Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel

Andrea Arcangeli
SuSE Labs
Mingming Cao, Paul E. McKenney, and Dipankar Sarma
IBM Corporation

Abstract

Read-copy update (RCU) allows lock-free read-only access to data structures that are concurrently modified on SMP systems. Despite the concurrent modifications, read-only access requires neither locks nor atomic instructions, and can often be written as if the data were unchanging, in a "CS 101" style. RCU is typically applied to read-mostly linked structures that the read-side code traverses unidirectionally.

Previous work has shown no clear best RCU implementation for all measures of performance. This paper combines ideas from several RCU implementations in an attempt to create an overall best algorithm, and presents a RCU-based implementation of the System V IPC primitives, improving performance by more than an order of magnitude, while increasing code size by less than 5% (151 lines). This implementation has been accepted into the Linux 2.5 kernel.

1 Introduction

The past two years have seen much discussion of RCU, along with the design and coding of a number of implementations and uses of RCU, one of which is now part of the Linux 2.5 kernel [Sarma02].

Comparisons with other concurrent update mechanisms [McK01b, Linder02a] have shown that RCU can greatly simplify and improve performance of code accessing read-mostly linked-list data structures. This paper adds a performance evaluation of RCU applied to the Linux System-V IPC primitives. RCU can also improve performance of code modifying linked-list structures when there is a high system-wide aggregate update rate across all such structures [McK98a].

Comparison of multiple RCU implementations [McK02a] showed, as noted in the abstract, that there is no overall best algorithm. The *rcu-poll* algorithm had the shortest latency, while the *rcu-ltimer* algorithm had the lowest overhead. This paper presents a parallelized variant of *rcu-poll* in an attempt to gain the best of both worlds.

The views expressed in this paper are the authors' only, and should not be attributed to SuSE or IBM.

Section 2 provides background on RCU, Section 3 reviews attempts to produce a single best RCU implementation, Section 4 describes an RCU-based implementation Linux's System V IPC primitives, and Section 5 describes future plans.

2 Background

This section gives a brief overview of RCU; more details are available elsewhere [McK98a, McK01b, McK02a]. Section 2.1 presents a visual example of RCU-based list manipulation, Section 2.2 contains a glossary of RCU-related terms, Section 2.3 presents concepts, Section 2.4 presents the RCU API, and Section 2.5 illustrates an analogy between RCU and reader-writer locking.

2.1 Example

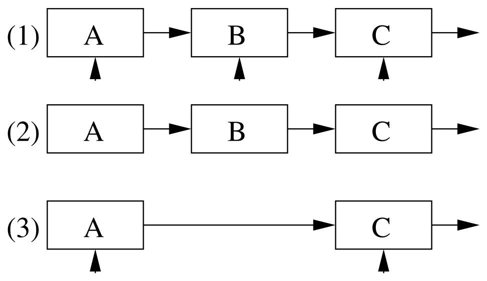

Figure 1: Lock Protecting Deletion and Search

On SMP systems, any searching of or deletion from a linked list must be protected by a lock. When element B is deleted from the list shown in Figure 1, searching code is guaranteed to see this list in either the initial state (1) or the final state (3). In state (2), when element B is being deleted, the reader-writer lock guarantees that no readers (indicated by the triangles) will be accessing the list.

However, many lists are searched much more often than they are modified. For example, an IP routing table would normally change at most once per few minutes,

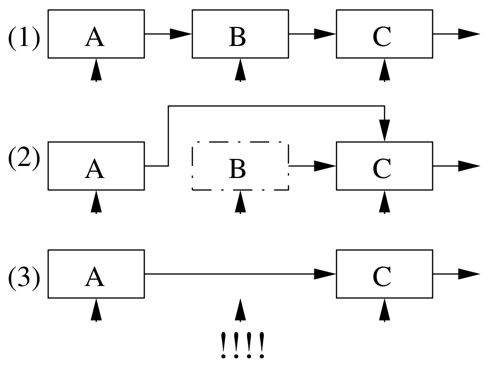

Figure 2: Race Between Deletion and Search

but might be searched many thousands of times per second. This could result in well over a million accesses per update, making lock-acquisition overhead burdensome to searches.

Unfortunately, omitting locking when searching means that the update no longer appears to be atomic. Instead, the update takes the multiple steps shown in Figure 2. A search might be referencing element B just as it was freed up, resulting in crashes, or worse, as indicated by the reader referencing nothingness in step (3).

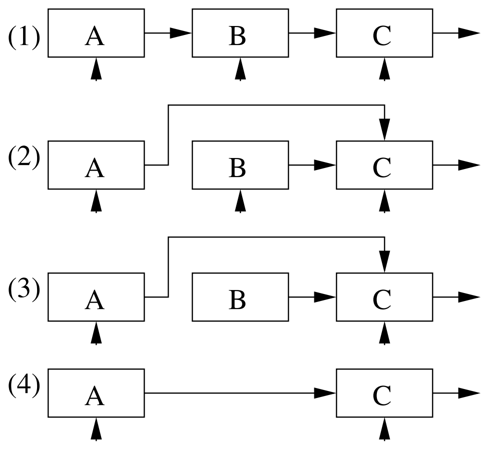

Figure 3: RCU Protecting Deletion and Search

One solution to this problem is to delay freeing up element B until all searches have given up their references to it, as shown in Figure 3. RCU indirectly determines when all references have been given up. To see how this works, recall that there are normally restrictions on what operations may be performed while holding a lock. For example, in the Linux kernel, it is forbidden to do a context switch while holding any spinlock. RCU mandates these same restrictions: even though the RCU-protected search need not acquire any locks, it is forbidden from performing any operation that would be forbidden if it were in fact holding a lock.

Therefore, any CPU that is seen performing a context switch after the linked-list deletion shown in step (2) of Figure 3 cannot possibly hold a reference to element B. As soon as all CPUs have performed a context switch, there can no longer be any readers, as shown in step (3). Element B may then be safely freed, as shown in step (4).

A simple, though inefficient, RCU-based deletion algorithm could perform the following steps in a non-preemptive Linux kernel (preemptive kernels can be handled as well [McK02a]):

1. Unlink element B from the list, but do *not* free it. The state of the list will be that shown in step (2) of Figure 3.
2. Run on each CPU in turn. At this point, each CPU has performed one context switch after element B has been unlinked. Thus, there cannot be any more references to element B.
3. Free up element B.

Much more efficient implementations have been described elsewhere [McK98a, McK02a]. The following sections present the concepts underlying RCU.

2.2 Glossary

The following definitions help illuminate the concepts underlying RCU.

Live Variable: A variable that might be accessed before it is next modified, so that its current value has some possibility of influencing future execution state.

Dead Variable: A variable that will be modified before it is next accessed, so that its current value cannot possibly have any influence over future execution state.

Critical Section: A region of code that is protected from outside interference by some locking mechanism, such as spinlocks or RCU.

Read-Side Critical Section: A region of code that is protected from other CPUs' modifications, but which allows multiple CPUs to read simultaneously.

Temporary Variable: A variable that is only live inside a critical section. One example is an auto variable used as a pointer while traversing a linked list.

Permanent Variable: A variable that is live outside of critical sections. One example would be the header for a linked list. Although it is possible for the same variable to be temporary sometimes and permanent at other times, this practice can lead to confusion, so is not generally recommended. Relying on the register-allocation capabilities of modern optimizing compilers is usually a far better strategy.

Quiescent State: A point in the code where all of the current CPU's temporary variables that were previously in use in a critical section are dead. In a non-preemptive Linux kernel, a context switch is a quiescent state for CPUs. In a preemptive Linux kernel, `rcu_read_lock()` and `rcu_read_unlock()` suppress preemption for short read-side critical sections, so that context switch is still a quiescent state with respect to these read-side critical sections. Although there are implementations of RCU that do not require preemption to be suppressed [Gamsa99, McK02a], they can be prone to excessively long grace periods.

Grace Period: Time interval during which all CPUs pass through at least one quiescent state. The key property of a grace period is that the values that were contained in any temporary variable in use in a critical section at the beginning of the grace period cannot possibly have any direct effect after the end of the grace period. This property will be examined more closely in the next section. Note that any time interval containing a grace period is itself a grace period.

2.3 Concepts

Without special action in the deletion code, the search code would be prone to races with those deletions, described in Section 2.1. To handle such race conditions, the update side performs the update in two phases as follows: (1) remove permanent-variable pointers to the item being deleted, and (2) after a grace period has elapsed, free up the item's memory.

The grace period is not a fixed time duration, but is instead inferred by checking for per-CPU quiescent states, such as context switches in non-preemptive environments. Since kernel threads are prohibited from holding locks across a context switch, they also prohibited from holding pointers to data structures protected by those locks across context switches–after all, the entire data structure could well be deleted by some other CPU at any time this CPU does not hold the lock.

A trivial implementation of RCU in a non-preemptive kernel could simply declare the grace period over once it observed each CPU undergoing a context switch. Once the grace period completes, no CPU references the deleted item, and no CPU can gain such a reference. It is therefore safe to free up the deleted item.

With this approach, searches already in progress when the first phase executes might (or might not) see the item being deleted. However, searches that start after the first phase completes are guaranteed to never reference this item. Efficient mechanisms for determining the duration of the grace period are key to RCU, and are described fully elsewhere [McK02a]. These mechanisms track "naturally occurring" quiescent states, which removes the need for adding expensive context switches. In addition, these mechanisms take advantage of the fact that a single grace period can satisfy multiple concurrent updates, amortizing the cost of detecting a grace period over the updates.

```
void synchronize_kernel(void);
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg);
struct rcu_head {
        struct list_head list;
        void (*func)(void *obj);
        void *arg;
};
void rcu_read_lock(void);
void rcu_read_unlock(void);
```

Figure 4: RCU API

2.4 RCU API

Figure 4 shows the external API for RCU. The `synchronize_kernel()` function blocks for a full grace period. This is a simple, easy-to-use function, but imposes expensive context-switch overhead on its caller. It also cannot be called with locks held or from softirq and irq (interrupt) contexts.

Another approach, taken by `call_rcu()`, is to schedule a callback function `func` to be called with argument `arg` after the end of a full grace period. Since `call_rcu()` never sleeps, it may be called with locks held. It may also be called from the softirq and irq contexts. The `call_rcu()` function uses its `struct rcu_head` argument to remember the specified callback function (in the `func` field) and argument (in the `arg` field) for the duration of the grace period. At the end of the grace period, the RCU subsystem invokes the function pointed to by the `func` field, passing it the contents of the `arg` field. An `rcu_head` is often placed within a structure being protected by RCU, eliminating the need to separately allocate it.

The primitives `rcu_read_lock()` and `rcu_read_unlock()` demark a read-side RCU critical section, but generate no code in non-preemptive kernels. In preemptive kernels, they disable preemption within the critical section, which is required because both `synchronize_kernel()` and `call_rcu()` declare the grace period to be over once each CPU has completed a context switch. Suppressing preemption during the read-side RCU critical section prevents the crashes that could result from such prematurely ending grace periods. There is a patch that implements a proposed `call_rcu_preempt()` [McK02a] that tolerates preemption in read-side critical sections (based on the K42 implementation [Gamsa99]), but at this writing, there are no RCU uses envisioned in Linux

```
list_add_rcu(struct list_head *new,
             struct list_head *head);
list_add_rcu_tail(struct list_head *new,
                  struct list_head *head);
list_del_rcu(struct list_head *entry);
list_for_each_rcu(struct list_head *pos,
                  struct list_head *head);
list_for_each_safe_rcu(struct list_head *pos,
                       struct list_head *n,
                       struct list_head *head);
```

Figure 5: RCU List API

```
struct el {
     struct list_head list;
     long key;
     long data;
     struct rcu_head my_rcu_head;
};
```

Figure 6: List Element Data Structure

that could tolerate the extremely long grace periods that might result from preemption on a busy system.

Modern microprocessors, particularly the DEC/Compaq/HP Alpha, feature very weak memory consistency models. These models require use of special "memory barrier" instructions. However, since proper use of these instructions is often difficult to understand and even more difficult to build good test suites for, there is an extension to the Linux list-manipulation API for use with RCU, as shown in Figure 5. Each primitive in this extension is equivalent to its non-RCU counterpart, but with the addition of whatever memory-barrier instructions are required on the machine in question [Spraul01].

RCU may be applied to data structures other than lists, but in such cases, memory-barrier instructions must be used explicitly. An example of such a situation is shown in Section 4.5.

2.5 Reader-Writer-Locking/RCU Analogy

Although RCU has been used in a great many interesting and surprising ways, one of the most straightforward is as a replacement for reader-writer locking. In this section, we present this analogy, protecting the simple doubly-linked-list data structure shown in Figure 6 with reader-writer locks and then with RCU, comparing the results. This structure has a `struct list_head` that is manipulated by the standard Linux list-manipulation primitives, a search key, and a single integer for data. The RCU primitive `call_rcu()` requires some space in the data element, which is supplied by the `my_rcu_head` field.

The reader-writer-lock/RCU analogy substitutes primitives as shown in Table 1. The asterisked primitives are no-ops in non-preemptible kernels; in preemptible kernels, they suppress preemption, which is an extremely cheap operation on the local task structure. Note that since neither `rcu_read_lock()` nor `rcu_read_unlock` block irq or softirq contexts, it is necessary to add primitives for this purpose where needed. For example, `read_lock_irqsave` must become `rcu_read_lock()` followed by `local_irq_save()`.

Reader-Writer Lock	Read-Copy Update
`rwlock_t`	`spinlock_t`
`read_lock()`	`rcu_read_lock()` *
`read_unlock()`	`rcu_read_unlock()` *
`write_lock()`	`spin_lock()`
`write_unlock()`	`spin_unlock()`
`list_add()`	`list_add_rcu()`
`list_add_tail()`	`list_add_tail_rcu()`
`list_del()`	`list_del_rcu()`
`list_for_each()`	`list_for_each_rcu()`

* no-op unless CONFIG_PREEMPT, in which case preemption is suppressed

Table 1: Reader-Writer-Lock/RCU Substitutions

Deletion from the list is illustrated by the upper section of Table 2. The `write_lock()` and `write_unlock()` are replaced by `spin_lock()` and `spin_unlock()`, respectively, the `list_del()` is replaced by `list_del_rcu()`, and `my_free()` is replaced by `call_rcu()`. The `call_rcu()` will, after a grace period elapses, pass `p` to function `my_free()`, using the `struct rcu_head` in the `my_rcu_head` field to keep track of the deferred function call and argument.

Insertion into the list is illustrated by the second section of Table 2. Again, the `write_lock()` and `write_unlock()` are replaced by the simple spinlock primitives `spin_lock()` and `spin_unlock()`, respectively. The `list_add_tail()` is replaced by `list_add_tail_rcu()`. The rest of the code remains the same.

Searching the list is illustrated by the third section of Table 2. Locking is handled by the caller, so the two variants differ only in that the `list_for_each()` is replaced by `list_for_each_rcu()`.

Searching the list for read-only access is illustrated by the last section of Table 2. The only difference is that the `read_lock()` and `read_unlock()` primitives are replaced by `rcu_read_lock()` and `rcu_read_unlock()`, respectively. Again, the bulk of the code remains the same for both cases.

Although this analogy can be quite compelling and useful, there are some caveats:

1. Read-side critical sections may see "stale data," that has been removed from the list but not yet

Reader-Writer Lock

```
 1 void delete(long mykey)
 2 {
 3    struct el *p;
 4    write_lock(&list_lock);
 5    p = search(mykey);
 6    if (p != NULL) {
 7       list_del(p);
 8    }
 9    write_unlock(&list_lock);
10    my_free(p);
11 }
```

```
 1 void insert(long key, long data)
 2 {
 3     struct el *p;
 4     p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5     p->key = key;
 6     p->data = data;
 7     write_lock(&list_lock);
 8     list_add_tail(&(p->list), &head);
 9     write_unlock(&list_lock);
10 }
```

```
 1 struct el *search(long mykey)
 2 {
 3     struct el *p;
 4     list_for_each(p, &head) {
 5        if (p->key == mykey) {
 6           return (p);
 7        }
 8     }
 9     return (NULL);
10 }
```

```
 1 /* Read-only search */
 2 struct el *p;
 3 read_lock(&list_lock);
 4 p = search(mykey);
 5 if (p == NULL) {
 6    /* handle error condition */
 7 } else {
 8    /* access *p w/out modifying */
 9 }
10 read_unlock(&list_lock);
```

Read-Copy Update

```
 1 void delete(long mykey)
 2 {
 3    struct el *p;
 4    spin_lock(&list_lock);
 5    p = search(mykey);
 6    if (p != NULL) {
 7       list_del_rcu(p);
 8    }
 9    spin_unlock(&list_lock);
10    call_rcu(&p->rcuhead,
11             (void (*)(void *))my_free, p);
12 }
```

```
 1 void insert(long key, long data)
 2 {
 3     struct el *p;
 4     p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5     p->key = key;
 6     p->data = data;
 7     spin_lock(&list_lock);
 8     list_add_tail_rcu(&(p->list), &head);
 9     spin_unlock(&list_lock);
10 }
```

```
 1 struct el *search(long mykey)
 2 {
 3     struct el *p;
 4     list_for_each_rcu(p, &head) {
 5        if (p->key == mykey) {
 6           return (p);
 7        }
 8     }
 9     return (NULL);
10 }
```

```
 1 /* Read-only search */
 2 struct el *p;
 3 rcu_read_lock(); /* nop unless CONFIG_PREEMPT */
 4 p = search(mykey);
 5 if (p == NULL) {
 6    /* handle error condition */
 7 } else {
 8    /* access *p w/out modifying */
 9 }
10 rcu_read_unlock(); /* nop unless CONFIG_PREEMPT */
```

Table 2: Reader-Writer-Lock and Read-Copy-Update Analogy

freed. There are some situations (e.g., routing tables for best-effort protocols) where this is not a problem. In other situations, such stale data may be detected and rejected [Pugh90], as illustrated in Section 4.

2. Read-side critical sections may run concurrently with write-side critical sections.
3. The grace period will delay freeing of memory, which means that both the memory and the cache footprint of the code will be somewhat larger when using RCU than when using reader-writer locking.
4. When changing to RCU, write-side reader-writer locking code that modifies list elements in place must often be restructured to prevent read-side RCU code from seeing the data in an inconsistent state. In many cases, this restructuring will be quite straightforward, for example, creating a new list element with the desired state, then replacing the old element with the new.

Where it applies, this analogy can deliver full parallelism with almost no increase in complexity. For example, Section 4 shows how applying this analogy to System V IPC yields order-of-magnitude speedups with a very small increase in code size and complexity.

RCU has also been used as a lazy barrier synchronization mechanism, as a mode-change control mechanism, as well as for more sophisticated list maintenance. Retrofitting existing code with RCU as shown above can produce significant performance gains, but of course the best results are obtained by designing RCU into the algorithms and code from the start.

Other methods have been proposed for eliminating locking [Herlihy93], and, when combined with more recent refinements [Michael02a, Michael02b], these methods are practical in some circumstances. However, they still require expensive atomic operations on shared storage, resulting in pipeline stalls, cache thrashing, and memory contention, even for read-only accesses.

3 RCU Implementation

The performance of RCU is critically dependent on an efficient implementation of the `call_rcu()` primitive. The more efficient the implementation, the greater the number of situations that RCU may profitably be applied to.

However, grace-period latency is also an important measure of performance – while CPU overhead negates the performance benefits of RCU, excessively long grace-period latencies can result in all available memory queued up waiting for a grace period to end. However, the conditions that cause excessively long grace-period latencies have other bad effects, even in absence of RCU, such as grossly degraded response times.

A detailed description of these algorithms has been presented elsewhere [McK02a], but a brief description of each follows. The *rcu_poll* algorithm uses IPIs to force each CPU to quickly enter a quiescent state, which results in grace periods of much less than a millisecond on idle systems. However, the resulting increased scheduler overhead can outweigh the performance benefits of RCU. The *rcu_ltimer* algorithm instruments the `scheduler_tick()` function that is called with HZ frequency on each CPU, resulting in extremely low overhead, but with grace periods in excess of 100 milliseconds. The *rcu_sched* algorithm uses a token-passing scheme that holds the promise of extremely low overheads (which are currently masked by the cache thrashing of a global counter), but can have grace periods of almost one *minute* in duration. It is likely that the *rcu_sched()* algorithm can be reworked to eliminate these disadvantages, which may result in it being the best overall algorithm. However, the *rcu_ltimer* is the best match for current RCU uses in the Linux kernel, so this algorithm is implemented in the 2.5 Linux kernel.

The best grace-period latencies were obtained using *rcu-poll*, which invokes `resched_task()` on each CPU to force context switches, resulting in latencies more than an order of magnitude shorter than those of *rcu-ltimer* and *rcu-sched*, as shown in Figure 7. This benchmark was run on an 8-CPU 700MHz Intel Xeon system with 1MB L2 caches and 6GB of memory using the *dcache-rcu* patch [LSE].

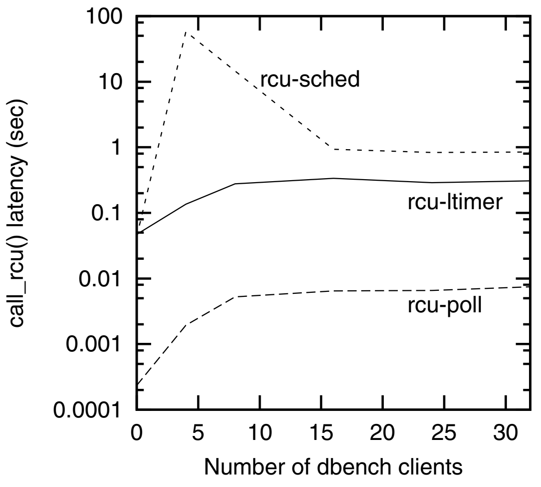

Figure 7: call_rcu() Latency Under dbench Load (logscale)

However, *rcu-poll*'s single global callback list results in cache thrashing and high overhead, and it obtains the short grace-period latencies by frequently invoking the scheduler, the combination of which at times overwhelms the performance benefits of RCU. The cache thrashing is caused by: (1) enqueuing onto this single list from multiple CPUs and (2) callbacks running on a CPU other than the one that registered them. This latter effect causes data structures processed by the callback to be pulled from the registering CPU to the CPU running the callback.

A modified *rcu-poll* algorithm was constructed having per-CPU lists of callbacks. This eliminates both sources of cache thrashing: each CPU manipulates only its own list and callbacks always run on the CPU that registered them. The frequent scheduler invocation remains, as this is required to obtain the excellent `call_rcu()` latencies.

Table 3 compares the performance of *rcu-ltimer* (in Linux 2.5 kernel), *rcu-sched*, and the parallelized version of *rcu-poll*. These results show that *rcu-ltimer* completes each iteration slightly (but statistically significantly) more quickly than does *rcu-poll*, and with 8.6% less CPU utilization. They also show `rcu-sched` completes each iteration as fast as does `rcu-ltimer`, but with 5.7% more CPU utilization. This benchmark was run on a 4-CPU PIII Intel Xeon with 1MB L2 cache and 1GB of memory. Profile results show that *rcu-poll* is incurring significant overhead in the scheduler and in its `force_cpu_reschedule()` function, indicating that although its cache-thrashing behavior has been addressed, *rcu-poll* is buying its excellent grace-period latency with significantly increased overhead. The *rcu-sched* implementation is unchanged; future work includes optimizing it to eliminate cache thrashing and atomic instructions in order to reduce its overhead.

	CPU Utilization		ms/Iteration	
	Avg	Std	Avg	Std
rcu-ltimer (2.5)	77.52%	0.05%	22.47	0.01
rcu-poll	84.20%	0.13%	22.95	0.03
rcu-sched	81.95%	0.20%	22.46	0.02

Table 3: dcachebench Comparison

Therefore, unless grace-period latency is of paramount concern, the *rcu-ltimer* implementation of RCU should be used. Should latency become a critical issue in the future, we will investigate modifications to improve the latency of *rcu-ltimer*.

An optimized *rcu-sched* might beat *rcu-ltimer*'s overhead. If this is the case, reduction of grace-period latency would become a considerably more urgent matter.

4 RCU Implementation of System V IPC

This section describes how the reader-writer-lock/RCU analogy described in Section 2.5 was used to break up the global locks used by Linux's System V IPC primitives. These locks guard the following: (1) mapping from IPC identifiers to corresponding `kern_ipc_perm` structures, (2) expanding the mapping arrays, and (3) individual IPC operations. A straightforward modification would replace these global locks with reader-writer locks, allowing mapping operations to be performed in parallel.

However, we took the additional step of following the analogy, replacing the global locks with RCU [Cao02] to guard the mapping arrays and per-`kern_ipc_perm` locks to guard the IPC operations, which resulted in significant system-level speedups on database benchmarks. This modification also serves to illustrate use of explicit memory barriers and use of a `deleted` flag to prevent access to stale data.

The remainder of this section focuses on the changes to the System V semaphores; analogous changes were made to message queues and shared memory.

4.1 Semaphore Data Structures

The semaphore data structures are shown in Figure 8. The global `ipc_ids` structure tracks the state of all semaphores currently in use. Among other things, it contains a global lock `ary` and a pointer `entries` that points to an array of pointers of `ipc_id` structures. Each such entry is either NULL or points to a `sem_array` structure, which represents a set of semaphores that has been created by a single `semget()` system call. The array of `ipc_id` structures is dynamically expanded as required; see the discussion of the `grow_ary()` function in Section 4.5. The `sem_array` structure is allocated by a `semget()` system call and deleted by a `semctl(IPC_RMID)` system call. The individual semaphores in a set are each represented by a `sem` structure.

Each `semop()` system call presents the `semid` for the semaphore, which must be looked up in this data structure to locate the corresponding `sem_array`. Thus, each and every semaphore operation requires that this data structure be traversed.

The `ipc_ids` field `ary` is a `spinlock_t` that protects the entire data structure. This simple locking design prevents System-V semaphore operations from proceeding in parallel. In addition, the cacheline containing the `ary` spinlock is thrashed among all CPUs.

Use of RCU permits fully parallel operation of different semaphores and fully parallel translation of a semaphore ID into the corresponding `sem_array` pointer. However, a few changes to the data structure are required, as shown in Figure 9. To begin with, the `ipc_id` array and the `sem_array` are each prefixed with an `ipc_rcu_kmalloc` structure which contains the `rcu_head` structure that RCU's `call_rcu()`

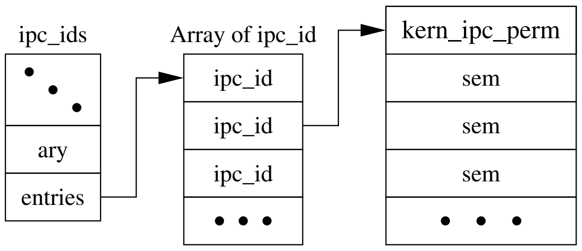

Figure 8: Semaphore Structures with Global Locking

function needs to track these structures during a grace period. In addition, since there is no longer a global ary lock, each individual sem_array must have its own individual lock to protect operations on the corresponding set of semaphores.

The final change is motivated by fact that the translation from semaphore ID to kern_ipc_perm cannot tolerate the stale data that could result when an ID translation races with an semctl(IPC_RMID) removing that same ID. The possibility of stale data is avoided through use of a deleted flag in the kern_ipc_perm structure, guarded by that structure's lock field. This deleted flag is set just after removing the corresponding sem_array but before starting the grace period. The entire removal operation is performed holding the lock field in the kern_ipc_perm structure. Any attempt to lock a semaphore structure that has the deleted flag set then behaves as if the structure is nonexistent, as will be shown in the following sections.

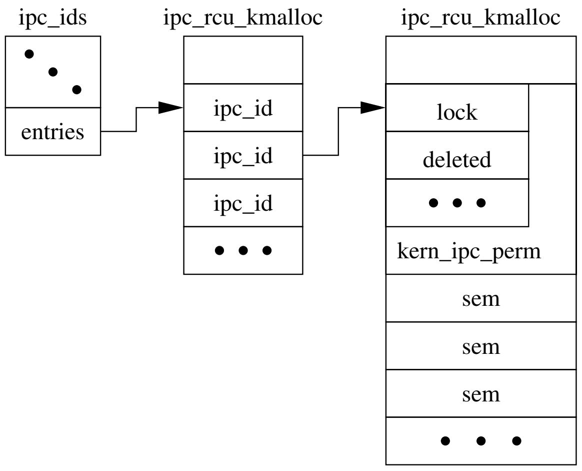

Figure 9: Semaphore Structures with RCU

4.2 Semaphore Removal

The deletion process is performed by ipc_rmid, as shown in Figure 10. This function is called with the

```
 1 struct kern_ipc_perm*
 2 ipc_rmid(struct ipc_ids* ids, int id)
 3 {
 4    struct kern_ipc_perm* p;
 5    int lid = id % SEQ_MULTIPLIER;
 6    if(lid >= ids->size)
 7       BUG();
 8
 9    p = ids->entries[lid].p;
10    ids->entries[lid].p = NULL;
11    if(p==NULL)
12       BUG();
13    ids->in_use--;
14
15    if (lid == ids->max_id) {
16       do {
17          lid--;
18          if(lid == -1)
19             break;
20       } while (ids->entries[lid].p == NULL);
21       ids->max_id = lid;
22    }
23    p->deleted = 1;
24    return p;
25 }
```

Figure 10: Semaphore Deletion

lock held, and returns with it held. Lines 4-9 obtain a pointer to the sem_array structure. Line 10 NULLs the pointer to sem_array, removing any path from a permanent variable to this structure. Lines 11-12 perform a debug check, which could be triggered by locking design bugs, among other things. Lines 13-22 adjust the count of semaphores in response to the removal of this one, and then, if this semaphore had the largest ID, scans down the array of ipc_ids to find the new largest ID. Line 23 sets the deleted flag, so that the next acquisition of the lock will fail (see Section 4.3 below), and Line 24 returns a pointer to the newly removed semaphore. The semaphore's memory is freed up by a call to ipc_rcu_free() by freeary(), which is ipc_rmid()'s caller and which also performs other cleanup actions, including waking up any processes that were sleeping on the newly removed semaphore.

The deleted flag, once set, makes the corresponding semaphore set appear to be freed up even though it is still in memory awaiting expiration of its grace period, as will be shown in the next section.

4.3 Semaphore Lock Acquisition

As noted earlier, each semop() system call presents the semid for the semaphore, which must be looked up to locate the corresponding sem_array. In addition, the semaphore state must be locked. The semop() system call invokes the ipc_lock() kernel function to do this lookup and locking, and later invokes the ipc_unlock() kernel function to do the corresponding unlocking. Since the ipc_lock() kernel function was responsible for the lock contention that motivated use of RCU, we focus on ipc_lock() and the functions that it interacts with.

```
 1 struct kern_ipc_perm*
 2 ipc_lock(struct ipc_ids* ids, int id)
 3 {
 4    struct kern_ipc_perm* out;
 5    int lid = id % SEQ_MULTIPLIER;
 6    struct ipc_id* entries;
 7
 8    rcu_read_lock();
 9    if(lid >= ids->size) {
10       rcu_read_unlock();
11       return NULL;
12    }
13    /* barrier syncs with grow_ary() */
14    smp_rmb();
15    entries = ids->entries;
16    read_barrier_depends();
17    out = entries[lid].p;
18    if(out == NULL) {
19       rcu_read_unlock();
20       return NULL;
21    }
22    spin_lock(&out->lock);
23    /* in case ipc_rmid() just freed ID */
24    if (out->deleted) {
25       spin_unlock(&out->lock);
26       rcu_read_unlock();
27       return NULL;
28    }
29    return out;
30 }
```

Figure 11: Detecting Semaphore Deletion

```
1 void ipc_rcu_free(void* ptr, int size)
2 {
3    struct ipc_rcu_kmalloc *free;
4    free = ptr - sizeof(*free);
5    call_rcu(&free->rcu,
6             (void (*)(void *))kfree,
7             free);
8 }
```

Figure 12: Freeing a Semaphore

Since the read-code is lock-free, nothing will prevent `ipc_lock()` from racing with `ipc_rmid()`, thus possibly gaining a reference to the structure after it is marked deleted. Figure 11 shows how `ipc_lock()` handles this race. Note that the `ids` argument is a pointer to the sole permanent variable, while the `out` pointer declared in Line 4 is a temporary variable. Line 5 computes the "hash" used to access the array of `ipc_ids`. Line 8 marks the beginning of the RCU read-side critical section. In preemptive kernels, this will disable preemption; in non-preemptive kernels, it does absolutely nothing other than serve as a documentation aid. Lines 9-12 check for the specified ID being out of range, returning `NULL` for failure if so. Line 14 allows for interactions with the `grow_ary()` function on multiprocessors with weak memory consistency models as described in Section 2.4. Note that since the semaphore implementation does not use linked lists, these memory-barrier primitives must be invoked explicitly – the RCU variants of the Linux list-manipulation primitives cannot be used. Lines 15-17 obtain a stable pointer to the semaphore structure. The `read_barrier_depends()` allows for interactions with the `grow_ary()` function on multiprocessors with extremely weak memory consistency models, such as the Alpha. Lines 18-21 attempt to get a reference to the semaphore structure, returning `NULL` if there is no such structure (perhaps due to the specified ID no longer being valid). Line 22 acquires the semaphore structure's `lock`. Lines 24-28 check the `deleted` flag to determine if the semaphore is being removed, and, if such a race occurred, returns `NULL` to signal failure. Note that because `ipc_lock()` does not block, the normal RCU grace period keeps the semaphore structure around for long enough that there is no danger of this structure being freed up before `ipc_lock()` can check the `deleted` flag. Finally, Line 29 returns a pointer to the semaphore structure, having successfully translated the specified ID. Note that this function returns with the semaphore's `lock` held inside a RCU read-side critical section. The `ipc_unlock()` function therefore releases the semaphore's `lock` and then ends the RCU read-side critical section by executing a `rcu_read_unlock()`.

4.4 Semaphore Deferred Deletion

Since `ipc_lock()` can gain a reference to a semaphore as it is being removed, a grace period must elapse between the removal and the actual freeing of the corresponding data structures, as illustrated by Figure 12, which shows a simplified version of `ipc_rcu_free()` function. The actual function is more complex due to the fact that blocks of memory larger than a page must be freed with `vfree()` rather than `kfree()`. Line 4 computes a pointer to the beginning of the structure (see Figure 9), which is an `ipc_rcu_kmalloc()`, which in turn is just a wrapper around an `rcu_head` structure (see Figure 4). This wrapping allows more common code between the `kmalloc()` and `vmalloc()` cases. Lines 5-7 then pass to `call_rcu()` pointers to the `rcu_head` structure, to the `kfree()` function, and to the semaphore structure. The `call_rcu()` function uses the `rcu_head` structure to queue up the semaphore structure during the grace period. The actual invocation of the `kfree()` function on the semaphore structure is deferred until after the end of a subsequent grace period.

4.5 Semaphore Array Expansion

If a large number of semaphores are created, the kernel will need to expand the `ipc_id` array. Use of RCU dictates that this expansion occur in parallel with ongoing searching by `ipc_lock()`. The function `grow_ary()`, shown in Figure 13, implements this expansion.

Lines 8-11 do limit checking. Lines 13-21 allocate the new array, copy the old array to the first part of the

```
 1 static int grow_ary(struct ipc_ids* ids,
 2                     int newsize)
 3 {
 4    struct ipc_id* new;
 5    struct ipc_id* old;
 6    int i;
 7
 8    if(newsize > IPCMNI)
 9       newsize = IPCMNI;
10    if(newsize <= ids->size)
11       return newsize;
12
13    new = ipc_rcu_alloc(sizeof(struct ipc_id) *
14                        newsize);
15    if(new == NULL)
16       return ids->size;
17    memcpy(new, ids->entries,
18           sizeof(struct ipc_id)*ids->size);
19    for(i=ids->size;i<newsize;i++) {
20       new[i].p = NULL;
21    }
22    old = ids->entries;
23    i = ids->size;
24
25    smp_wmb();
26    ids->entries = new;
27    smp_wmb();
28    ids->size = newsize;
29
30    ipc_rcu_free(old, sizeof(struct ipc_id)*i);
31    return ids->size;
32 }
```

Figure 13: Expanding the Array of Pointers to Semaphores

new array, and initialize the remainder of the new array. Lines 22 and 23 retain the size of the old array and a pointer to it. Line 25 is a memory barrier that prevents the CPU and the compiler from reordering the array initialization with the assignment of the pointer. Any such reordering could cause other CPUs to see uninitialized segments of the array, possibly crashing (or worse!). Line 26 switches the pointer over to the new array, but any accesses at this point will recognize only the old entries as valid, since the size is still the old size. Line 27 is a memory barrier that prevents the CPU and the compiler from reordering the assignment of the size to precede the pointer assignment. If such a reordering were to occur while a user was attempting to access an erroneously larger `semid`, the kernel would run off the end of the old array, again, possibly crashing. Line 28 updates the size, so that new semaphores with larger `semids` may now be accommodated. Line 30 invokes `ipc_rcu_free()`, which frees the old structure after a full grace period has elapsed. Note that `ipc_rcu_free()` returns immediately, having used `call_rcu()` to queue the old array for a later `kfree()`. Finally, Line 31 returns the new size of the array.

Note that no `deleted` flag is needed here, since the old version of the array is kept valid throughout the grace period. Any semaphore in existence at the start of the racing access that is still in existence when the racing access completes will still be correctly referenced by the old array. Note that the racing access must by definition complete before the grace period ends – otherwise, it is not a grace period.

This tolerance of stale data is typical of ID-to-address mappings, and of routing tables as well.

4.6 Semaphore Operation

This section presents a graphical demonstration of how `grow_ary()`, `ipc_rmid()`, and `ipc_lock` operate. The figures in this section are abbreviated forms of Figure 8 and Figure 9. Figure 14 shows a system with three semaphores allocated out of a maximum of eight that could be accommodated.

The results of a concurrent `grow_ary()`, `ipc_rmid()` and creation of a new semaphore are shown in Figure 15, but with the additional `ipc_id` array elements omitted from the figure. At this point, a concurrent `ipc_lock()` would see semaphore 4 as being deleted (note the "D" in the diagram), and would have no way of reaching the newly created semaphore 2. The lack of visibility to semaphore 2 is legal, since this semaphore was created *after* `ipc_lock()` started execution. A subsequent `ipc_lock()` would see semaphores 0, 2, and 6, but would not newly deleted semaphore 4.

Finally, Figure 16 shows the state of the system after a grace period. The old `ipc_id` array has been freed, as has semaphore 4. Because the grace period has completed, there can no longer be any references either to the old array or to the now-deleted semaphore 4.

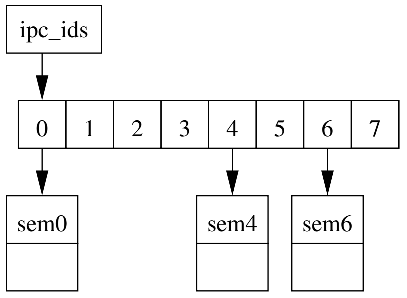

Figure 14: Semaphore Initial State

4.7 Semaphore Performance

Use of RCU improves the performance of System V semaphores as measured by both system-level benchmarks and focused microbenchmarks.

The Open Source Development Lab (OSDL) used a DBT1 benchmark to evaluate system-level performance, comparing Andrew Morton's Linux 2.5.42-mm2 both with and without *ipc-rcu*. These tests were run on an Intel$^{(R)}$ dual-CPU 900MHz PIII with 256MB of mem-

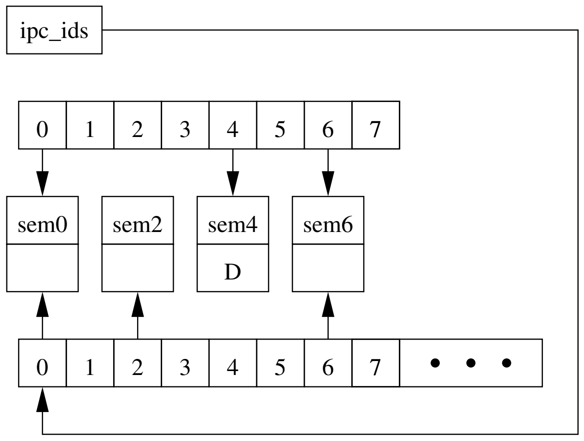

Figure 15: Semaphore Structures After Array Replacement

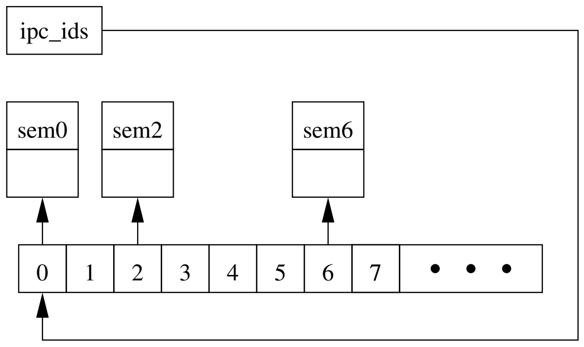

Figure 16: Semaphore Structures After Grace Period

ory.

The raw transaction rate for each of the five runs with each kernel are shown in Figure 17. The erratic results for the stock kernel are not unusual for workloads with lock contention. The reason for this is that if the lock contention is not too extreme, relatively deterministic workloads can "get lucky" such that multiple CPUs happen to be less likely to be contending for the same lock at the same time. As shown in Table 4, the difference is statistically significant: not only is *ipc-rcu*'s average three standard deviations above that of the stock kernel, but *ipc-rcu*'s smallest value of 90.4 TPS exceeds the stock kernel's median of 87.6 TPS.

Bill Hartner [Hartner02] constructed a System V

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

Table 4: DBT1 Database Benchmark Results (TPS)

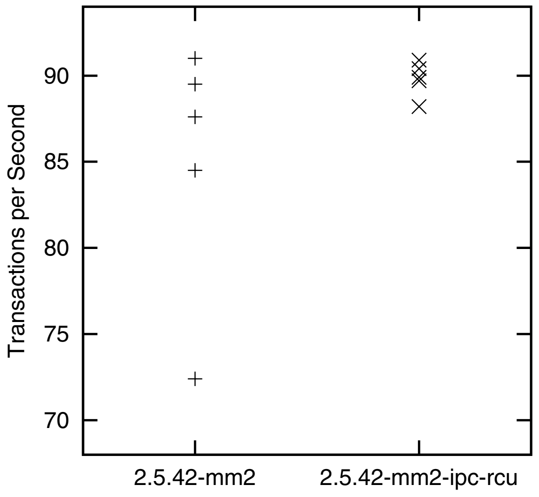

Figure 17: DBT1 Database Benchmark Raw Results

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

Table 5: semopbench Microbenchmark Results (seconds)

semaphore microbenchmark named semopbench and ran it on an Intel 8-CPU 700 MHz PIII system. The results in Table 5 clearly show the order-of-magnitude reduction in runtime obtained by applying the reader-writer-locking/RCU analogy with RCU to System V IPC mechanisms.

4.8 Semaphore Complexity

The RCU changes to the System V IPC implementations inflicted less than 5% expansion of code size, as shown in Table 6. This change increased the overall code size by only 151 lines. The RCU implememtation itself (which is also used by both module unloading and the IP route cache) adds only an additional 408 lines of code. This order-of-magnitude performance benefit is well worth the modest increase in complexity.

Of course, the system-level performance increase is a much smaller 5.3%. On the other hand, the 151-line increase in code size is an insignificant fraction of the 11.7 million lines of code in the full kernel, and even this does not include the size of the database and other software involved in the benchmark.

5 Conclusions and Future Plans

Since the modified version of *rcu-poll* was unable to match the performance of *rcu-ltimer*, and since there are

	Ins/Del/Delta			Total Lines New	Old	% Delta
msg.c	23	26	-3	885	888	-0.34%
sem.c	29	30	-1	1289	1290	-0.08%
shm.c	102	69	33	785	752	4.39%
util.c	178	13	165	581	416	39.66%
util.h	10	53	-43	64	107	-40.19%
Total	342	191	151	3604	3453	4.37%

Table 6: Semaphore Change in Lines of Code

currently no known uses of RCU that require low grace-period latency, *rcu-ltimer* is used in the Linux 2.5 kernel. If needed, we will investigate use of *rcu-poll*'s grace-period-latency mechanisms in *rcu-ltimer* after a fixed delay. A modified version of *rcu-sched* may attain even lower overheads than those of *rcu-ltimer*.

We will continue investigating how RCU may be used in the Linux kernel, including using it to provide NMI handler support and applying it to the `tasklist_lock` to eliminate some starvation scenarios that have been observed in the Linux 2.5 kernel. Experience gained from use of RCU in DYNIX/ptx$^{(R)}$, K42 [Gamsa99], and Linux indicate that it will continue to be quite helpful in obtaining dramatic performance improvements with little increase in complexity.

6 Acknowledgments

We owe thanks to Bill Hartner and Cliff White for running many benchmarks, to Rusty Russell, Anton Blanchard, Paul Mackerras, and the rest of the IBM OzLabs group for many valuable discussions, to Andrew Morton, Hugh Dickens, Davide Libenzi, and Rusty Russell again for their careful review of much code, and to Matt Dobson, the anonymous referees, and shepherd Ray Bryant for their substantial help in rendering this paper human-readable. We are indebted to Dan Frye, Hans Tannenberg, Chris Maher, Randy Kalmeta, Vijay Sukthankar, and Hugh Blemings for their support of this effort.

References

[Cao02] M. Cao *[PATCH] Latest IPC lock patch-2.5.44*, Linux Kernel Mailing List, October 2002. `http://marc.theaimsgroup.com/?l=linux-kernel\&m=103610704923787\&w=2`.

[Gamsa99] B. Gamsa, O. Kreiger, J. Appavoo, and M. Stumm. *Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system*, Proceedings of the 3rd Symposium on Operating System Design and Implementation, New Orleans, LA, February, 1999.

[Hartner02] B. Hartner. *semopbench 1.0.0*, IBM DeveloperWorks, October 2002. `http://www.ibm.com/developerworks/opensource/linuxperf/semopbench/semopbench.c`.

[Herlihy93] M. Herlihy. *Implementing Highly Concurrent Data Objects*, ACM Transactions on Programming Languages and Systems, vol. 15 #5, November 1993, pages 745-770.

[Linder02a] H. Linder, D. Sarma, and Maneesh Soni. *Scalability of the Directory Entry Cache*, Ottawa Linux Symposium, June 2002.

[LSE] D. Sarma et al. *Linux Scaling Effort (LSE)*, SourceForge Project, April 2002. `http://prdownloads.sourceforge.net/lse/`.

[McK98a] P. E. McKenney and J. D. Slingwine. *Read-copy update: using execution history to solve concurrency problems*, Parallel and Distributed Computing and Systems, October 1998. (revised version available at `http://www.rdrop.com/users/paulmck/rclockpdcsproof.pdf`).

[McK01b] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni. *Read-Copy Update*, Ottawa Linux Symposium, July 2001. (revised version available at `http://www.rdrop.com/users/paulmck/rclock/rclock\_OLS.2001.05.01c.pdf`).

[McK02a] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. *Read-Copy Update*, Ottawa Linux Symposium, July 2002. (revised version available at `http://www.rdrop.com/users/paulmck/rclock/rcu.2002.07.08.pdf`).

[Michael02a] M. M. Michael. *Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes*, In Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing, pages 21-30, July 2002.

[Michael02b] M. M. Michael. *High Performance Dynamic Lock-Free Hash Tables and List-Based Sets*, In Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architecture, pages 73,82, August 2002.

[Pugh90] W. Pugh. *Concurrent Maintenance of Skip Lists*, Department of Computer Science, University of Maryland, CS-TR-2222.1, June 1990.

[Sarma02] D. Sarma *[PATCH] Read-Copy Update 2.5.42*, Linux-Kernel Mailing List, October 2002. `http://marc.theaimsgroup.com/?l=linux-kernel\&m=103461974415359\&w=2`.

[Spraul01] M. Spraul *Re: RFC: patch to allow lock-free traversal of lists with insertion*, Linux-Kernel Mailing List, October 2001. `http://marc.theaimsgroup.com/?l=linux-kernel\&m=100264675012867\&w=2`.