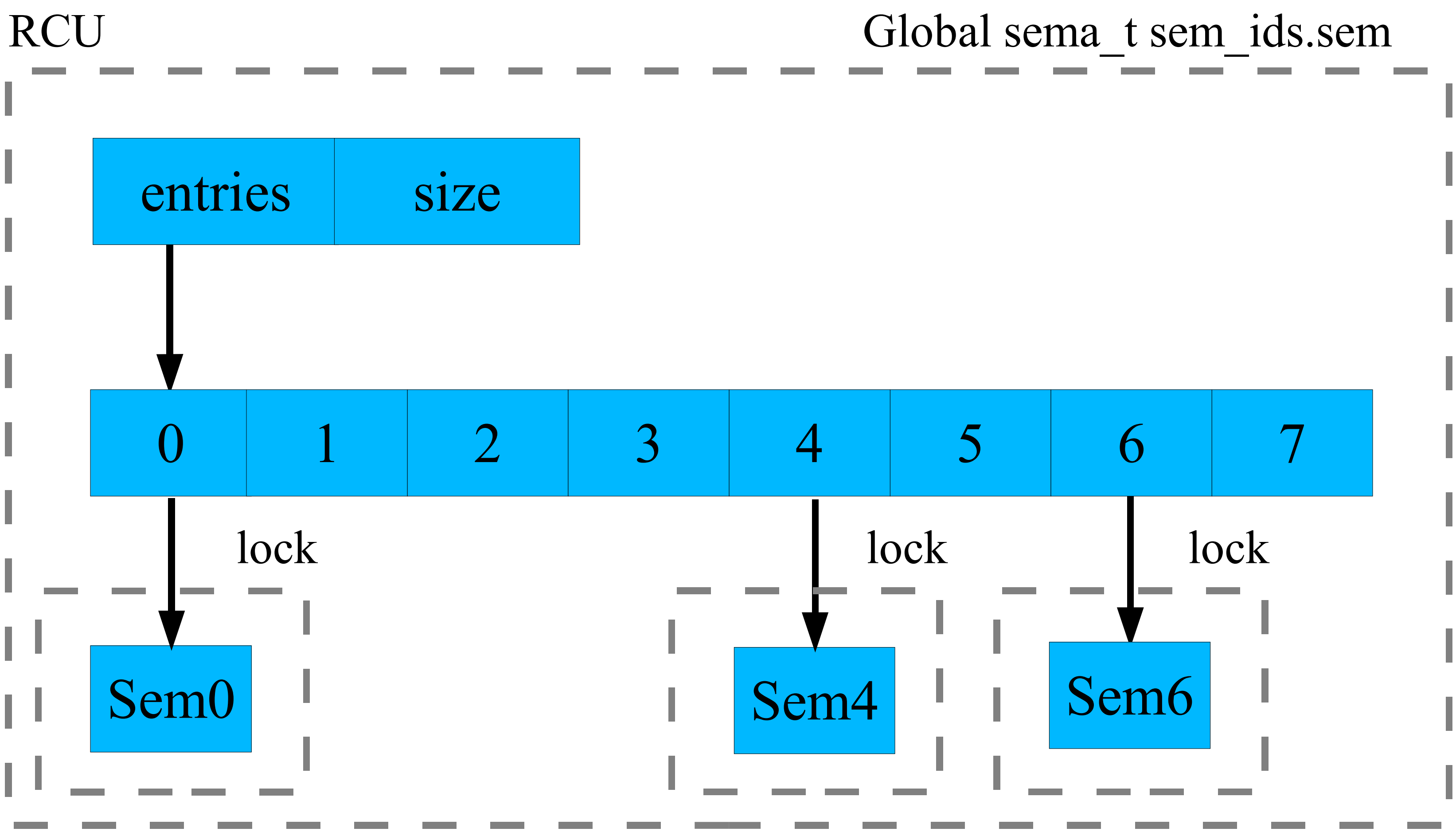
Putting It All Together

2.4 System V Semaphore Locking

Global sema_t sem_ids.sem
Global spinlock_t sem_ids.ary

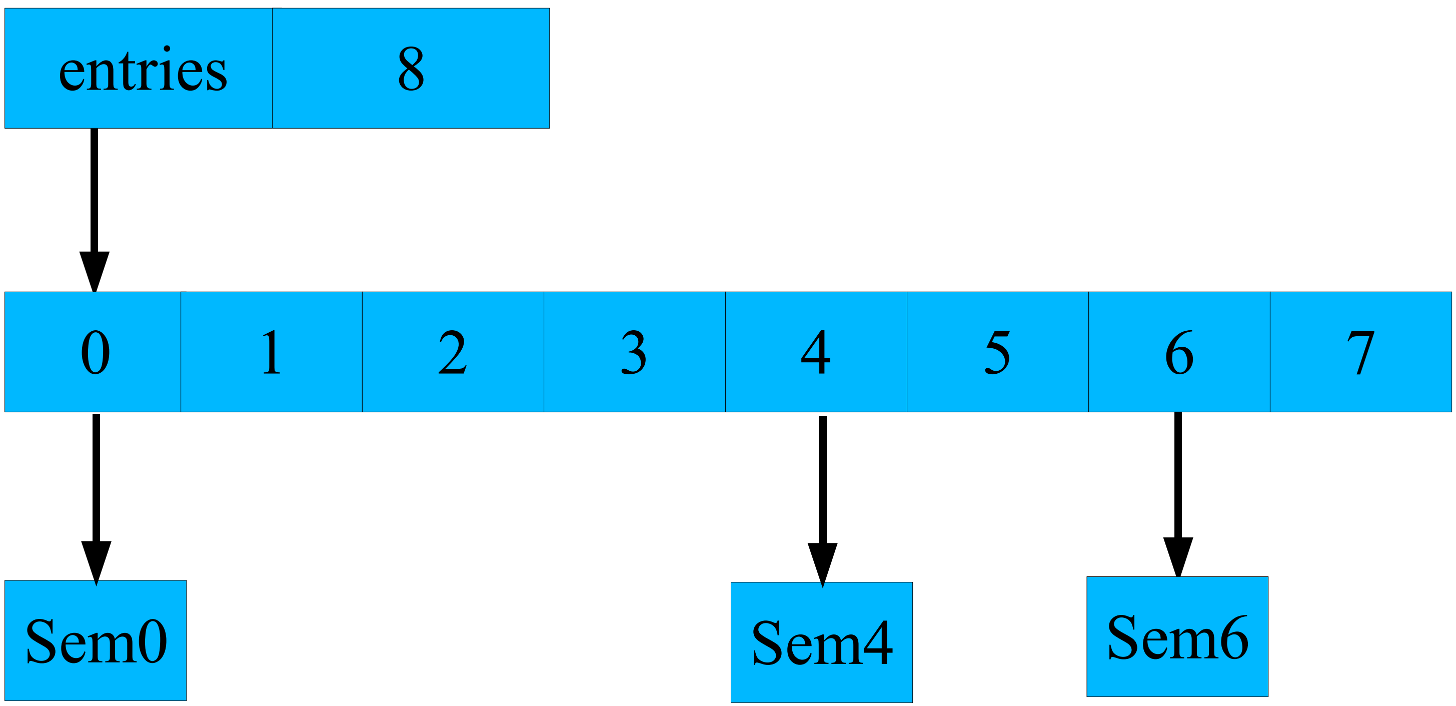


2.6 System V Semaphore Locking

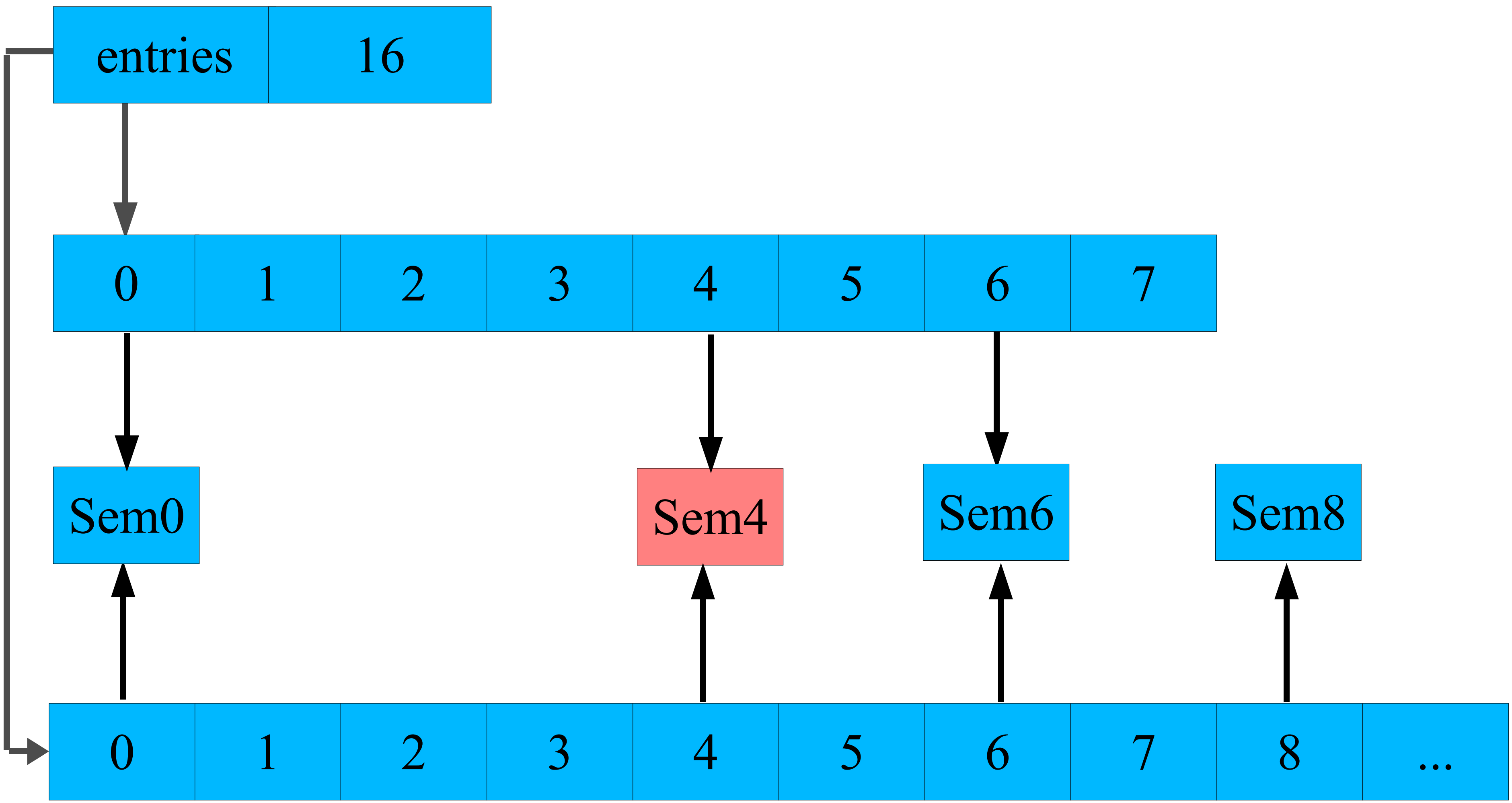


Each semaphore has a “deleted” flag to force search failure

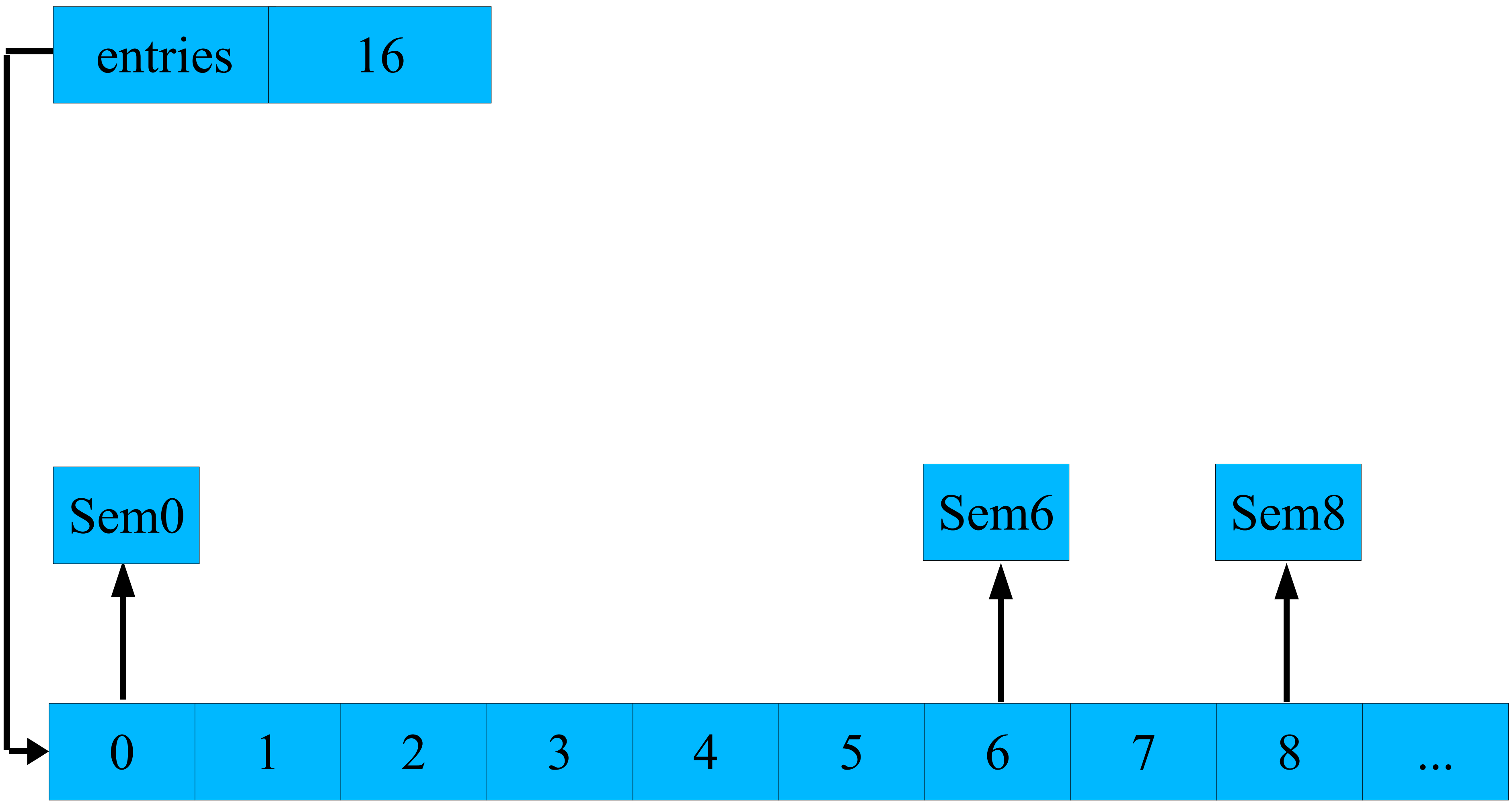
2.6 SysV Sema Animation (1)



2.6 SysV Sema Animation (2)



2.6 SysV Sema Animation (3)



Searching for Semaphore

```
rcu_read_lock();
if(lid >= ids->size) {
        rcu_read_unlock();
        return NULL;
}
smp_rmb(); /* prevent indexing old array with new size */
entries = ids->entries;
read_barrier_depends(); /*prevent seeing new array unitialized */
out = entries[lid].p;
if(out == NULL) {
        rcu_read_unlock();
        return NULL;
}
spin_lock(&out->lock);
if (out->deleted) {
        spin_unlock(&out->lock);
        rcu_read_unlock();
        return NULL;
}
return out;
```

Expanding Semaphore Array

```
old = ids->entries;
i = ids->size;

smp_wmb();      /* prevent seeing new array uninitialized. */
ids->entries = new;
smp_wmb();      /* prevent indexing into old array based on new size. */
ids->size = newsize;

ipc_rcu_free(old, sizeof(struct ipc_id)*i);
return ids->size;
```

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

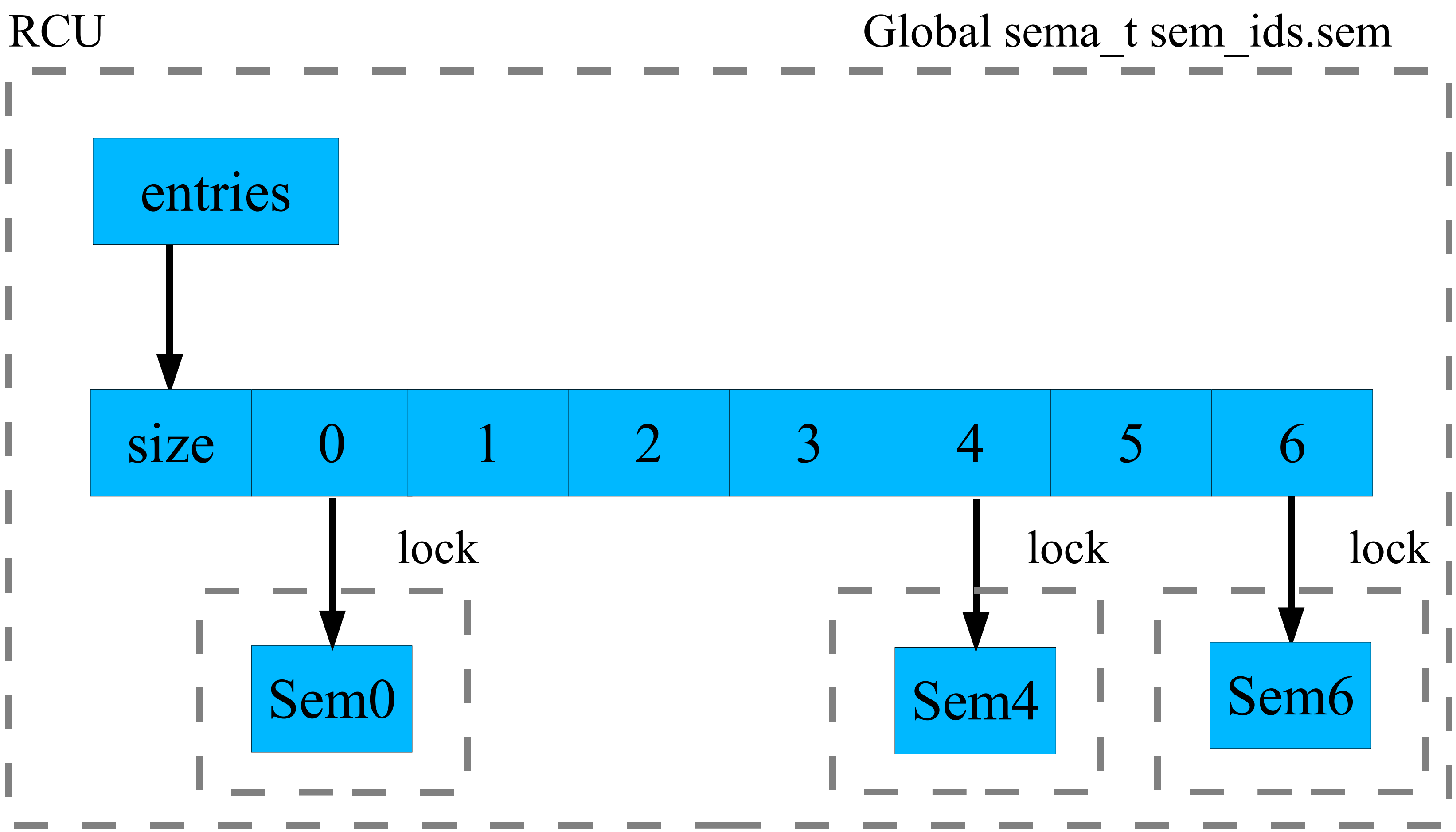
Numbers are test duration, smaller is better.

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.

Proposed Locking

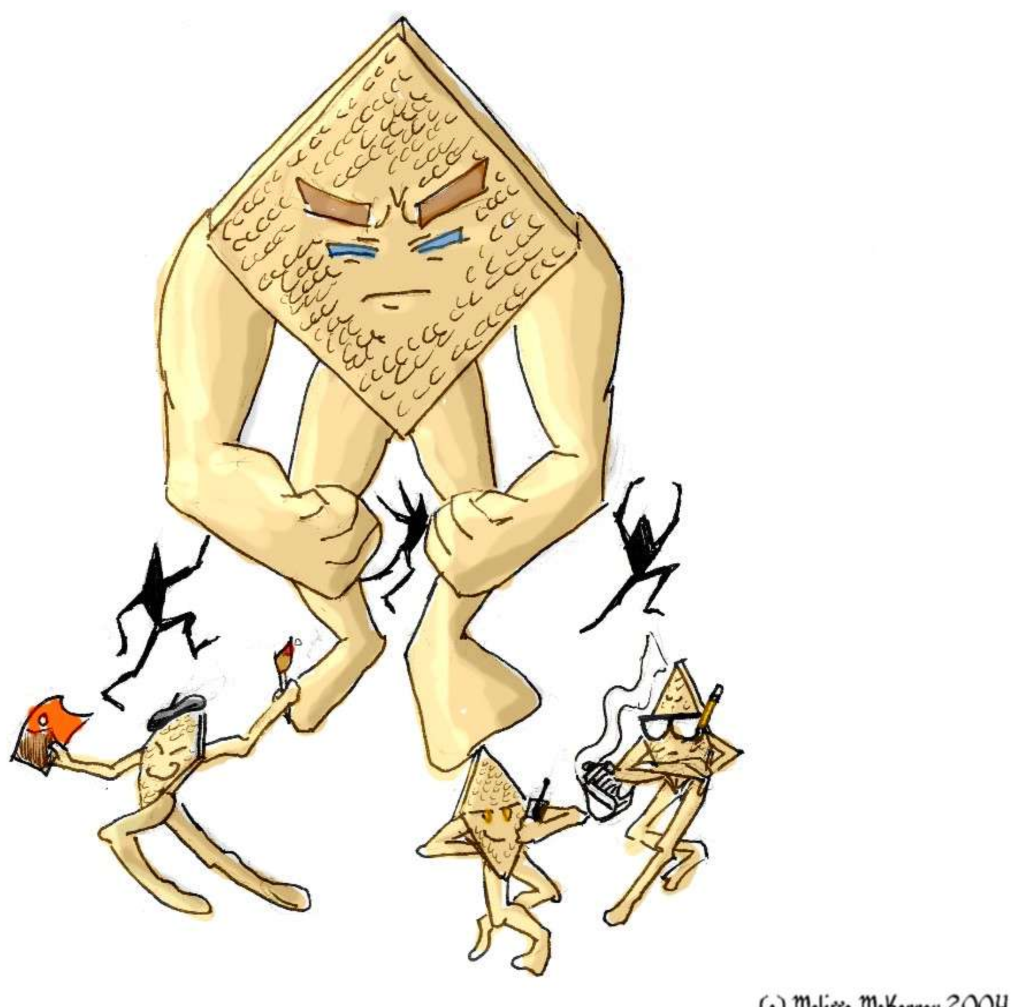


Each semaphore has a "deleted" flag to force search failure

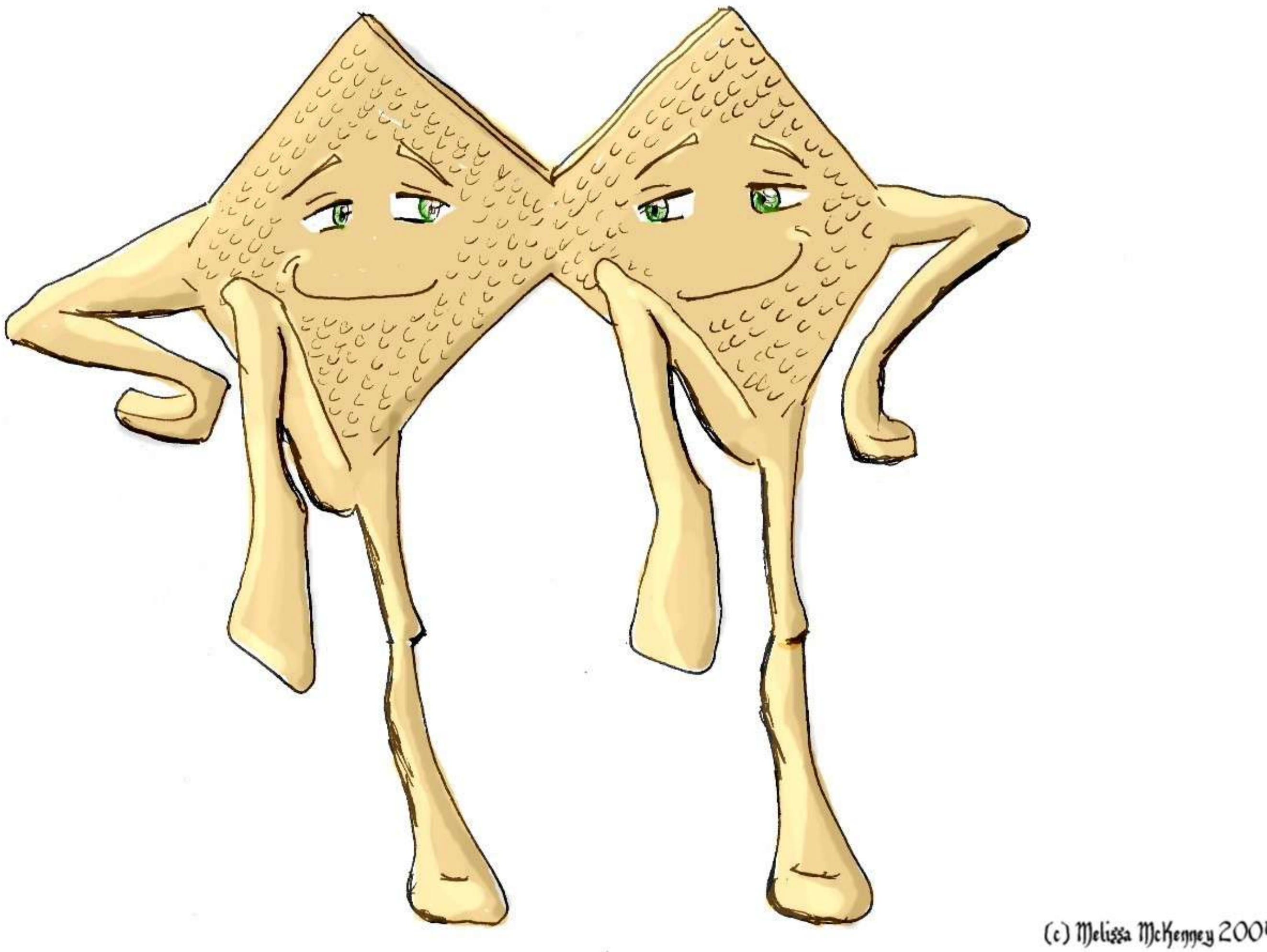
The Road Ahead

Uniprocessor Űber Alles

Uniprocessor With Friends

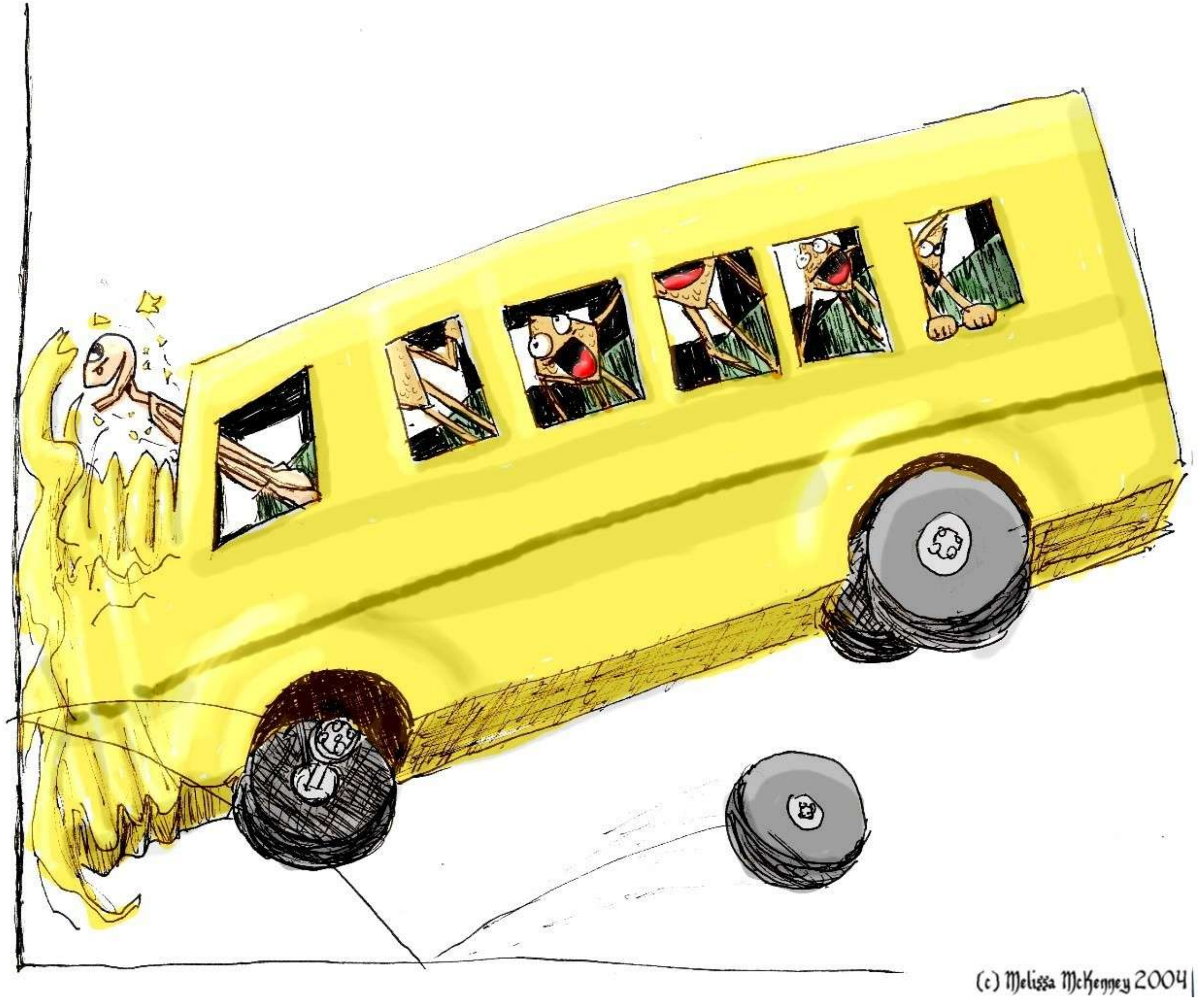


Multithreaded Mania

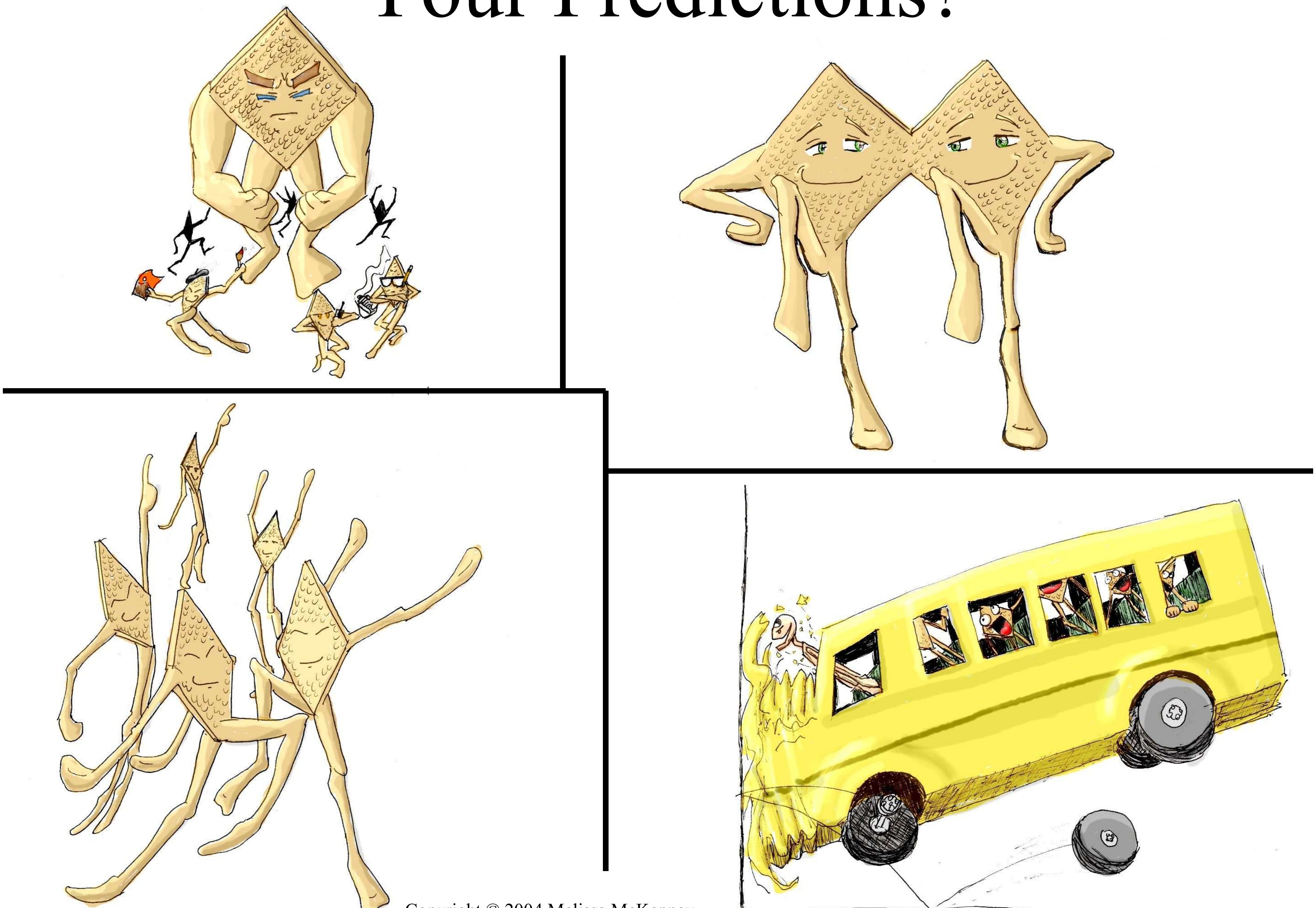


More of the Same

Crash Dummies Slamming into the Memory Wall



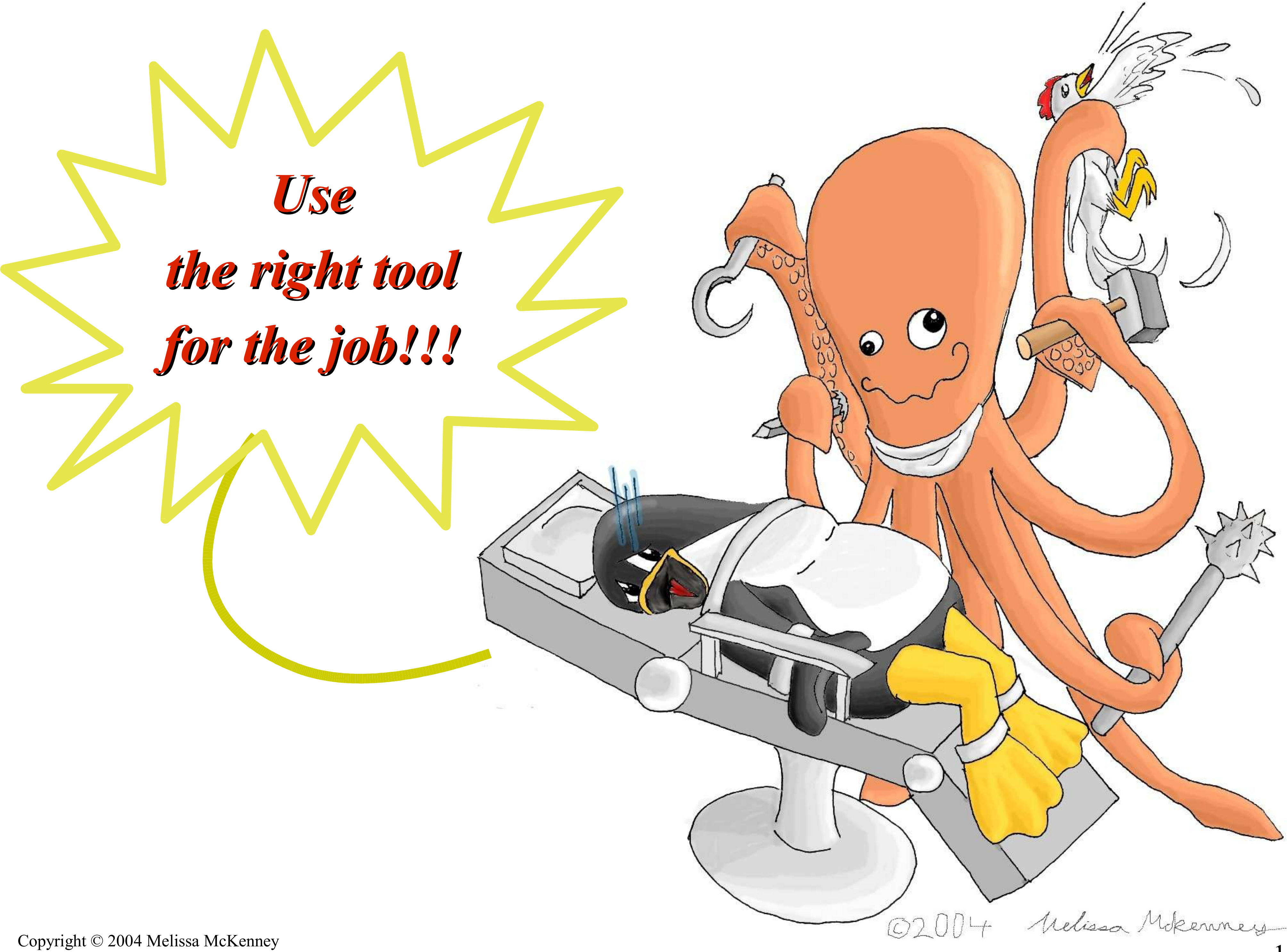
Your Predictions?



My Guess...

Somewhere between Multithreaded Mania and More of the Same, with both hardware threading and multicore dies.

Summary and Conclusions


Use
the right tool
for the job!!!
©2004 Melissa McKenney

Legal

- 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.
- Other company, product, and service names may be trademarks or service marks of others.

BACKUP