Read-Copy Update

An Update

Paul E. McKenney, *IBM Beaverton*
Dipankar Sarma, *IBM India Software Lab*
Andrea Arcangeli, *SuSE*
Andi Kleen, *SuSE*
Orran Krieger, *IBM T.J. Watson*
Rusty Russell, *IBM OzLabs*

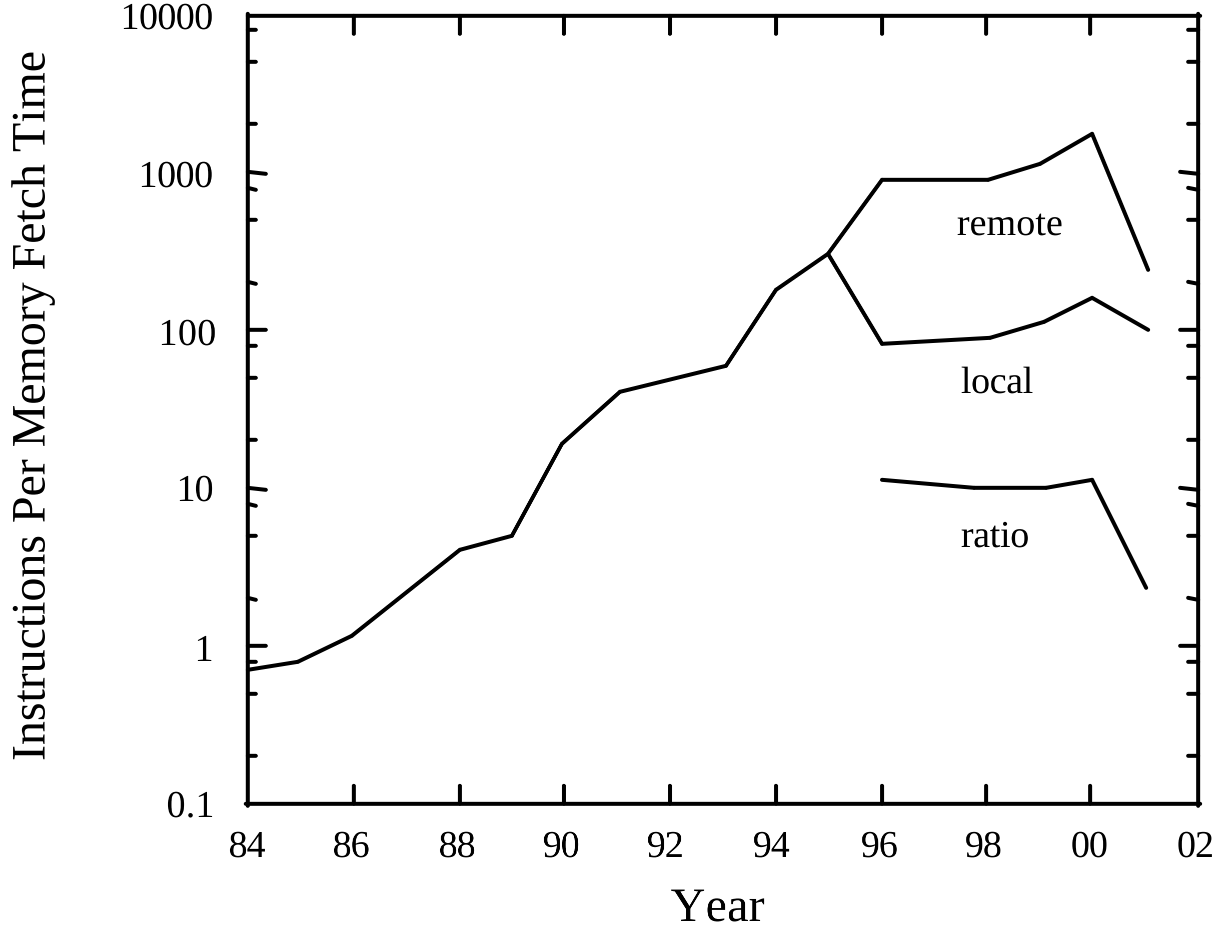
Overview

- *Background on read-copy update (RCU)*
  - *Motivation and concepts*
  - *Implementation outline and API*
  - *Example use (IP route cache)*
- Linux® implementations of RCU
- Experience with Linux RCU implementations
- Details of rcu-poll implementation
- Concluding remarks

Goal

- Simple, high-performance and -scaling algorithms for read-mostly situations
  - Reads must be *fast…*
    - Readers must *not* be required to acquire locks, execute atomic operations, or disable interrupts
    - Read-side code same as UP user-level implementation
    - Want to scale with CPU core clock, *not* with memory latency
  - OK if writers have to do a *little* more work
    - But writes can be faster too, especially if RCU is heavily used or in cases of high read-side lock contention.
- Handle preemptive Linux 2.5 kernel

Long-Term Architectural Trends



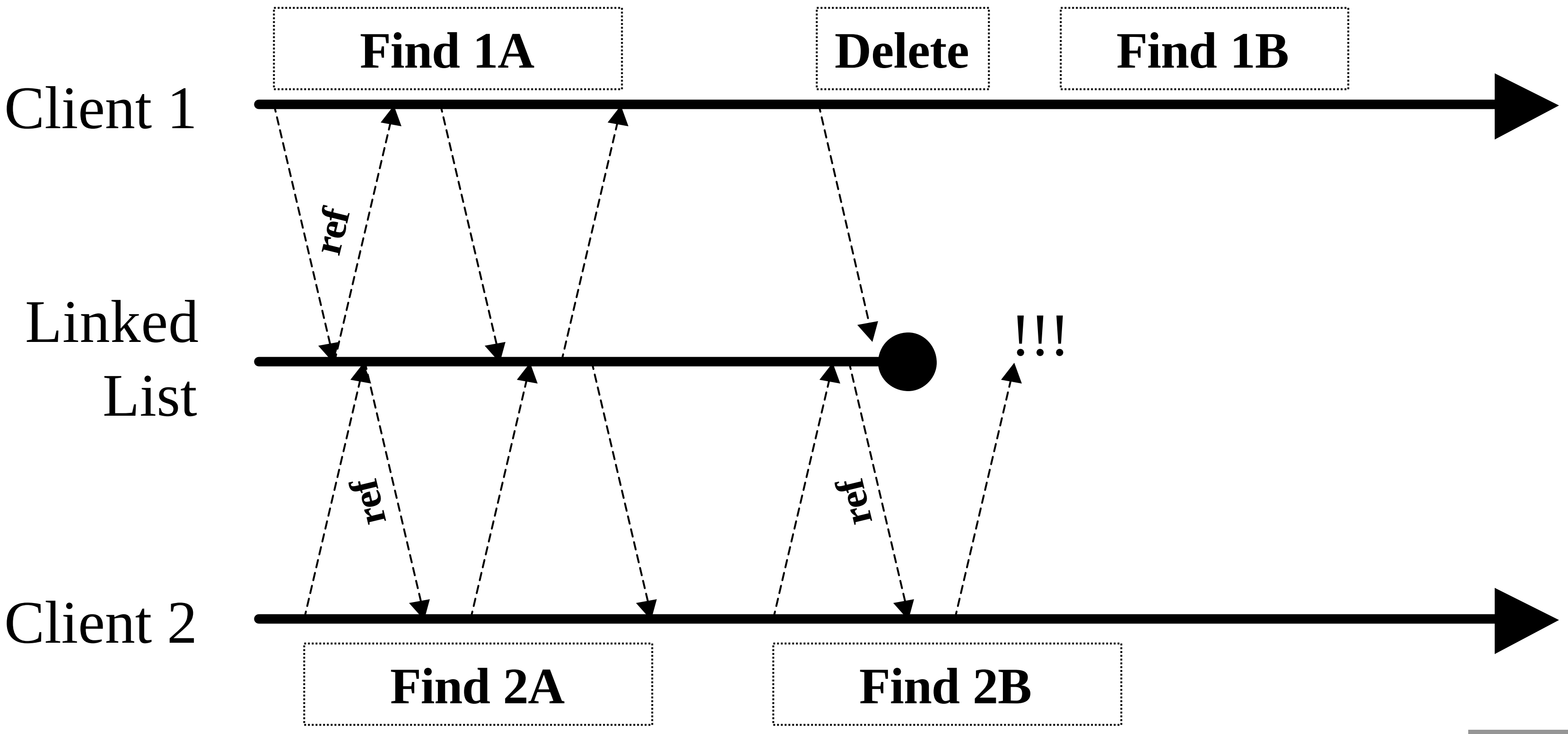
Data from Sequent®/IBM® NUMA-Q® Machines

Architectural-Trend Consequences

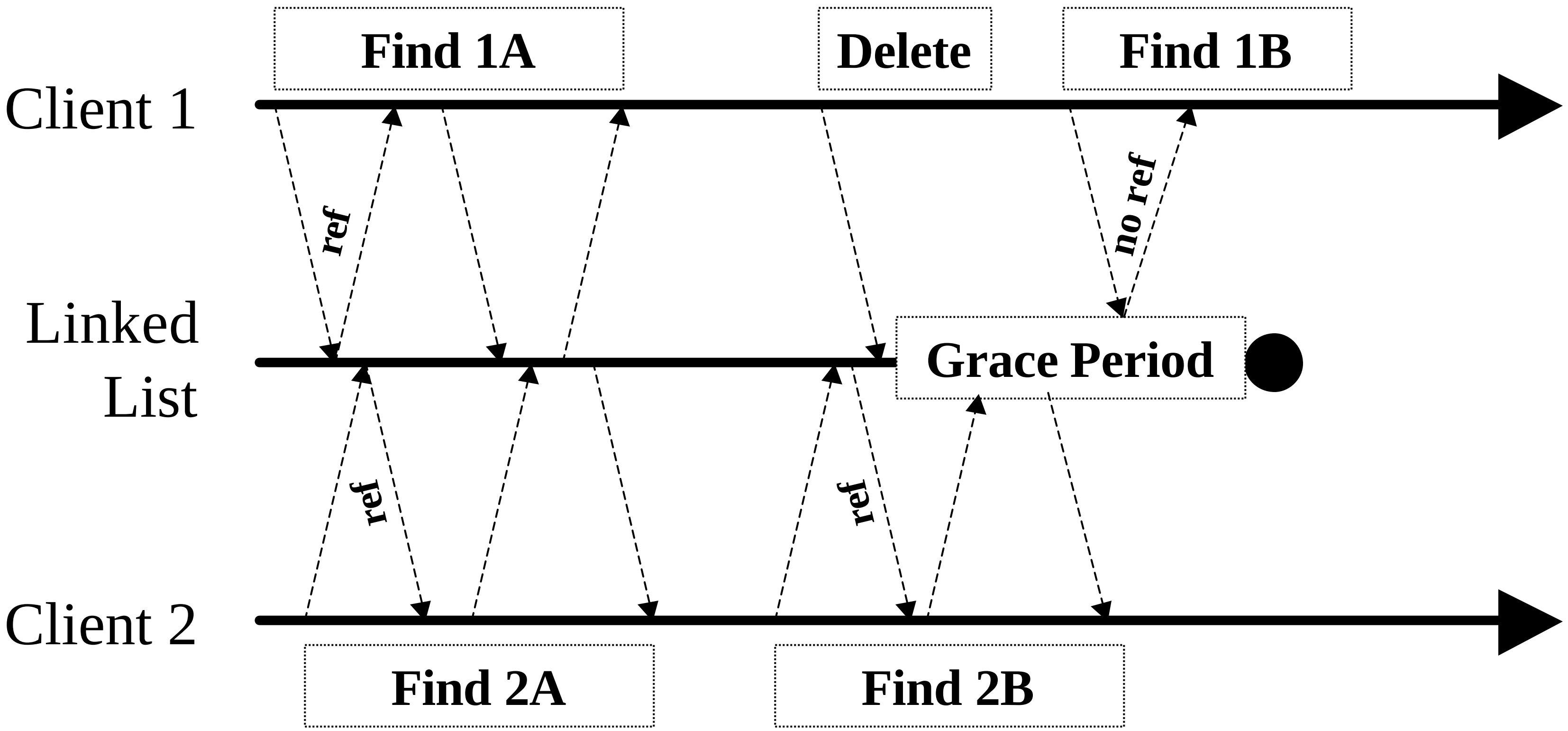
- Global locks becoming increasingly expensive
- Globally-used reference counters also becoming increasingly expensive
- *Would like some way for read-only accesses to read-mostly data structures to avoid locks and reference-count manipulation...*
  - *Take advantage of event-driven nature of the Linux kernel*

Example Problem

- Race between use and deletion of list element



Read-Copy Update: Grace Period



Code to Detect Grace Period

```
void synchronize_kernel()
{
   for (i = 0; i < smp_num_cpus; i++) {
      run_on(i);
   }
}
```

- This simple implementation has some shortcomings:
  - Blocks caller, so can't be invoked from interrupt, with spinlock held, or with interrupts disabled
  - Does not work in preemptible kernel
  - *Slooooow*:
    - Cannot "batch" requests for grace periods
    - Multiple context switches per grace period: high overhead
    - Can be stalled indefinitely by real-time tasks (see paper)
    - Can throttle update rate

Better Grace-Period Detection

- Update code registers a callback
- Callbacks queued onto list
- Daemon/timer/tasklet/whatever periodically:
  - Checks for end of grace period:
    - invoking all callbacks waiting for the end of that grace period
    - starting a new grace period if new callbacks have arrived
- Allows batching, use from all execution contexts, and is *much* faster

Read-Copy Update API

- synchronize_kernel(): Wait for a grace period.
- call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg):
  – Invoke func(arg) at the end of a grace period, but read-side code must not allow preemption.
- call_rcu_preempt(struct rcu_head *head, void (*func)(void *arg), void *arg):
  – Invoke func(arg) at the end of a grace period, read-side code may safely be preempted.
- list_add_rcu(), list_add_tail_rcu(), list_for_each_rcu(), list_for_each_safe_rcu()
- read_barrier_depends() and/or write_barrier_depends()
- preempt_disable() and preempt_enable(): encapsulate as rcu_read_lock() and rcu_read_unlock()

Known Uses

- DYNIX/ptx® (since 1993)
- Tornado/K42 (pervasive)
- Patches to Linux:
  - Module unloading
  - File descriptor management
  - Hotplug CPU support
  - dentry lookup patch
  - IP route-cache lookup

IP Route Cache Search (1/2)

ip_route_output_key() in net/ipv4/route.c

```
@@ -1988,8 +1989,9 @@
        hash = rt_hash_code(key->dst,
                            key->src ^ (key->oif << 5),
                            key->tos);
-       read_lock_bh(&rt_hash_table[hash].lock);
+       rcu_read_lock();
        for (rth = rt_hash_table[hash].chain; rth;
             rth = rth->u.rt_next) {
+               read_barrier_depends(); /* list macros */
                if (rth->key.dst == key->dst &&
                    rth->key.src == key->src &&
                    rth->key.iif == 0 &&
```

IP Route Cache Search (2/2)

ip_route_output_key() in net/ipv4/route.c

```
@@ -2003,12 +2005,10 @@
            dst_hold(&rth->u.dst);
            rth->u.dst.__use++;
            rt_cache_stat[smp_processor_id()].out_hit++;
-           read_unlock_bh(&rt_hash_table[hash].lock);
+           rcu_read_unlock();
            *rp = rth;
            return 0;
        }
    }
-   read_unlock_bh(&rt_hash_table[hash].lock);
+   rcu_read_unlock();
    return ip_route_output_slow(rp, key);
 }
```

IP Route Cache Update

rt_free() in net/ipv4/route.c

```
static __inline__ void rt_free(struct rtable *rt) {
-       dst_free(&rt->u.dst);
+       call_rcu(&rt->u.dst.rcu_head,
+                (void (*)(void *))dst_free,
+                &rt->u.dst);
 }
```

struct rtable & struct dst

include/net/route.h

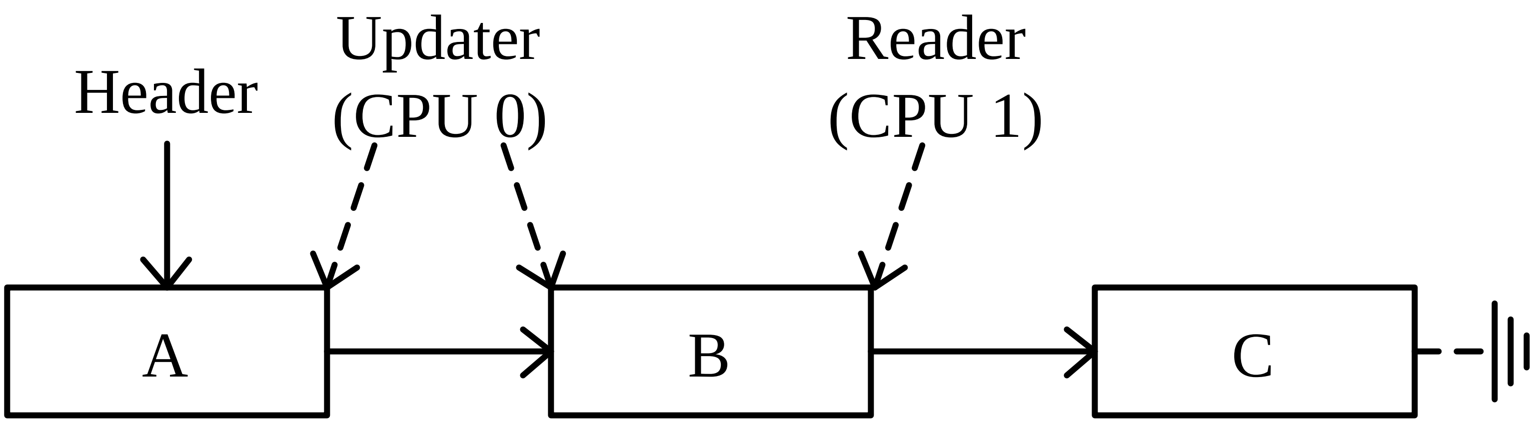
```
struct rtable{
        union       {
                struct dst_entry        dst;
                struct rtable           *rt_next;
        } u;
        unsigned                        rt_flags;
        unsigned                        rt_type;

. . .

#ifdef CONFIG_IP_ROUTE_NAT
        __u32                           rt_src_map;
        __u32                           rt_dst_map;
#endif
};
```

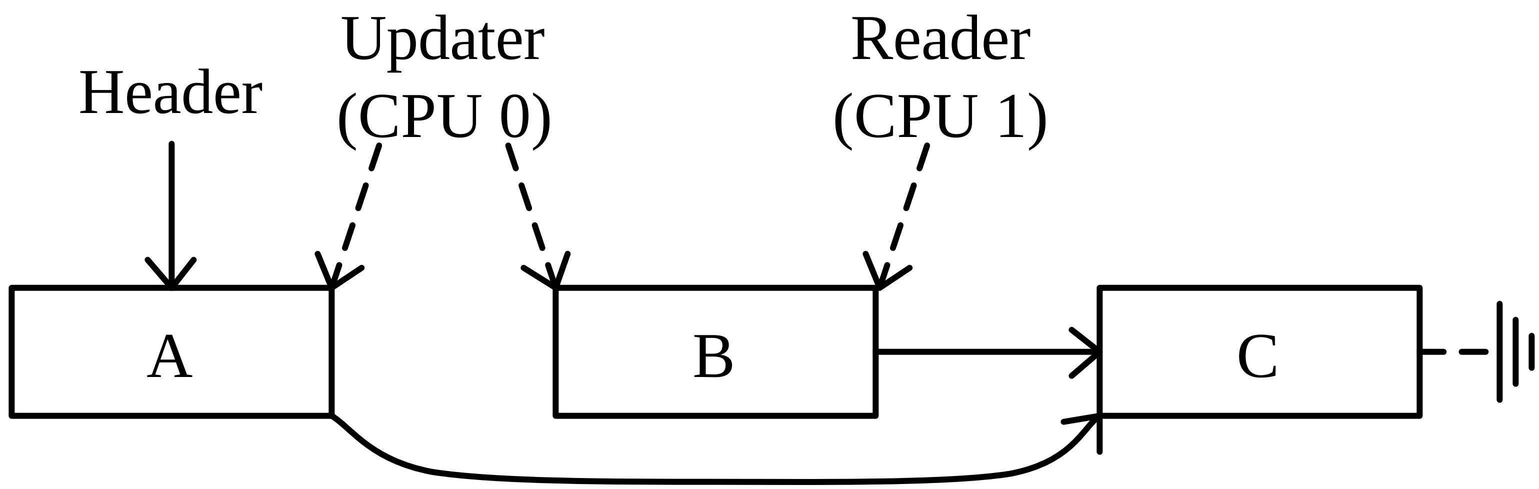
Read-Copy Update Animation

```
*rthp = rth->u.rt_next;
```



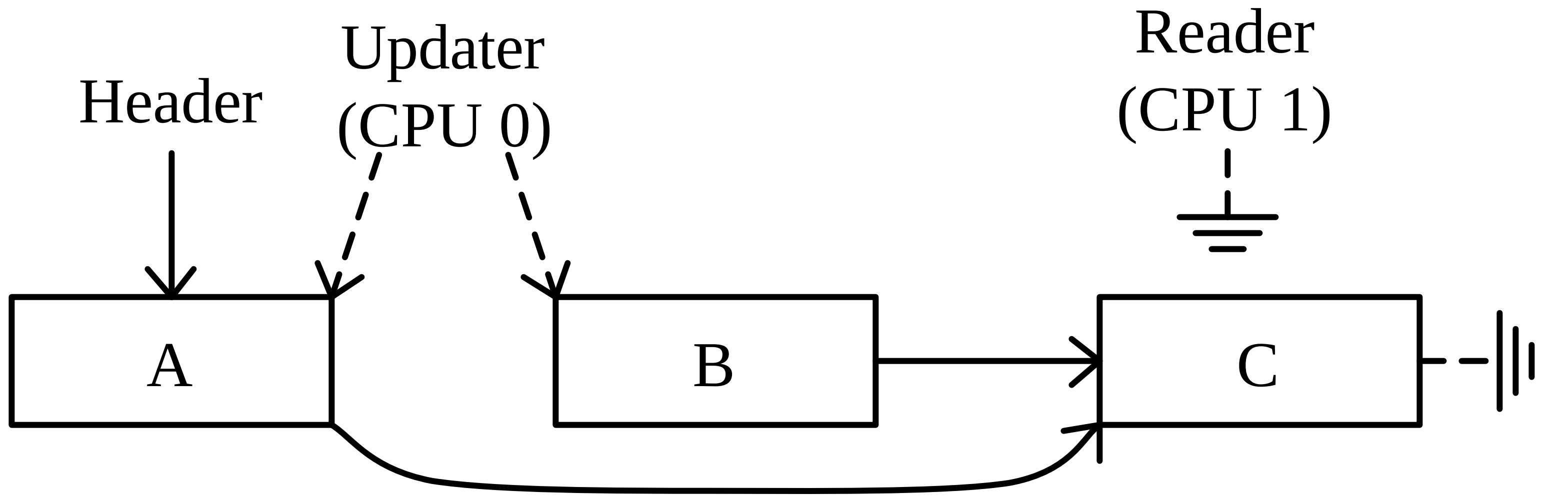
Read-Copy Update Animation

```
call_rcu(&rt->u.dst.rcu_head,
         (void (*)(void *))dst_free,
         &rt->u.dst);
```

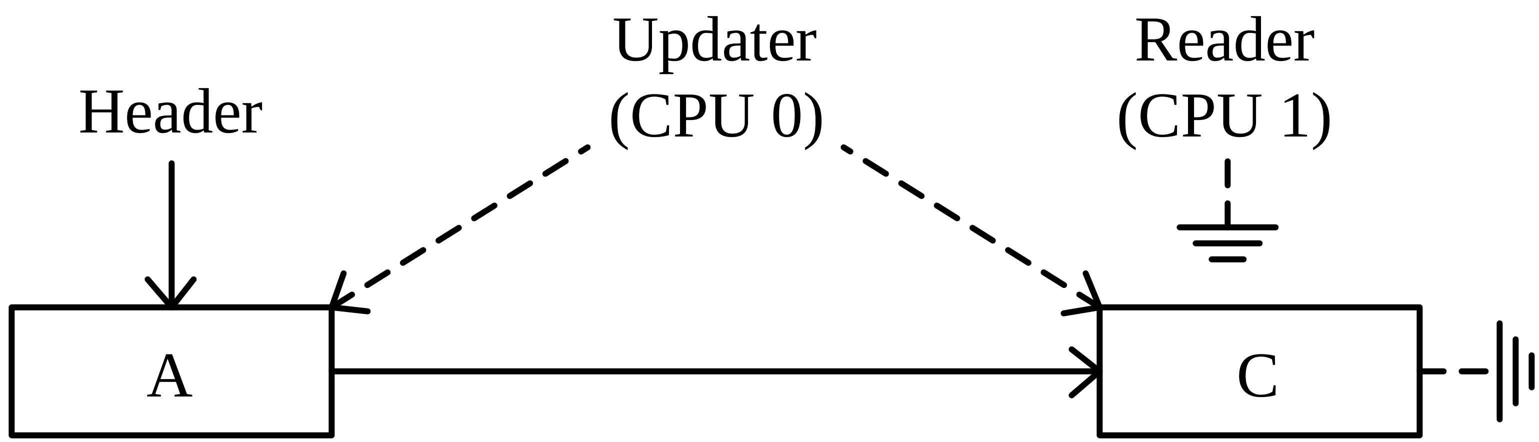


Read-Copy Update Animation

```
. . .
kmem_cache_free(dst->ops->kmem_cachep, dst);
. . .
while ((rth = *rthp) != NULL) {
```



Read-Copy Update Animation



Overview

- Background on read-copy update (RCU)
- *Linux implementations of RCU*
- Experience with Linux RCU implementations
- Details of rcu-poll implementation
- Concluding remarks

Linux RCU Implementations

- Non-preemptible
  - X-rcu (ptx-derived)
  - rcu (ptx-derived)
  - rcu-ltimer (ptx-derived)
  - rcu-poll (Sarma & Arcangeli: low latency)
  - rcu-taskq (Sarma: low complexity)
  - rcu-sched (Russell: lock-free implementation)
- *Focus on rcu-poll and a preemptible version of it*

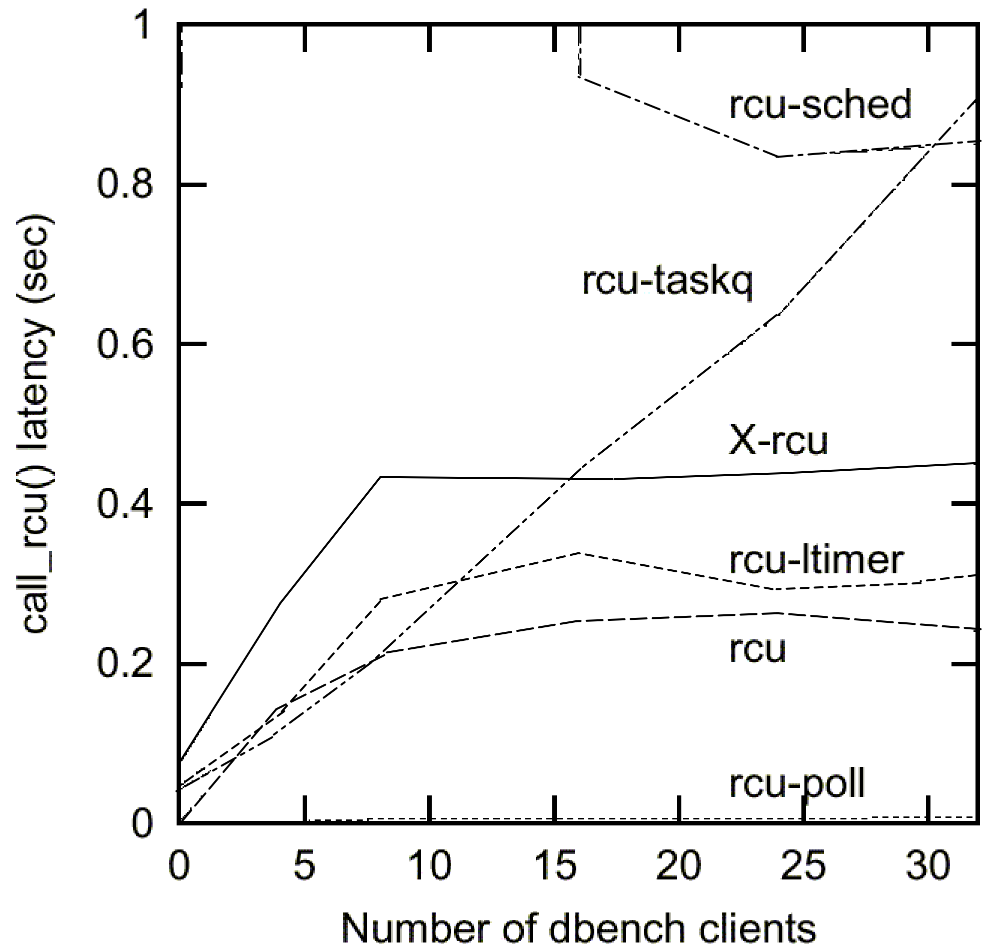
Linux RCU Implementations

- Preemptible
  - rcu_preempt (ptx-derived)
  - rcu_poll_preempt (Sarma & Arcangeli: low latency)
  - Others TBD
- *Again, focus on rcu_poll() and rcu_poll_preempt()*

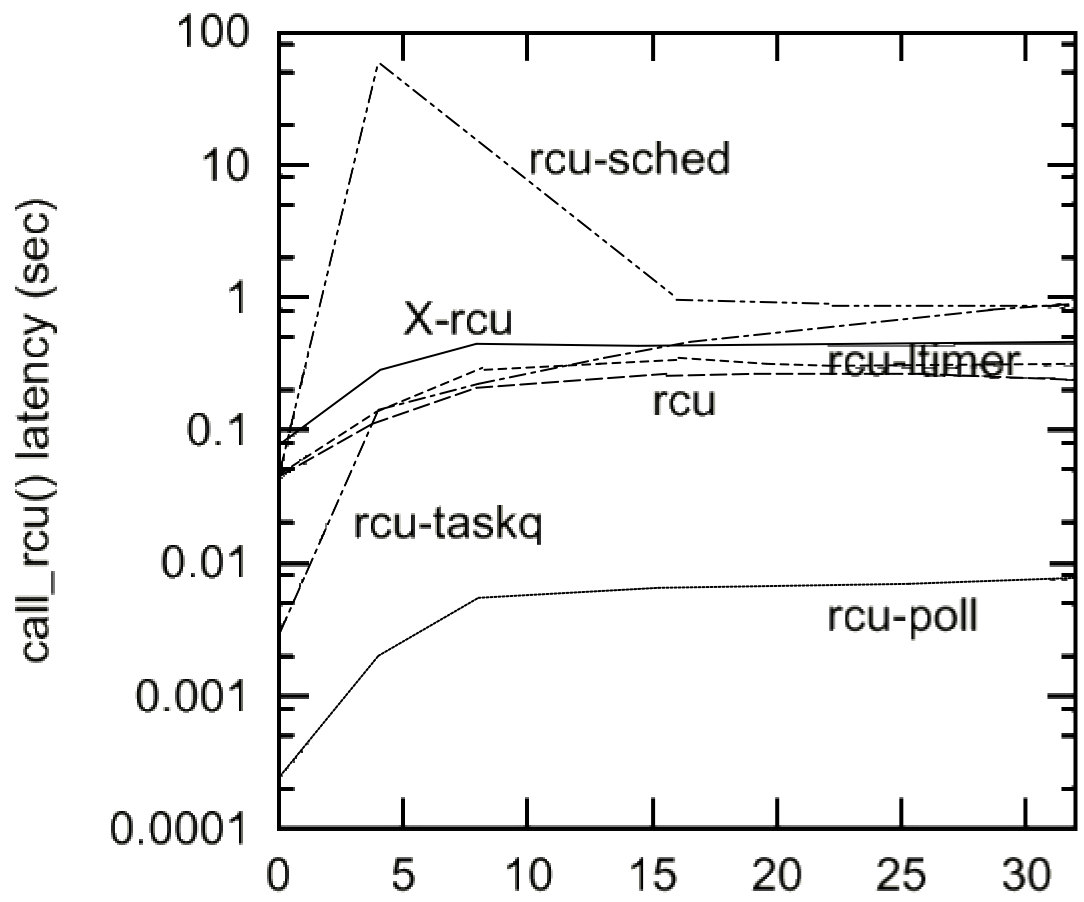
Overview

- Background on read-copy update (RCU)
- Linux implementations of RCU
- *Experience with Linux RCU implementations*
  - *Grace-period latency*
  - *Overhead reduction*
  - *Complexity*
- Details of rcu-poll implementation
- Concluding remarks

RCU Latencies

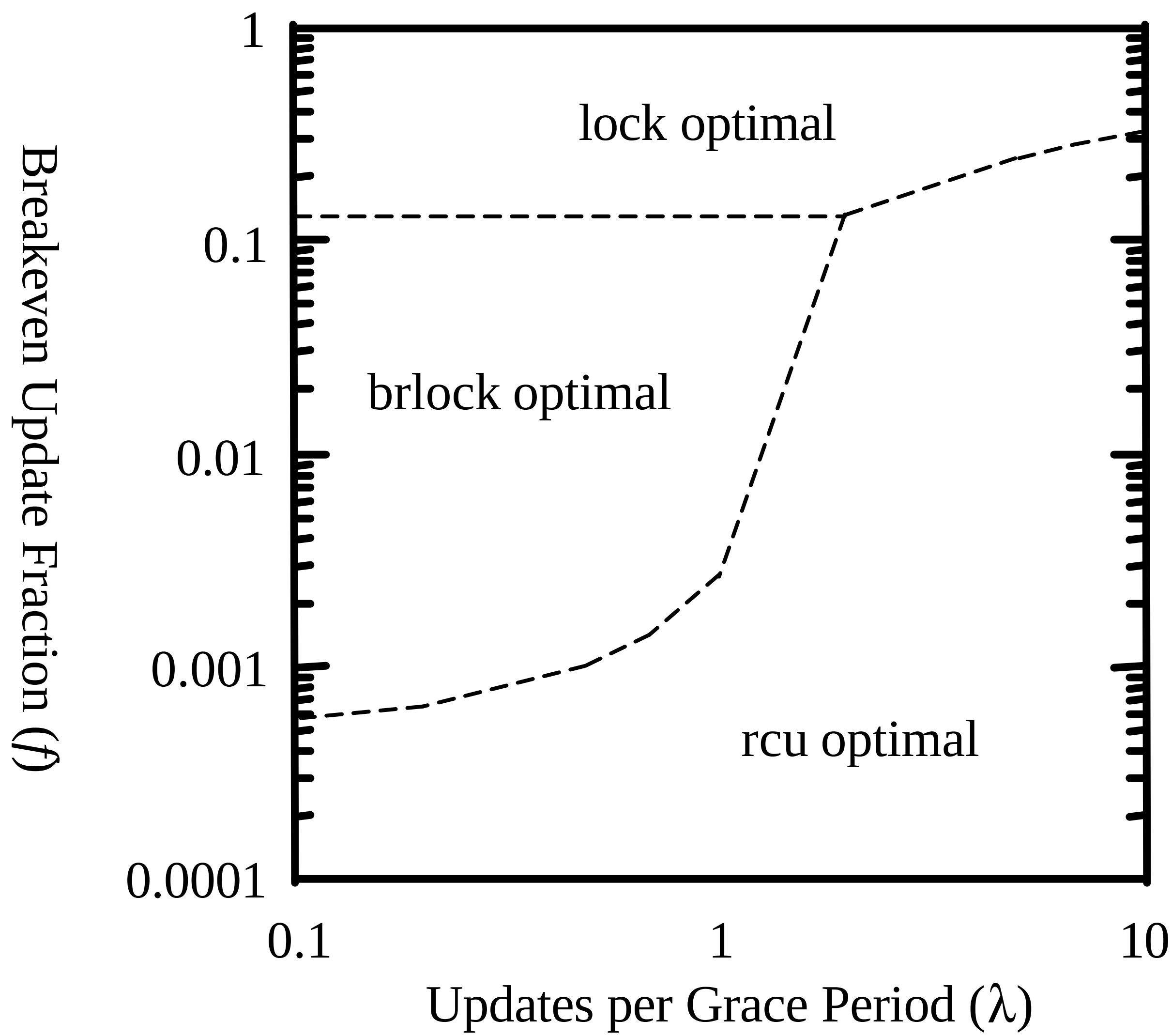


RCU Latencies (log scale)



When Used More -- Gets Better

4 CPUs
250:1
SMP



RCU Overhead (Profile Ticks)

Random thrashing of IP route cache with infrequent garbage collection

Name	2.5.3	rt_rcu_ltimer
ip_route_output_key	4486	2026
call_rcu	N/A	11
rcu_process_callbacks	N/A	4
rcu_invoke_callbacks	N/A	4
Total	4486	2045

RCU Overhead (Profile ticks)

Random thrashing of IP route cache with default garbage collection

Name	2.5.3	rt_rcu_ltimer
ip_route_output_key	2358	1646
call_rcu	N/A	262
rcu_invoke_callbacks	N/A	57
rcu_process_callbacks	N/A	49
rcu_check_quiescent_state	N/A	27
rcu_check_callbacks	N/A	24
rcu_reg_batch	N/A	3
Total	2358	2068

RCU Implementation Complexity

(Size of unified diff patch in lines)

Name	*All Archs*	*One Arch*
rcu-taskq	237	237
rcu-poll	378	378
X-rcu	424	424
rcu-sched	575	333
rcu	603	603
rcu-preempt	682	682
rcu-ltimer	712 (+371,-2)	514

Overview

- Background on read-copy update (RCU)
- Linux implementations of RCU
- Experience with Linux RCU implementations
- *Implementation details*
  - *rcu_poll Implementation*
  - *Handling CONFIG_PREEMPT*
  - *Memory ordering and list macros*
- Concluding remarks

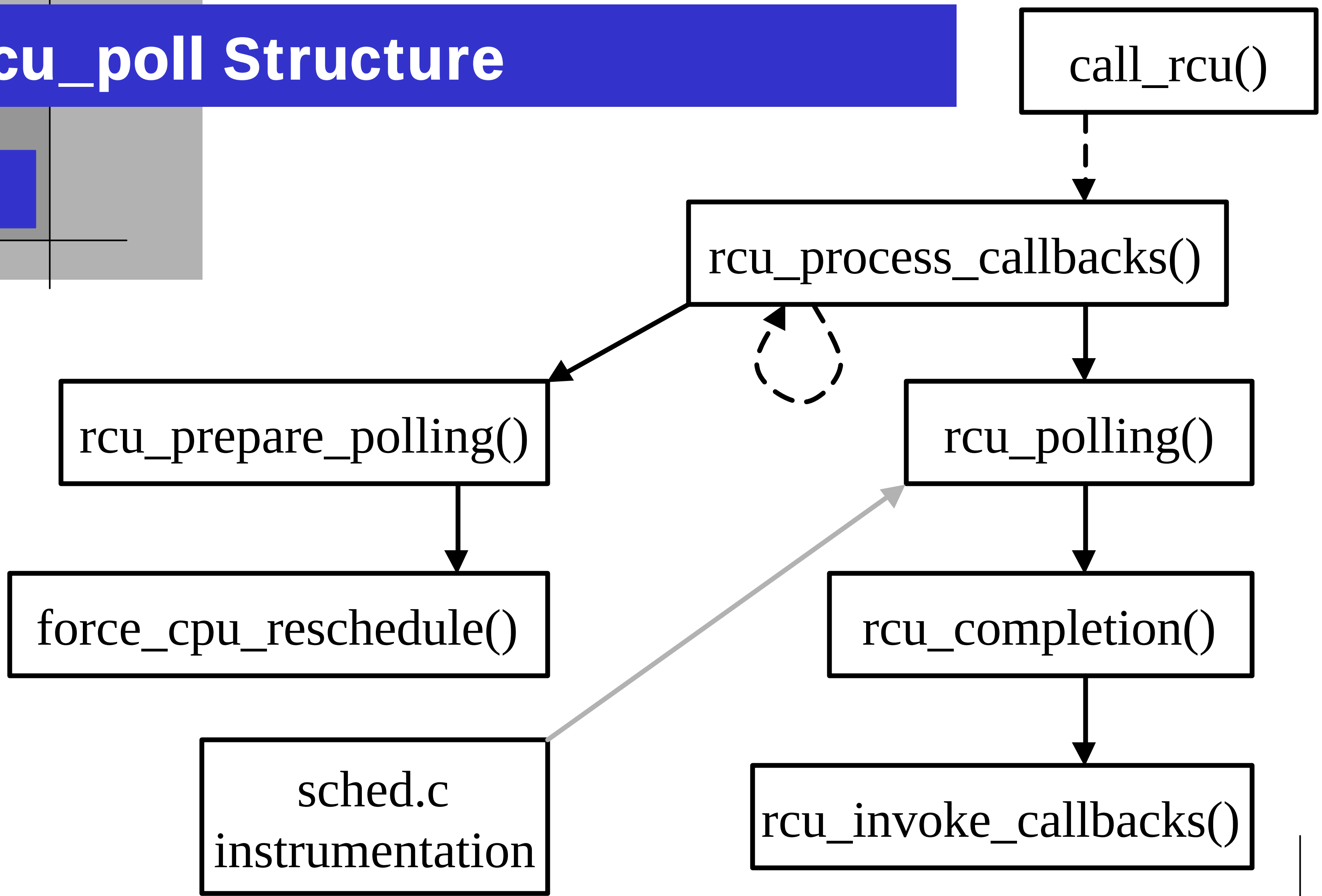
rcu_poll

- struct rcu_head
- call_rcu()
- scheduler instrumentation
- rcu_process_callbacks()
- rcu_prepare_polling()
- force_cpu_reschedule()
- rcu_polling()
- rcu_completion()
- rcu_invoke_callbacks()

struct rcu_head

```
struct rcu_head {
        struct list_head list;
        void (*func)(void *obj);
        void *arg;
};
```

rcu_poll Structure



rcu_poll call_rcu()

```
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg)
{
    head->func = func;
    head->arg = arg;

    spin_lock_bh(&rcu_lock);
    list_add(&head->list, &rcu_nxtlist);
    spin_unlock_bh(&rcu_lock);

    tasklet_hi_schedule(&rcu_tasklet);
}
```

sched.c Instrumentation

```
@@ -685,6 +686,7 @@
 switch_tasks:
         prefetch(next);
         prev->work.need_resched = 0;
+        RCU_quiescent(prev->cpu)++;

         if (likely(prev != next)) {
                 rq->nr_switches++;
```

rcu_process_callbacks

```
static void rcu_process_callbacks(
                          unsigned long data)
{
     int stop;

     spin_lock(&rcu_lock);
     if (!rcu_polling_in_progress)
          stop = rcu_prepare_polling();
     else
          stop = rcu_polling();
     spin_unlock(&rcu_lock);

     if (!stop)
          tasklet_hi_schedule(&rcu_tasklet);
}
```

rcu_prepare_polling

```
    stop = 1;
    if (!list_empty(&rcu_nxtlist)) {
        list_splice(&rcu_nxtlist, &rcu_curlist);
        INIT_LIST_HEAD(&rcu_nxtlist);
        rcu_polling_in_progress = 1;

        for (i = 0; i < smp_num_cpus; i++) {
            int cpu = cpu_logical_map(i);
            rcu_qsmask |= 1UL << cpu;
            rcu_quiescent_checkpoint[cpu] =
                            RCU_quiescent(cpu);
            force_cpu_reschedule(cpu);
        }
        stop = 0;
    }
```

force_cpu_reschedule()

```
rq = cpu_rq(cpu);
p = rq->curr;
newrq = lock_task_rq(p, &flags);
if (newrq == rq)
        resched_task(p);
unlock_task_rq(newrq, &flags);
```

rcu_polling()

```
static int rcu_polling(void)
{
    int i;
    int stop;

    for (i = 0; i < smp_num_cpus; i++) {
        int cpu = cpu_logical_map(i);

        if (rcu_qsmask & (1UL << cpu))
            if (rcu_quiescent_checkpoint[cpu]
                  != RCU_quiescent(cpu))
                rcu_qsmask &= ~(1UL << cpu);
    }

    stop = 0;
    if (!rcu_qsmask)
        stop = rcu_completion();

    return stop;
}
```

rcu_completion

```
static int rcu_completion(void)
{
	int stop;

	rcu_polling_in_progress = 0;
	rcu_invoke_callbacks();

	stop = rcu_prepare_polling();

	return stop;
}
```

rcu_invoke_callbacks()

```
static void rcu_invoke_callbacks(void)
{
    struct list_head *entry;
    struct rcu_head *head;


    entry = rcu_curlist.prev;
    do {
        head = list_entry(entry,
                 struct rcu_head, list);
        entry = entry->prev;

        head->func(head->arg);
    } while (entry != &rcu_curlist);

    INIT_LIST_HEAD(&rcu_curlist);
}
```

Handling Preemption

- Use per-CPU counts of preempted tasks
- Grace period ends when all tasks running or preempted at the beginning of the grace period have executed a *voluntary* context switch
  - rcu_preempt_get() upon task preempted
  - rcu_preempt_put() when preempted task does voluntary context switch or exits

Handling Preemption

Preemption path (preempt_schedule() in sched.c):

```
@@ -857,6 +882,7 @@
                        return;
        ti->preempt_count = PREEMPT_ACTIVE;
+       rcu_preempt_get();
        schedule();
        ti->preempt_count = 0;
        barrier();
```

Handling Preemption

Voluntary-context-switch path
(schedule() in sched.c)

```
@@ -773,8 +794,11 @@
          * if entering from preempt_schedule,
          * off a kernel preemption,
          * go straight to picking the next task.
          */
        if (unlikely(preempt_get_count() & PREEMPT_ACTIVE))
                goto pick_next_task;
+       else
+               rcu_preempt_put();
        switch (prev->state) {
        case TASK_INTERRUPTIBLE:
```

Handling Preemption

Switch counters at beginning of grace period:

```
static inline void rcu_switch_preempt_cntr(int cpu)
{
        atomic_t *tmp;
        tmp = per_cpu(curr_preempt_cntr, cpu);
        per_cpu(curr_preempt_cntr, cpu) =
                per_cpu(next_preempt_cntr, cpu);
        per_cpu(next_preempt_cntr, cpu) = tmp;
}
```

Handling Preemption

rcu_preempt_get:

```
static inline void rcu_preempt_get(void)
{
        if (likely(current->cpu_preempt_cntr == NULL)) {
                current->cpu_preempt_cntr =
                                this_cpu(next_preempt_cntr);
                atomic_inc(current->cpu_preempt_cntr);
        }
}
```

Handling Preemption

rcu_preempt_put:

```
static inline void rcu_preempt_put(void)
{
        if (unlikely(current->cpu_preempt_cntr != NULL)) {
                atomic_dec(current->cpu_preempt_cntr);
                current->cpu_preempt_cntr = NULL;
        }
}
```

Handling Preemption

Detecting end of grace period:

```
return ((rdata->qsmask & (1UL << cpu)) &&
        (rdata->quiescent_checkpoint[cpu] !=
                                  RCU_quiescent(cpu)) &&
        !rcu_cpu_preempted(cpu));
```

Memory Ordering and Alpha...

From HP/Compaq's web site describing Alpha:

For instance, your producer must issue a "memory barrier" instruction after writing the data to shared memory and before inserting it on the queue; likewise, your consumer must issue a memory barrier instruction after removing an item from the queue and before reading from its memory. Otherwise, you risk seeing stale data, since, while the Alpha processor does provide coherent memory, it does not provide implicit ordering of reads and writes. (That is, the write of the producer's data might reach memory after the write of the queue, such that the consumer might read the new item from the queue but get the previous values from the item's memory.)

RCU List Macros

- list_add_rcu(): wmb() between setting new elements pointers and setting pointers to it
- list_add_tail_rcu: ditto
- list_for_each_rcu(): read_barrier_depends() between fetching pointer and dereferencing it
- list_for_each_safe_rcu(): ditto

Using RCU List Macros

Adding an element:

```
spin_lock(&mylock);
list_add_tail_rcu(new, head);
spin_unlock(&mylock);
```

Searching the list:

```
rcu_read_lock();
list_for_each_rcu(p, head) {
        if (p->key == mykey) {
                break;
        }
}
rcu_read_unlock();
```

Overview

- Background on read-copy update (RCU)
- Linux implementations of RCU
- Experience with Linux RCU implementations
- Implementation details
- *Concluding remarks*

Future Work

- Continue performance measurement/tuning
  - Grace-period latency
  - per-call_rcu() overhead
  - complexity
- Continue investigating RCU uses in Linux
- Analysis at high contention levels
- Formal description and correctness proofs

Availability

- Read-copy update may be freely used under GPL.
  - http://sourceforge.net/projects/lse/

Conclusions

- A number of Linux implementations available
  - Continuing complexity and performance analysis
  - Performance benefits are quite real
- Patches available for preemptive environments
  - Both preemptive and non-preemptive grace periods required
- Additions to list-macro set greatly simplifies use of RCU

#include <disclaimers.h>

1. The views expressed in this paper are the authors' only, and should not be attributed to SuSE or IBM.
2. IBM, DYNIX/ptx, NUMA-Q, and Sequent are registered trademarks of International Business Machines Corporation in the United States and/or other countries.
3. Linux is a registered trademark of Linus Torvalds.
4. Other company, product, and service names may be trademarks or service marks of others.