

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

Paul E. McKenney
IBM Linux Technology Center
15400 SW Koll Parkway
Beaverton, OR 97006

Jonathan Walpole
Computer Science Department
Portland State University
walpole@cs.pdx.edu

Paul.McKenney@us.ibm.com
<http://www.rdrop.com/users/paulmck>

Draft of 2005/01/03 15:48

Abstract

The performance of synchronization instructions on shared memory multiprocessors (SMMP) has declined dramatically compared to the performance of simple instructions. As a result, operating system developers for SMMPs have sought out synchronization algorithms that avoid using these instructions, especially in commonly executed paths. One such algorithm that has been applied successfully in state-of-the-art SMMP operating systems is Read-Copy Update (RCU). RCU is a reader-writer synchronization algorithm that uses multi-version and deferred-destruction techniques to allow readers to execute in parallel with writers, while avoiding expensive synchronization instructions. For read-mostly scenarios, which occur frequently in operating systems in the guise of configuration and routing structures that track infrequently changing external state, this approach leads to significant performance and scalability improvements. However, successful use of RCU poses software engineering challenges because it can require reorganization of code and transformation of some existing algorithms.

In this paper we describe the conceptual basis for RCU, outline the work that contributed to its development, and detail how it can be implemented efficiently in an SMMP operating system kernel. We then analyse its performance in comparison with other synchronization mechanisms. To address the software engineering challenges posed by RCU, we introduce a set of transformational design patterns mined from various uses of RCU in production systems, including the Linux 2.6 kernel. Finally, we discuss the impact on RCU of potential future architectural trends and outline some of RCU's open research challenges.

1 Introduction

Advances in processor design have led to dramatic improvements in the performance of regular instructions. Among the architectural features that have contributed to these improvements are processor caches, multi-level cache hierarchies, multi-stage pipelines, and out-of-order execution. Unfortunately, these very features have also degraded the performance of synchronization instructions. The reason for this degradation is that, by definition, synchronization instructions must provide a consistent view of the corresponding in-memory data across the entire shared-memory multiprocessor (SMMP) system.

For SMMPs with per-CPU caches, achieving this consistent view involves checking the contents of each CPU's cache and potentially writing back or invalidating cache lines. These operations are very expensive compared to normal instructions. As the number of cache levels increases, so does the cost and complexity of this checking.

Out-of-order execution improves the performance of normal instructions by allowing stalls that would otherwise result from cache misses to be filled by unrelated instructions. However, synchronization instructions necessarily impose constraints on this reordering. For example, the instructions within a critical section can not be reordered to precede the acquisition of the lock that protects the critical section. These constraints require that synchronization instructions be treated differently than normal instructions, and prevent the CPU from filling in pipeline stalls that precede them.

Long pipelines improve performance of normal instructions by increasing the allowable clock frequency. However, they also take more clock cycles to fill, and therefore further degrade performance in the presence of stalls caused by synchronization instructions.

The effect of these architectural features on the performance of synchronization instructions relative to normal instructions has been dramatic. Table 1 shows that even by the late 1990s, the cost of synchronization instructions was already 2-3 orders of magnitude higher than that of regular instructions. To show that these costs are not peculiar to the P-III, Figure 1 shows the trend for critical section “efficiency” on Sequent Computers over the past 20 years. Critical section efficiency relates the cost of synchronization, e.g., the acquisition and release of a lock, to the cost of normal instructions in the body of a critical section, and is defined as follows:

$$e = \frac{T_c}{T_c + T_s} \quad (1)$$

where T_c is the cost of the instructions in the body of the critical section and T_s is the synchronization cost. Since the value of T_c varies depending on the workload, Figure 1 plots efficiencies for T_c of 10, 100 and 1000 instruction critical sections. Similar effects have been observed on other machines [25], and the portion of the plot from 1996 onwards is typical of Intel-based systems during that time period.

Table 1: 700 MHz P-III Operation Costs

Operation	Cost (ns)
Instruction	0.7
Clock Cycle	1.4
Atomic Increment	58.2
Main Memory	162.4
cmpxchg Blind Cache Transfer	170.4
cmpxchg Cache Transfer and Invalidate	360.9

The high cost of synchronization instructions has not gone unnoticed by SMMP operating system designers. Historically, efforts to improve scalability focused on reducing lock contention. For example, large critical sections were broken into numerous smaller critical sections, and read-mostly scenarios were handled using asymmetric synchronization algorithms such as reader-writer locking. However, with the high cost of synchronization instructions, these techniques have been shown to exacerbate performance and scalability problems because they can increase the number of lock-acquisition operations, causing substantial overhead even in the absence of lock-contention.

To illustrate the scalability behavior of various commonly-used asymmetric synchronization approaches, we evaluated their performance on a hash table search benchmark running on a 4-CPU 700MHz Pentium-III system. The results are presented in Figure 2. The graph on the left is for a small 32-element hash-table that fits entirely in the CPU’s on-chip cache. The graph on the right is for a larger 16,384 element hash-table that does not fit in cache.

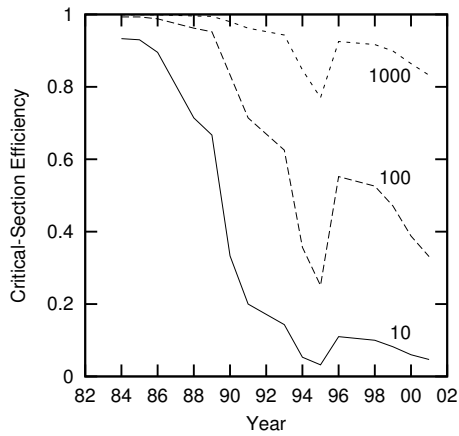


Figure 1: Critical-Section Efficiency for Sequent Computers

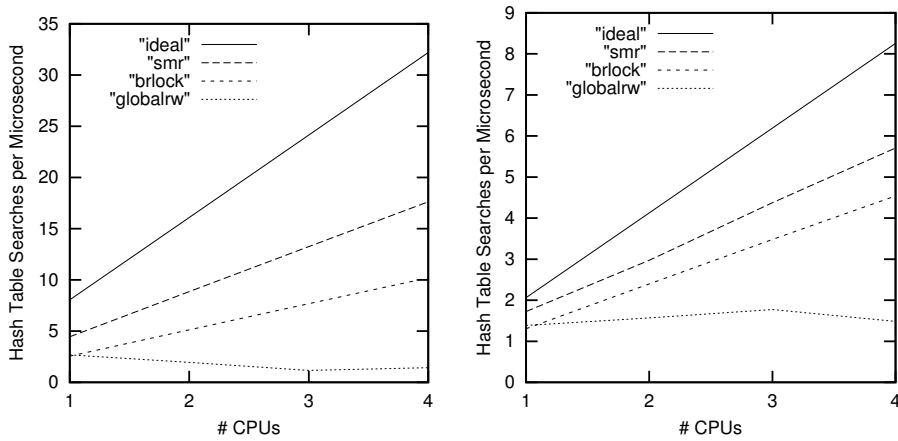


Figure 2: Scalability of Selected Synchronization Algorithms (Small and Large Table)

We evaluated three algorithms and compared them to the ideal case, which is a single-CPU hash-table search with no synchronization, scaled up linearly with the number of CPUs. The three algorithms evaluated were:

1. global reader-writer locking, in which readers and writers must acquire a single global reader-writer lock that allows readers to proceed concurrently with other readers, but ensures that writers get exclusive access.
2. per-CPU reader-writer locking, in which each CPU is given its own lock, with readers acquiring only their own CPU's lock, but writers acquiring all CPUs' locks [15, 22, 32, 49]. Note that this algorithm is called "brlock" in Linux.
3. safe memory reclamation, in which "hazard pointers" are maintained by each CPU to inform writers of which elements are currently being referenced [13, 14, 30, 31].

Figure 2 shows that global reader-writer locking has the worst performance, with negative scalability even on two processors in the small hash table case, and minimal scalability in the large

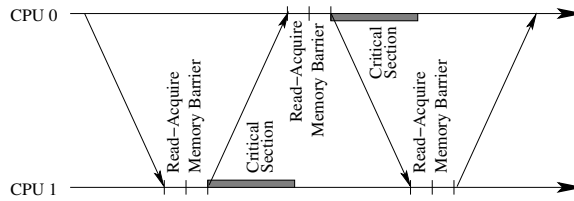


Figure 3: Cache Thrashing During Reader-Writer Lock Acquisition

hash table case. The reason for this poor scalability is that all CPUs must manipulate a single data item, namely the reader-writer lock. Hence, they spend most of their time waiting for this item to be shuttled among their caches. For a moderately sized critical section, the cost of moving the item from one cache to another rivals that of the critical section itself, and when combined with memory barrier overhead, exceeds it. Figure 3 illustrates this behavior on a pair of CPUs. Note that since the cost of lock acquisition exceeds the cost of the critical section itself, global reader-writer locking does not actually allow readers to proceed in parallel.

Figure 2 shows that per-CPU reader-writer locking (brlock) improves performance compared to global reader-writer locking, since it no longer requires readers to shuttle a lock among the CPU caches. Although brlock attains linear scalability, its performance is still poor compared to the ideal case. The reason for the degradation is because the cost of acquiring a local lock is still high, for the reasons discussed earlier.

The safe memory reclamation approach improves performance further because it avoids read-side use of locks by replacing them with per-CPU hazard pointers. Under a read-only workload, these hazard pointers are updated locally on each CPU, and never referenced from another CPU. Hence, cache locality is excellent. However, each update to a hazard pointer requires a memory-barrier operation before the element is referenced. This memory-barrier operation is the cause of the performance degradation compared to the ideal.

The performance and scalability problems illustrated in the above examples have led operating systems researchers to search for synchronization approaches that do not require *any* synchronization operations in the commonly executed read paths. For an example of such an approach consider the following strategy for atomically inserting an element B into a linked list that contains elements A and C (see Figure 4).

1. Allocate element B and set its successor pointer to point to element C.
2. Update element A's successor pointer to point to element B.

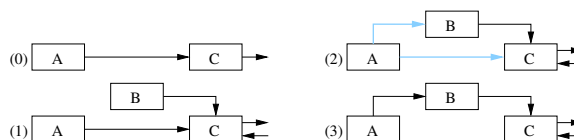


Figure 4: Atomic Insertion Into a Linked List

On an architecture with a sequentially consistent memory model, this strategy is sufficient for making the insertion atomic with respect to readers, even though readers do not execute any synchronization operations. Atomicity is enabled because the update of element A's successor pointer, assuming it is properly aligned, is a single word write, which is atomic on all CPUs in common use today. Prior to this update readers are unable to reference element B, but all list traversals starting after this update will see element B. Of course, writers must still synchronize with each other, but in read-mostly cases this writer-writer synchronization is rare and hence does not impact

performance or scalability significantly. On architectures with weaker memory consistency models, writers must execute a memory-barrier. However in most cases, with the exception of the DEC Alpha [24], readers do not need to execute a memory barrier, and hence proceed at full speed.

The linked-list insertion example demonstrates the benefit of hiding a set of complex operations behind a single atomic “commit point” in order to eliminate the need for readers to synchronize with writers. This approach can be applied more generally, but its application can be non-trivial. For example, consider the case of linked-list removal, shown in Figure 5. An element B can be removed atomically from a linked list containing elements A,B and C, by simply updating element A’s successor pointer to point to element C. Since the update of the pointer is atomic, any traversal of the list prior to the update will see element B and any traversal commencing after it will not see element B. However, this algorithm has the side effect of allowing a reader already having a reference to element B to continue using it even after element B has been removed from the list. In other words, this algorithm allows readers to see stale data.

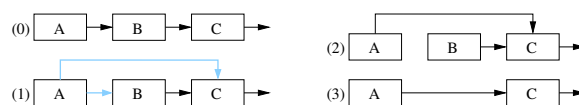


Figure 5: Atomic Deletion From a Linked List

In some scenarios stale data can be tolerated. For example, in TCP/IP routing tables packets are retransmitted in the event of an error, and the routing protocols have built-in delays, on the order of seconds, to prevent route thrashing, so small additional delays are insignificant. In other cases, however, stale data may not be tolerated, so it is important to understand where the algorithm can be applied, and to understand the transformation pattern language presented in Section 4, which can transform many algorithms into a form that *can* tolerate stale data.

A separate problem introduced by the linked-list removal example is knowing when it is safe to reclaim the memory of the removed element. It is necessary to reclaim the memory in order to avoid memory leaks, but the memory can only be reclaimed once the last reader holding a reference to the element has finished using it. This is basically a garbage collection problem. One way to signal the reader-completion event to the garbage collector would be to maintain reference counts, reclaiming an element’s memory when the corresponding count reaches zero. However, if maintaining the consistency of this reference count requires readers to use synchronization instructions, the algorithm would no longer be free of synchronization instructions in the commonly executed path, and hence would neither perform nor scale well. So the challenge is to find a way for the garbage collector to discover when readers complete, but without forcing readers to use synchronization instructions.

The solution to this challenge is derived from the observation that communication between readers and a garbage collector is not the same problem as synchronization between readers and writers. First, the memory associated with an unlinked element need not be reclaimed immediately. As long as the system does not run out of memory, reclamation can be delayed and correctness is unaffected. Second, by delaying reclamation, multiple elements’ memory can be reclaimed at the same time. Third, because of the ability to delay and batch memory reclamation, a single signal can be used to communicate the completion of several readers to the garbage collector. The challenge, now, is how to implement this “signal”.

Fortunately, in operating systems, it is possible to determine indirectly when references are no longer in use. To illustrate a possible approach, consider the analogous case of coding conventions for locking. In non-preemptive operating systems, it is common practice to prohibit yielding the CPU while holding a spinlock. The purpose of this coding convention is to avoid deadlocks in which all CPUs are consumed spinning on a lock held by a thread that is not running. Importantly

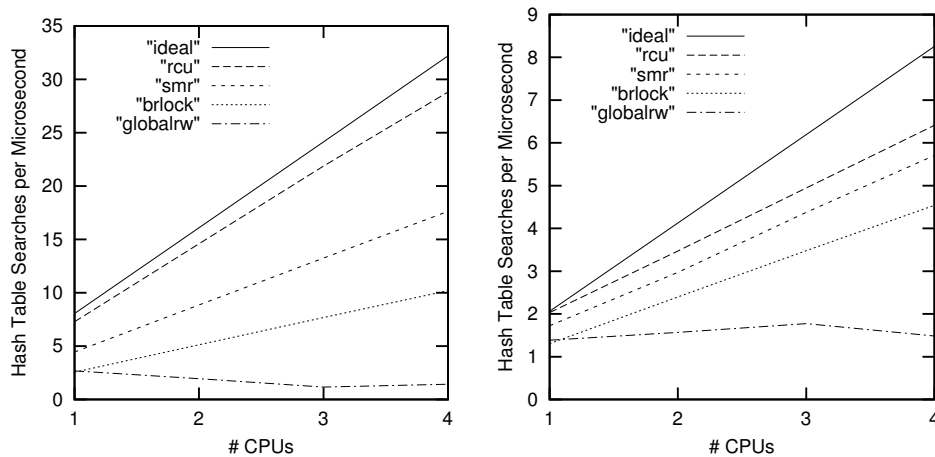


Figure 6: Read-Mostly Linked-List Deletion Scalability (Small and Large Table)

for the purposes of this discussion though, it is also possible to observe that once a context switch has occurred on every CPU, none of the locks that were held prior to the first of these context switches can still be held after the last context switch. This observation is relevant because a similar coding convention can be used to determine whether references are still held. By following the convention of not yielding the CPU while holding a reference to a list element, it is possible to determine when an unlinked element is no longer referenced, simply by observing context switches. Since context switches occur anyway, there is minimal cost associated with this technique. In effect, the completion “signal” for a whole generation of readers has been piggy-backed on the naturally occurring context switch events.

The resulting algorithm for deleting an element B from a linked-list that initially contains elements A, B, and C, and reclaiming element B’s memory, is as follows.

1. Update element A’s successor pointer to point to element C.
2. Wait until a context switch has been observed on every CPU.
3. Return element B’s memory to the free list.

This algorithm is a special case of Read-Copy Update (RCU). Figure 6 shows that the performance of this algorithm (labeled ‘rcu’) for read-only work-loads is fairly close to ideal for large and small data sets.

So far, the algorithm has been described only for non-preemptible environments. However, it can also be used in preemptible environments, for example by disabling preemption during list traversal. Alternative implementation approaches are described elsewhere [4, 9, 27, 29, 24].

The examples so far have also been restricted to pure insertion or deletion. To extend the approach to more complex, multi-word updates it is necessary to introduce one final concept – copying to produce multiple versions, instead of updating in place. To illustrate this idea, consider the task of atomically updating multiple independent fields of element B in a linked list containing elements A, B and C. This task can be accomplished by the following steps, illustrated in Figure 7:

1. Allocate a new element, denoted by the empty box in Figure 7.
2. Copy all fields from element B to the new element, then perform the desired updates to the fields of the new element, denoted B’ in Figure 7.
3. Update the successor pointer of element A to point to element B’.
4. Wait until a context switch has been observed on every CPU.
5. Free the old element B.

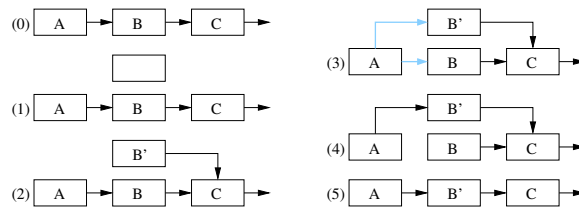


Figure 7: Atomic Update to a Linked List

Note that until the new element B' is linked into the list it is unreachable and hence can be updated in place without the need for synchronization. This algorithm is another example of RCU, and in fact, led to the name "RCU" since "readers" run concurrently with a "copy" operation that is used to "update" the list.

In summary then, RCU is a combination of techniques including copying to create new versions that are updated in place before being atomically linked, and the piggy-backing of read completion signals on naturally occurring system events, such as context switches, in order to enable deferred destruction of unreferenced versions. The result is that on most architectures readers can execute without using expensive synchronization instructions, and hence achieve near optimal performance and scalability for read-mostly scenarios. The overhead of multi-version creation, deferred destruction, and writer-writer synchronization is borne by writers, which execute only rarely in the read-mostly scenarios that are the focus of RCU.

In the following section we present a more precise description of the conceptual basis for RCU. Then Section 3 describes how RCU is implemented in several production operating systems, and outlines some of its applications in those systems. An analysis of RCU's performance and scalability characteristics, compared to locking, is presented in Section 4. Section 5 describes the software engineering challenges posed by RCU, and presents a set of transformational design patterns to help guide its application. Performance results are presented in Section 6, and related research is discussed in Section 7, as are the implications of various ongoing architectural trends on RCU performance and applicability. Finally, Section 8 concludes the paper and outlines some future research directions.

@@@ Paul - It occurs to me that we are using the terminology "reclamation" and "destruction" interchangeably. We need to tidy that up at some point.

"reclamation" is a special case of "destruction". Another special case would be a change of state that relied on all executing NMIs to be using the new NMI vector.

2 RCU Overview

The previous section gave examples of the RCU technique. Although such examples can be valuable, they are no substitute for an abstract conceptual basis for RCU, which this section provides.

Figure 8 shows a schematic of last section's linked-list deletion example. This schematic shows the actions of a pair of CPUs, with time proceeding from left to right. CPU 1 removes an element from the linked list, however, CPU 0 might concurrently be referencing it at point A, so the element cannot be immediately freed. At the two points labeled "B", there can be no read-side references to the deleted element, but RCU is not aware of this, since the required context switches have not yet occurred. At the point labeled "C", RCU would be aware that CPU 1 holds no read-side reference to the element, but this is not sufficient to free the element, since CPU 0 has not yet executed a context switch. Only at the points labeled "D" can RCU be sure that there are no read-side references to the deleted element, since by that time both CPUs have executed a context switch, thereby signalling that they have completed any read-side critical sections that were in progress at the time that the element was deleted.

A CPU is an example of a locus of control, or a "thread", which also includes processes,

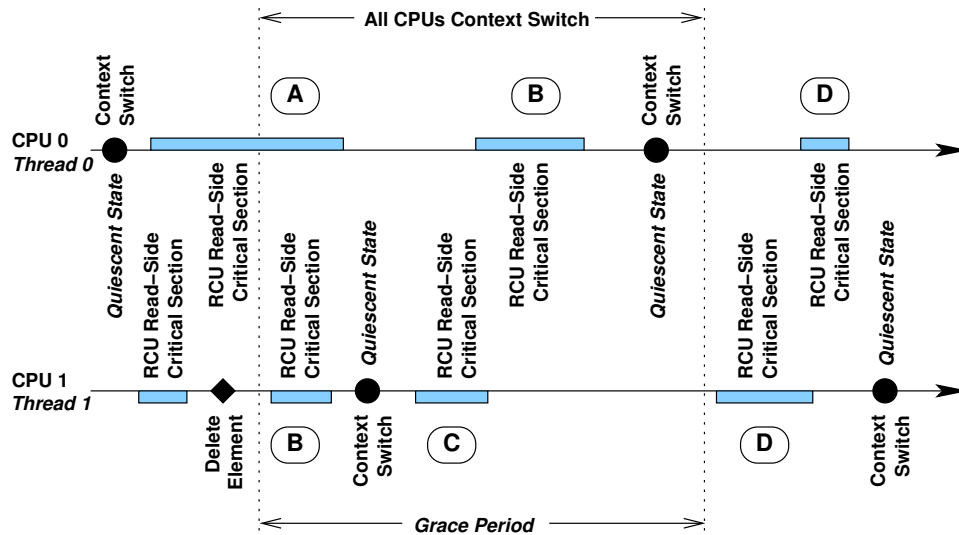


Figure 8: RCU Concepts

tasks, coroutines, and interrupt handlers. A context switch is an example of a “quiescent state”, which is a state in which a given thread is known to have completed any preceding RCU read-side critical sections. Similarly, any period of time in which all threads each pass through at least one quiescent state is a “grace period”. Figure 8 is labeled with both the abstract and concrete terms, with the abstract terms in *italics*.

Since any period of time during which all threads each pass through at least one quiescent state is a grace period, there can be multiple overlapping grace periods, as shown in Figure 9. In addition, any period of time that fully encloses a grace period is itself a grace period.

Careful choice of quiescent states is critical to an efficient and useful RCU implementation. In principle, *any* state that is guaranteed never to occur within an RCU read-side critical section can be considered a candidate quiescent state. From this infinite number of candidate quiescent states, the *observed* quiescent states must be chosen with the following considerations in mind:

1. Frequency. Too-frequent quiescent states impact performance and scalability, while too-infrequent quiescent states can result in excessive amounts of memory tied up waiting for the grace period to elapse. Implementations often use multiple quiescent states in order to attain the necessary balance between performance and memory.
2. Observability. Quiescent states must be visible to RCU in order to determine the end of the corresponding grace period. This visibility must not increase the quiescent state’s overhead significantly, preferably, not at all.
3. Ease of use. Since quiescent states must never appear in RCU read-side critical sections, it must be natural to code critical sections without using such states.
4. Error detection. Since placing a quiescent state in an RCU read-side critical section can result in difficult-to-debug memory-corruption bugs, it must be easy to detect and flag such errors, preferably automatically.

Different implementations of RCU can and do select different quiescent states:

1. VM/XA chose special “process checkpoints” as the quiescent states [11]. This variant of RCU has seen mission-critical production use on IBM mainframes.
2. DYNIX/ptx chose context switch, user-mode execution, idle loop, system-call initiation, and trap from user mode as the quiescent states [29]. This variant of RCU was designed and

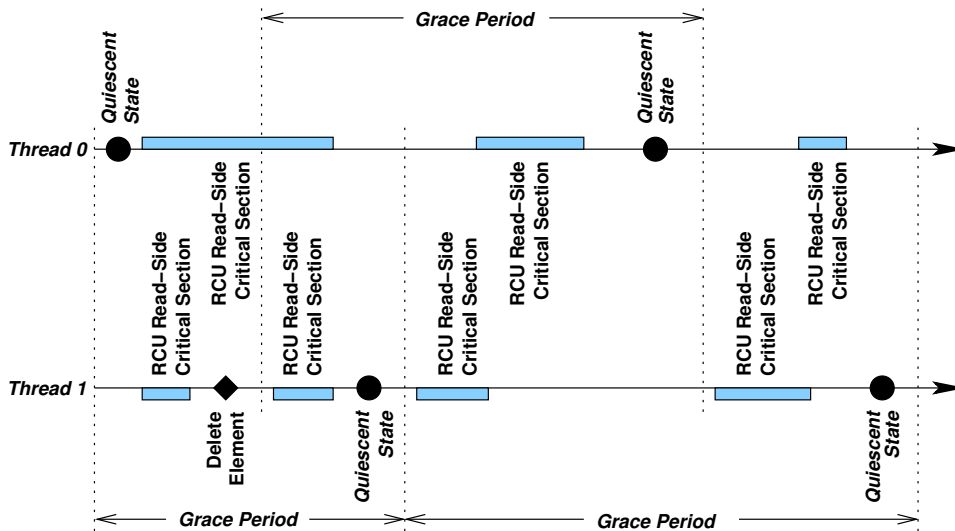


Figure 9: RCU Grace Periods

implemented by one of the authors (Paul), and has seen datacenter production use: in 1999, seven of the ten largest Oracle databases ran on a kernel using this variant of RCU.

3. K42 and Tornado chose thread termination and voluntary context switch as the quiescent states [9]. Since these operating systems featured kernel preemption, involuntary context switch was *not* a quiescent state.
4. Linux features two different sets of quiescent states: (a) context switch, user-mode execution, and idle loop for normal “process-context” RCU, and (b) exit from non-nested interrupt handler for “bottom half” RCU [24, 36]. This second implementation is used by networking protocols that can be subject to denial-of-service attacks.

It is therefore quite important to be able to discuss RCU independently of any particular implementation.

RCU is a reader-writer synchronization mechanism that takes asymmetric distribution of synchronization overhead to its logical extreme: read-side critical sections incur zero synchronization overhead, containing no locks, no atomic instructions, and, on most architectures, no memory-barrier instructions. RCU therefore achieves near-ideal performance for read-only workloads on most architectures, and can greatly simplify the structure of some algorithms [9]. Write-side critical sections must therefore incur substantial synchronization overhead, deferring destruction and maintaining multiple versions of data structures in order to accommodate the read-side critical sections. In addition, writers must use some synchronization mechanism, such as locking, to handle concurrent updates.

Readers must provide a signal enabling writers to determine when it is safe to complete destructive operations, but this signal may be deferred, permitting a single signal operation to serve multiple read-side RCU critical sections. RCU typically signals writers by non-atomically incrementing a local counter, which is an extremely inexpensive operation.

These read-side signals are observed by a specialized distributed garbage collector, which uses a lazy barrier or a combining tree to sense the reader signals, and carries out destructive operations once all readers have signalled that it is safe to do so. Garbage collectors are typically implemented in a manner similar to a barrier computation, or, on NUMA systems, a combining tree. Production-quality garbage collectors batch destructive operations, so as to amortize their overhead over many write-side update operations.

RCU provides concurrent reads: because readers do not use any synchronization mechanism, there is no way for readers to avoid concurrency. For the same reason, RCU provides concurrent reads and writes. RCU does not specify whether writers may run concurrently with each other; write-side concurrency depends instead on the chosen write-side synchronization mechanism. The fact that RCU read-side critical sections use no synchronization mechanisms means that there is no overhead due to pipeline stalls, memory latency, contention, or locking for readers. Write-side overhead depends on the chosen write-side synchronization mechanism, but contention is reduced due to the fact that readers do not use any synchronization mechanisms.

In practice, it is also important that the read-side signalling occur reasonably frequently, since during a time interval in which a given CPU fails to provide such a signal, pending destructive operations cannot be carried out. If this condition persists, all available memory could be consumed tracking these pending destructive operations.

It might seem that RCU is the antithesis of mutual exclusion, given that writers cannot exclude readers. However, RCU does still provide mutual exclusion in the following sense: RCU guarantees that readers are excluded from any data element removed from its data structure at least one grace period ago. It is exactly this mutual-exclusion guarantee that allows such data elements to be safely freed.

Given this conceptual basis for RCU, we are now in a position to present a high-level description of RCU implementations.

3 Implementing RCU

Variants of RCU have been implemented in DYNIX/ptx [29, 41], SuSE 7.3 Update [4], Linux 2.6 [50], K42/Tornado [9], and VM/XA [11], and three of these five implementations represent independent reinventions of RCU. Section 3.1 describes the Linux 2.6 implementation of the RCU infrastructure and Section 3.2 summarizes VM/XA, Dynix/PTX, K42, and Linux 2.4 kernel uses of RCU. A description of the Linux 2.6 kernel's RCU API may be found elsewhere [51, 24, 23], and Linux 2.6 uses of RCU are presented in Section 5.

3.1 Implementing Grace-Period Detection

The following sections summarize several different types of `call_rcu()` implementation. The two basic SMMP approaches are (1) inducing quiescent states, which is addressed elsewhere [26, 24] and (2) observing naturally occurring quiescent states, which is covered in the remainder of this section. Special considerations for uniprocessors are covered elsewhere [24]; the key point is that SMMP RCU implementations work as one would expect in uniprocessor systems.

3.1.1 Observing Naturally Occurring Quiescent States

When observing naturally occurring quiescent states, it is important to avoid significantly increasing their overhead. It is also important to determine efficiently, from the observations of quiescent states, when a given grace period has ended. This is achieved via a design that runs the common-case code paths on a per-CPU basis, so that these code paths do not incur any SMMP overhead. The overall architecture of the Linux 2.6.9 RCU infrastructure is shown in Figure 10, with the arrows indicating information flow among the subsystems. Quiescent states are detected on a per-CPU basis, using per-CPU data, represented by the boxes in the upper left of the figure. Similarly, batches of RCU callbacks are processed on a per-CPU basis using per-CPU list, represented by the boxes in the lower right of the figure. Grace periods are detected globally, based on the per-CPU quiescent-state detection, represented by the box in the upper right of the figure. As each quiescent state is detected, a global batch number is incremented, represented by the box in the lower left of the figure. This global batch number is used by each CPU's batch control to determine when a given batch of RCU callbacks may safely be invoked. It is also used by each CPU's quiescent-state detection to determine when to restart its detection of quiescent states.

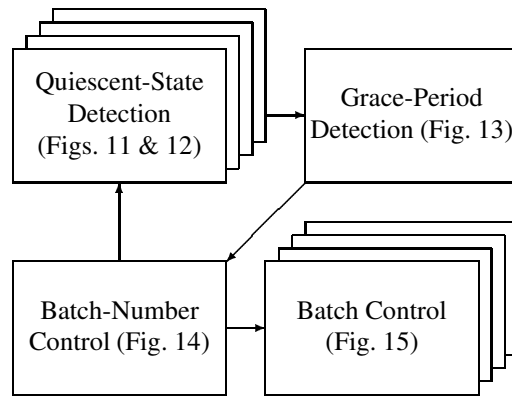


Figure 10: Linux 2.6.9 RCU Infrastructure

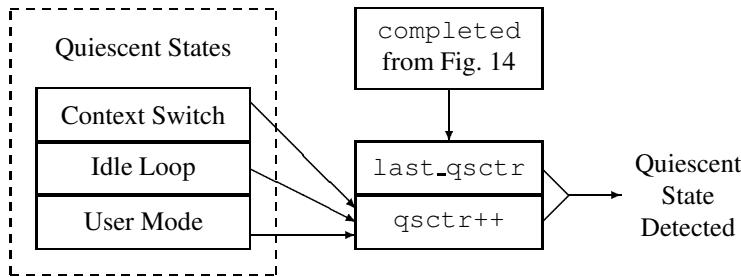


Figure 11: RCU Quiescent States

Figures 11 and 12 show the detection of the two sets of quiescent states tracked by the RCU infrastructure in the Linux 2.6.9 kernel. The quiescent states in Figure 11 are chosen for RCU use throughout the kernel in situations not subject to high-update-rate denial-of-service attacks. This set of quiescent states and corresponding grace periods are detected by the `call_rcu()` primitive.

The quiescent states in Figure 12 are chosen for algorithms executing entirely in Linux's bottom-half context which *are* subject to denial-of-service attacks, for example, as part of networking protocols such as TCP/IP. This set of quiescent states and corresponding grace periods are detected by the `call_rcu_bh()` primitive.

In either case, each CPU maintains a pair of counters, `qsctr` and `last_qsctr`. When the CPU detects that the global batch number `completed` has changed, it takes a snapshot of its `qsctr` counter into its `last_qsctr` counter. Since `qsctr` is incremented each time that the CPU passes through a quiescent state, the CPU can later compare the values of these two counters to determine whether it has passed through a quiescent state. This approach optimizes for the common case where CPUs pass through quiescent states much more frequently than they check for having done so.

Figure 13 shows how the RCU infrastructure determines when a grace period has elapsed, using a straightforward barrier computation. In the Linux 2.6 kernel, this is implemented as a bitmask, with one bit per CPU. When a given CPU determines that it has passed through a quiescent state, it clears its bit. If this results in all bits being zero, the end of a grace period has been detected. Large-scale systems are better served by a combining tree than by a simple bitmask, and Manfred Spraul has created such a patch to the Linux kernel [47].

Figure 14 shows batch-number control, which simply increments the global `completed`

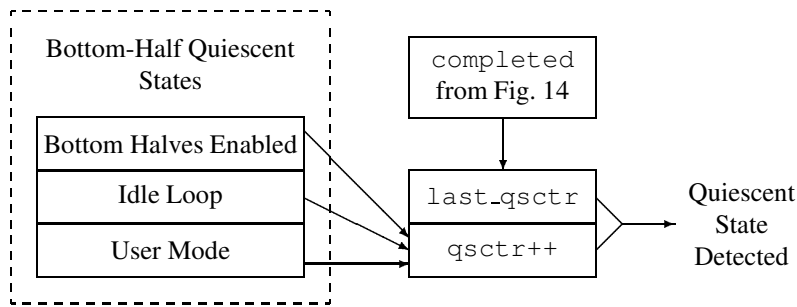


Figure 12: Bottom-Half RCU Quiescent States

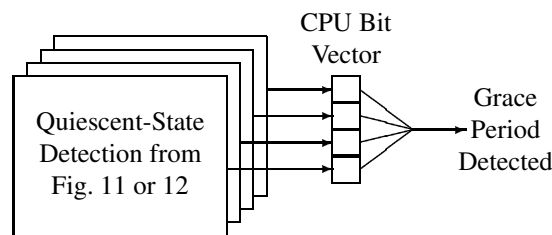


Figure 13: Grace-Period Detection

counter at the end of each grace period. Token-based grace-period-detection schemes [9] can maintain the batch number as an integral part of grace-period detection [27], however, more work is needed to address their long grace-period latencies..

Figure 15 shows how each CPU controls the batches of RCU callbacks that it registers via `call_rcu()` or `call_rcu_bh()`. These callbacks are sequenced through three per-CPU lists, `nextlist`, `curlist`, and `donelist`, in that order. The callbacks in `nextlist` are waiting for their grace period to start, those in `curlist` are waiting for their grace period to end, and those in `donelist` are waiting to be invoked, as their grace period has ended. Each CPU maintains the batch number corresponding to the callbacks in `curlist` in a per-CPU variable `batch`. Once the global `completed` counter becomes equal to `batch`, the end of the grace period corresponding to the callbacks in `curlist` has ended, and so those callbacks may be appended to `donelist`. Any time `nextlist` is non-empty and `curlist` is empty, the callbacks in `nextlist` may be moved to `curlist`, and the value of `batch` set to be either one or two greater than that of `completed`, depending on whether a grace period is currently in flight.

These figures give a conceptual view of a production RCU implementation; the actual code is publicly available [51]. This code handles a number of issues not described in this conceptual view, including the need to add and remove CPUs without reboot, optimizing layout of data structures to avoid cache misses, and the need to indirectly detect the idle-loop and user-mode quiescent states. The conceptual view presented in this section is nonetheless valuable as an overview of the RCU implementation.

3.2 RCU Usage

The following sections briefly overview use of RCU in VM/XA, DYNIX/ptx, K42/Tornado, SuSE Linux, and the Linux 2.4 kernel. Use of RCU in the Linux 2.6 kernel is summarized by Table 9 in Section 5.

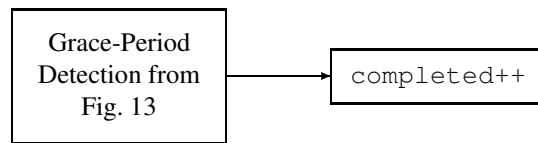


Figure 14: Batch-Number Control

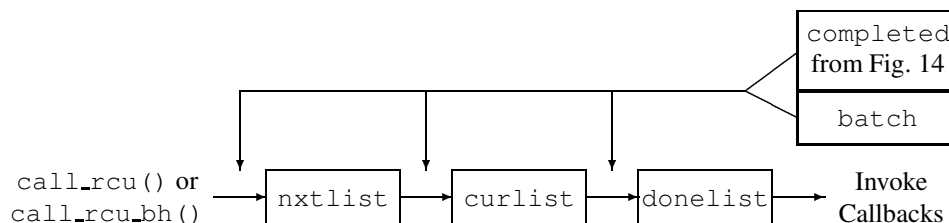


Figure 15: Batch Control

3.2.1 VM/XA

A mechanism resembling RCU [11] is used for per-user-ID tracing in IBM's VM/XA mainframe product [40].

3.2.2 DYNIX/ptx

The DYNIX/ptx development methodology was often extremely qualitative, and multiple improvements were often made simultaneously, as required by competitive pressures and the relatively small size of the DYNIX/ptx development team. There are therefore no accurate measures of the benefits of the RCU implementation for DYNIX/ptx. That said, DYNIX/ptx uses RCU for the following purposes [26]:

1. Distributed lock manager: recovery, lists of callbacks used to report completions and error conditions to user processes, and lists of server and client lock data structures. RCU reduced the complexity of the locking hierarchy, thereby greatly simplifying the deadlock-avoidance code. This subsystem inspired RCU.
2. TCP/IP: routing tables, interface tables, and protocol-control-block lists. This project was used as a testbed to architect DYNIX/ptx's RCU API.
3. Storage-area network (SAN): routing tables and error-injection tables (used for stress testing).
4. Clustered journaling file system: in-core inode lists and distributed-locking data structures.
5. Lock-contention measurement: B* tree used to map from spinlock addresses to the corresponding measurement data (since the spinlocks are only one byte in size, it is not possible to maintain a pointer to the corresponding measurement data within each spinlock).
6. Application regions manager (which is a workload-management subsystem): maintains lists of regions into which processes may be confined.
7. Process management: per-process system-call tables as well as the multi-processor trace data structures used to support user-level debugging of multi-threaded processes.
8. LAN drivers: resolve races between shutting down a LAN device and packets being received by that device. This change applied the Pure RCU pattern.

More information on the implementation and use of RCU in DYNIX/ptx may be found elsewhere [29, 41, 42, 43, 44].

3.2.3 RCU Use in K42

The Tornado and K42 [9] research operating systems independently developed a form of RCU, which is used as follows:

1. To provide existence guarantees throughout these operating systems. These existence guarantees simplify handling of races between use of a data structure and its deletion in fine-grained locking schemes [3, 9].
2. To identify quiescent states so that implementations of an object can be swapped on the fly while the object is in active use [46]. Note that hot swapping requires that the underlying RCU infrastructure have low grace-period latency.
3. To support a non-blocking hash-table implementation which is used throughout K42 [24].

Use of RCU to provide existence locks is a basic architectural tenet of Tornado and K42, where it reduces lock and memory contention and simplifies locking designs.

3.2.4 RCU in the Linux 2.4 Kernel

Numerous RCU patches were produced for the Linux 2.4 kernel [26, 27]. In addition, many uses of RCU later accepted into the Linux 2.6 kernel were first prototyped in 2.4 kernels.

RCU was also put into production in the 2.4 kernel. SuSE 7.3 Update and later releases include an implementation of RCU that resembles that of VM/XA. This was the first shipping version of RCU in Linux, which is used to reduce the probability of destructive races that can occur in Linux 2.4 kernels during module unloading. Similar code appears in the Linux 2.6.0-test1 kernel.

3.3 Discussion

This section has presented the Linux 2.6 kernel RCU infrastructure. This section examines some possible changes.

Although the RCU API has proven quite serviceable in the Linux 2.6 kernel, it is likely to continue evolving for the following reasons:

- Some filesystems may have unmount processing that requires all outstanding RCU callbacks for that mount point to complete. A suitable primitive can be easily provided, but there are interesting interactions with CPU hotplug, making this an important area of future work.
- The Linux 2.6 kernel RCU API has improved greatly over the past year, but its ease of use could be improved. One key difficulty is identifying which update corresponds to a given RCU read-side critical section. Ingo Molnar has proposed a change to the API that tags the read-side critical sections with the corresponding update-side lock or semaphore@@@ citation @@@, and it is possible that additional update-side annotations will be valuable.
- The Linux 2.6 kernel RCU implementation is much more friendly to real-time applications than in the past, but more work will likely be required to meet more-aggressive 10-microsecond scheduling latencies, to say nothing of hard-realtime requirements. @@@ cite Ingo and self (or maybe Jon Corbet) @@@

Evaluation of these changes is ongoing.

4 Design Patterns for RCU

Since RCU is not intended to replace all existing synchronization mechanisms, it is necessary to know when and how to use it. From a performance standpoint, it is clear that RCU is best suited for read-mostly data structures, and the relevant performance tradeoffs are discussed at length in Section 6. However, there are also software-engineering implications surrounding the use of RCU, and it is these implications that are addressed by this section via design patterns.

Coplien and Schmidt [5] define “design pattern” as follows; similar definitions may be found elsewhere [2, 8]:

Design patterns capture the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context. Because they address fundamental challenges in software system development, design patterns are an important technique for improving the quality of software. Key challenges addressed by design patterns include communication of architectural knowledge among developers, accommodating a new design paradigm or architectural style, and avoiding development traps and pitfalls that are usually learned only by (painful) experience.

RCU certainly qualifies as a new design paradigm, so it is reasonable to expect RCU-related design patterns. Such patterns do in fact exist; ten of them are presented in this section.

This section presents two types of RCU-related patterns. The first type is the RCU design pattern, presented in Section 4.1, which describes situations in which RCU may easily be applied. The second type is the RCU transformational pattern, presented in Section 4.2, which describes how to transform algorithms not directly amenable to RCU into forms to which RCU may easily be applied.

4.1 RCU Design Patterns

This section presents four RCU design patterns as part of a larger locking-design pattern language [21]. The patterns in this larger language are as follows:

1. Sequential Program (4.1.1)
2. Code Locking (4.1.2)
3. Data Locking (4.1.3)
4. Data Ownership (4.1.4)
5. Parallel Fastpath (4.1.5)
6. Reader/Writer Locking (4.1.6)
7. RCU, which includes these subpatterns:
 - (a) **Pure RCU** (4.1.7)
 - (b) **RCU Existence Locks** (4.1.9)
 - (c) **Reader-Writer-Lock/RCU Analogy** (4.1.8)
 - (d) **RCU Readers With NBS Writers** (4.1.10)
8. Hierarchical Locking (4.1.11)
9. Allocator Caches (4.1.12)
10. Critical-Section Fusing (4.1.13)
11. Critical-Section Partitioning (4.1.14)

These patterns are briefly summarized in this section; more details are available elsewhere [21, 24].

Relationships among these patterns are shown in Figure 16 and are described in the following paragraphs.

Parallel Fastpath and RCU are meta-patterns, that is, they are patterns that describe groups of lower-level patterns. Critical-Section Fusing and Critical-Section Partitioning are transformational patterns. For example, partitioning a code-locked program’s critical sections over instances of an object transforms that program’s lock design from code locking to data locking, and vice versa. Similarly, fusing *all* a code-locked program’s critical sections results in a sequential program, and vice versa. A data-locking design may be transformed into a data-ownership design by specifying owning threads (or CPUs) for each data item and coding any needed messaging.

Reader/Writer Locking, RCU, Hierarchical Locking, and Allocator Caches are instances of the Parallel Fastpath meta-pattern. Reader/Writer Locking, RCU, and Hierarchical Locking are themselves meta-patterns; they can be thought of as modifiers to the Code Locking and Data Locking

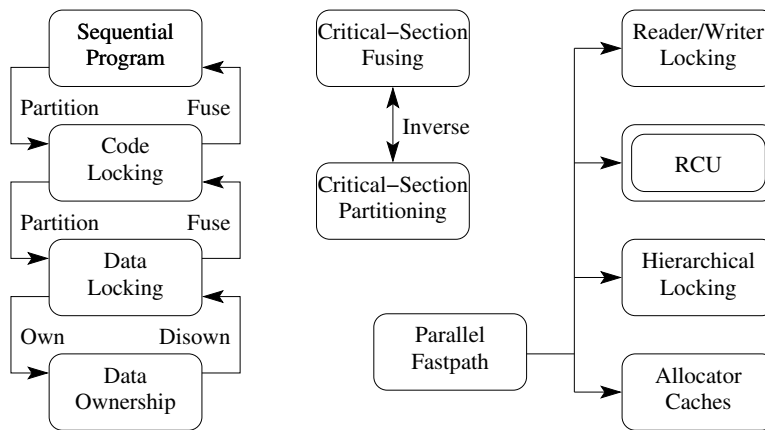


Figure 16: Relationship Among Patterns

patterns. Parallel Fastpath, Hierarchical Locking, and Allocator Caches are ways of combining other patterns and thus are template patterns.

Critical-Section Partitioning transforms Sequential Program into Code Locking and Code Locking into Data Locking. It also transforms conservative Code Locking and Data Locking into more aggressively parallel forms.

Critical-Section Fusing transforms Data Locking into Code Locking and Code Locking into Sequential Program. It also transforms aggressive Code Locking and Data Locking into more conservative forms.

Assigning a particular CPU or process to each partition of a data-locked data structure results in Data Ownership. A similar assignment of a particular CPU, process, or computer system to each critical section of a code-locked program results in Client/Server, which is used heavily in distributed systems but not discussed further in this paper.

Note that it is not appropriate to make absolute value judgements about any of these patterns. For example Sequential Program does not scale at all, but is the pattern of choice for workloads that a single CPU is capable of completing due to its simplicity.

The following section describes the forces that act on locking design patterns, and the ones after that describe the patterns themselves.

The following sections briefly describe the locking design pattern language, including the RCU-related patterns. Readers desiring more detail may consult this previously published locking design pattern language [21, 24].

4.1.1 Sequential Program

If the program runs fast enough on a single processor, and has no interactions with other processes, threads, or interrupt handlers, you should remove the synchronization primitives and spare yourself their overhead and complexity. Some would argue that Moore's Law will eventually force all programs into this category. Others would disagree.

4.1.2 Code Locking

Code locking is the simplest locking design, using only global locks. It is especially easy to retrofit an existing program to use code locking in order to run it on a multiprocessor. If the program has only a single shared resource, code locking will even give optimal performance. However, most programs of any size and complexity require much of the execution to occur in critical sections, which in turn sharply limits the scaling.

Therefore, use code locking on programs that spend only a small fraction of their run time in critical sections or from which only modest scaling is required. In these cases, code locking will

provide a relatively simple program that is very similar to its sequential counterpart.

4.1.3 Data Locking

Many algorithms and data structures may be partitioned into independent parts, with each part of the data structure having its own lock. Then the critical sections for each part of the data structure can execute in parallel, although only one instance of the critical section for a given part could be executing at a given time. Use data locking when contention must be reduced, and where synchronization overhead is not limiting speedups. Data locking reduces this overhead by distributing the instances of the overly-large critical section into multiple critical sections.

4.1.4 Data Ownership

Data ownership partitions a given data structure over the threads or CPUs, so that each thread/CPU accesses its subset of the data structure without any synchronization overhead whatsoever. However, if one thread wishes to access some other thread's data, the first thread is unable to do so directly. Instead, the first thread must communicate with the second thread, so that the second thread performs the operation on behalf of the first, or, alternatively, migrates the data to the first thread.

Data ownership might seem arcane, but it is used very frequently:

1. Any variables accessible by only one CPU or thread (such as `auto` variables in C and C++) are owned by that CPU or process.
2. An instance of a user interface owns the corresponding user's context. It is very common for applications interacting with parallel database engines to be written as if they were entirely sequential programs. Such applications own the user interface and his current action. Explicit parallelism is thus confined to the database engine itself.
3. Parametric simulations are often trivially parallelized by granting each thread ownership of a particular region of the parameter space.

If there is significant sharing, communication between the threads or CPUs can result in significant complexity and overhead. However, in situations where no sharing is required, data ownership achieves ideal performance. Such situations are commonly referred to as "embarrassingly parallel".

4.1.5 Parallel Fastpath

The idea behind the Parallel Fastpath design pattern is to aggressively parallelize the common-case code path without incurring the complexity that would be required to aggressively parallelize the entire algorithm. You must understand not only the specific algorithm you wish to parallelize, but also the workload that the algorithm will be subjected to. Great creativity and design effort is often required to construct a parallel fastpath.

Parallel fastpath combines different patterns (one for the fastpath, one elsewhere) and is therefore a template pattern. The following instances of parallel fastpath occur often enough to warrant their own patterns:

1. Reader/Writer Locking (4.1.6).
2. Pure RCU (4.1.7).
3. Reader-Writer-Lock/RCU Analogy (4.1.8).
4. RCU Existence Locks(4.1.9).
5. RCU Readers With NBS Writers (4.1.10).
6. Hierarchical Locking (4.1.11).
7. Resource Allocator Caches (4.1.12).

```

1 struct el {
2     struct el *next;
3     struct el *prev;
4     long key;
5     spinlock_t mutex;
6     struct rcu_head *rcu;
7     int data;
8     /* Other data fields */
9 };
10 spinlock_t listmutex;
11 struct el head;

```

Figure 17: Search-List Data Structures

```

1 struct el *_search(long key)
2 {
3     struct el *p;
4
5     p = head.next;
6     while (p != &head) {
7         if (p->key == key) {
8             return (p);
9         }
10        p = p->next;
11    }
12    return (NULL);
13 }

```

Figure 18: Internal Search Algorithm

4.1.6 Reader/Writer Locking

If synchronization overhead is negligible (i.e., the program uses coarse-grained parallelism), and only a small fraction of the critical sections modify data, then allowing multiple readers to proceed in parallel can greatly increase speedup. Writers exclude both readers and each other.

Reader/writer locking is a simple instance of asymmetric locking. Snaman [45] describes a more ornate six-mode asymmetric locking design used in several clustered systems. Asymmetric locking primitives can be used to implement a very simple form of the **Observer Pattern** [8]—when a writer releases the lock, all readers are notified of the change in state.

4.1.7 Pure RCU

The Pure RCU design pattern use RCU in its raw form to wait until all read-side critical sections have completed. This is used in Linux to allow non-maskable interrupt handlers to be dynamically updated at runtime. The updater waits for a grace period to elapse, after which it is guaranteed that there can no longer be any of the previous non-maskable interrupt handlers executing. It is then safe to clean up any state referenced by these handlers, and it is also safe to assume that any subsequent non-maskable interrupts will invoke the new handler.

4.1.8 Reader-Writer-Lock/RCU Analogy

The Reader-Writer-Lock/RCU Analogy describes how to replace reader-writer locking with RCU in cases where stale data and inconsistency can be tolerated, for example, in TCP/IP routing tables. This analogy is shown in Table 2. The left-hand column of this table shows a reader-writer-locked linked-list search and delete function, while the right-hand column shows the RCU equivalents. Figures 17 and 18 depict common code used by both the reader-writer locking and the RCU variants.

The locking in the left-hand side of Table 2 ensures that any deletions are atomic from the viewpoint of the searching code. However, if the list is read-intensive, the overhead of the locks in the `search()` code can be excessive, with the resulting contention restricting scaling.

Table 2: Reader-Writer Locking and RCU

Reader-Writer Lock	RCU
<pre> 1 int search(long key, int *result) 2 { 3 struct el *p; 4 5 read_lock(&listmutex); 6 p = _search(key); 7 if (p != NULL) 8 *result = p->data; 9 read_unlock(&listmutex); 10 return (p != NULL); 11 } </pre>	<pre> 1 int search(long key, int result) 2 { 3 struct el *p; 4 p = _search(key); 5 if (p != NULL) 6 *result = p->data; 7 return (p != NULL); 8 } </pre>
<pre> 1 int delete(long key) 2 { 3 struct el *p; 4 5 write_lock(&listmutex); 6 p = _search(key); 7 if (p == NULL) { 8 write_unlock(&listmutex); 9 } else { 10 p->next->prev = p->prev; 11 p->prev->next = p->next; 12 spin_unlock(&p->mutex); 13 write_unlock(&listmutex); 14 free(p); 15 } 16 return (p != NULL); 17 } </pre>	<pre> 1 int delete(long key) 2 { 3 struct el *p; 4 5 spin_lock(&listmutex); 6 p = _search(key); 7 if (p == NULL) { 8 spin_unlock(&listmutex); 9 } else { 10 p->next->prev = p->prev; 11 p->prev->next = p->next; 12 spin_unlock(&p->mutex); 13 spin_unlock(&listmutex); 14 call_rcu(&p->rcu, free, p); 15 } 16 return (p != NULL); 17 } </pre>

RCU permits the read-side locking to be eliminated in non-preemptive environments, as shown in the upper right portion of the table. Write-side locking can then be converted to the less-expensive spinlocks, as shown on the lower right portion of the table. However, the call to `free()` must be deferred through use of the `call_rcu()` primitive. The `call_rcu()` primitive enqueues a callback that, after a grace period has elapsed, invokes its second argument, passing this function its third argument. So, in this example, `call_rcu()` will enqueue a callback that invokes `free(p)` after a grace period elapses. The `call_rcu()` function uses its first argument to track this callback through the duration of the grace period.

Because readers run concurrently with writers, updates of multiple fields must of course be handled carefully.

4.1.9 RCU Existence Locks

One difficulty with conventional data locking is that the lock must reside outside of a given data structure if that structure is to be safely freed; otherwise, a reader might acquire a reference to the data structure just as it is being freed [9]. The RCU Existence Locks pattern permits such locks to be confined to the data structure that they protect by ensuring that freeing of a given data structure is deferred until all concurrent read-side dereferences complete. This design pattern is used pervasively in the K42 and Tornado research operating systems.

4.1.10 RCU Readers With NBS Writers

Non-blocking synchronization has required type-safe memory, where a given block of memory, once used for one type of structure, may never subsequently be used for any other type of structure. Recent NBS work removes this restriction [31, 13], but these algorithm require that readers execute expensive memory-barrier operations [7]. RCU readers may be used in conjunction with NBS updates to eliminate the need for both type-safe memory and read-side memory barriers.

4.1.11 Hierarchical Locking

One way of avoiding deadlock is to require that any task needing to acquire more than one lock first acquire a global lock. In this Hierarchical Locking design pattern, the fast path is available to tasks needing only one lock, as such tasks may simply directly acquire that lock. Tasks needing more than one lock must incur the greater overhead of also acquiring the global lock.

4.1.12 Allocator Caches

One way of efficiently implementing parallel memory allocators is to maintain per-CPU (or per-thread) caches [28]. In the common case, memory is available in the cache, and may thus be allocated without acquiring any locks. However, once this cache is exhausted, a more conservatively locked memory allocator must be used.

4.1.13 Critical-Section Fusing

If the overhead of the code between two critical sections is less than the overhead of the synchronization primitives, fusing the two critical sections will decrease overhead and increase speedups.

Critical-section fusing is a meta-pattern that transforms Data Locking into Code Locking and Code Locking into Sequential Program. In addition, it transforms more-aggressive variants of Code Locking and Data Locking into less-aggressive variants.

Critical-section fusing is the inverse of Critical-Section Partitioning.

4.1.14 Critical-Section Partitioning

If the overhead of the non-critical-section code inside a single critical section is greater than the overhead of the synchronization primitives, splitting the critical section can decrease overhead and increase speedups.

Critical-section partitioning is a meta-pattern that transforms Sequential Program into Code Locking and Code Locking into Data Locking. It also transforms less-aggressive variants of Code Locking and Data Locking into more-aggressive variants.

Critical-section partitioning is the inverse of Critical-Section Fusing.

4.2 Patterns for Transforming Algorithms to RCU

The basic RCU infrastructure has comparatively limited applicability. This section presents patterns that greatly extend RCU's reach by transforming algorithms to tolerate RCU's stale-data and inconsistency properties. These transformation patterns enable RCU to be used on a wide variety of real-world problems, as will be shown in Section 5. One of the authors (Paul) mined the following transformational patterns from earlier uses of RCU:

1. Mark Obsolete Objects (4.2.1),
2. Substitute Copy For Original (4.2.2),
3. Impose Level Of Indirection (4.2.3),
4. Global Version Number (4.2.4), and
5. Stall Updates (4.2.5).

The following sections give overviews of each of these patterns; more detail may be found elsewhere [24].

4.2.1 Mark Obsolete Objects

Mark Obsolete Objects transforms an algorithm that cannot tolerate stale data into one that is able to do so by marking deleted elements. Readers can then ignore any elements that are so marked.

This transformation requires some sort of synchronization, such as locking, at the element level in order to protect the “deleted” marking. The result of this transformation is therefore most useful for read-mostly mappings to often-written data structures that must be protected in any case. A single lock may then be used to protect both the element and the new “deleted” marking. In this case the mapping (e.g., linked list or hash table) is protected by RCU, while the elements themselves are protected by locking.

4.2.2 Substitute Copy For Original

Atomic updates of multiple data elements may be accomplished through use of the Substitute Copy For Original pattern, as is illustrated by Figure 7 in Section 1. The basic idea, long used in non-blocking synchronization [10, 12], is to hide a complex update behind a single atomic pointer update.

The overall effect is similar to that of the close-consistency found in the Andrew File System (AFS) [39], in that the update is invisible until the substitution, just like AFS modifications are not guaranteed to be visible until the file closes.

4.2.3 Impose Level Of Indirection

When the data items to be updated are scattered across much other data, it can be difficult and expensive to apply the Substitute Copy For Original pattern. This situation is addressed by the Impose Level Of Indirection pattern, in which these data items are gathered into a single structure that is referenced by a single pointer, so that Substitute Copy For Original can be easily applied.

4.2.4 Global Version Number

Global Version Number transforms an algorithm into a form where it can tolerate both stale and inconsistent data by maintaining a global version number and also associating a version number with each element. Readers can then sample the global version number before and after the access, and retry the access if there was an intervening change.

4.2.5 Stall Updates

The Stall Updates pattern may be used in conjunction with Global Version Number to prevent too-frequent updates from stalling readers. If a reader has performed an excessive number of retries, it may set a flag that stalls updates until the reader has successfully completed its access.

4.3 Discussion

The following two sections present an index of the locking design patterns and the RCU transformational design patterns.

4.3.1 Index to Locking Design Patterns

This section summarizes the locking design patterns with an index that compares and contrasts them, showing where each is most appropriate. The pattern listed in italics is a provisional pattern, because although it seems likely to be quite important, there is only one known use. Three uses are required for it to officially be considered a true pattern.

Table 3 compares how each of the design patterns resolves each of the following forces:

Speedup: Getting a program to run faster is the only reason to go to all of the time and trouble required to parallelize it. Speedup is defined to be the ratio of the time required to run a sequential version of the program to the time required to run a parallel version.

Table 3: Locking Design Pattern Force Index

Speedup	Contend	Ovhd	R/W	Complex	Pattern
---	+++	+++	---	+++	Sequential Program (4.1.1)
0	--	0	---	++	Code Locking (4.1.2)
+	+	0	+	--	Data Locking (4.1.3)
+++	+++	+?	+	?	Data Ownership (4.1.4)
++	++	++	+	--	Parallel Fastpath (4.1.5)
++	+	+	+++	-	Reader/Writer Locking (4.1.6)
+	+	-	0	---	Hierarchical Locking (4.1.11)
+	+	+	N/A	-	Allocator Caches (4.1.12)
+++	++	++	+++	-?	Pure RCU (4.1.7)
+	+	++	+++	+	RCU Existence Locks (4.1.9)
+++	++	++	+++	0	Reader-Writer-Lock/RCU Analogy (4.1.8)
+	+	++	+++	++	<i>RCU Readers With NBS Writers</i> (4.1.10)
0	-	+	0	--	Critical-Section Fusing (4.1.13)
0	+	-	0	+	Critical-Section Partitioning (4.1.14)

Contention: If more CPUs are applied to a parallel program than can be kept busy by that program, the excess CPUs are prevented from doing useful work by contention.

Overhead: A uniprocessor, single-threaded, non-preemptible, and non-interruptible¹ version of a given parallel program would not need synchronization primitives. Therefore, any time consumed by these primitives (including communication cache misses as well as locking primitives, atomic instructions, and memory barriers) is overhead that does not contribute directly to the useful work that the program is intended to accomplish. Note that the important measure is the relationship between the synchronization overhead and the overhead of the code in the critical section, with larger critical sections able to tolerate greater synchronization overhead.

Read-to-Write Ratio: A data structure that is rarely updated may often be protected with asymmetric synchronization primitives that reduce readers' synchronization overhead at the expense of that of writers, thereby reducing overall synchronization overhead.

Complexity: A parallel program is more complex than an equivalent sequential program because the parallel program has a much larger state space than does the sequential program. A parallel programmer must consider synchronization primitives, locking design, critical-section identification, and deadlock in the context of this larger state space.

This greater complexity often translates to higher development and maintenance costs. Therefore, budgetary constraints can limit the number and types of modifications made to an existing program, since a given degree of speedup is worth only so much time and trouble.

Plus signs indicate that a pattern resolves a force well. For example, Sequential Program resolves Contention and Overhead perfectly due to lack of synchronization primitives, and Complexity perfectly because sequential implementations of programs are better understood and more readily available than are parallel versions.

Minus signs indicate that a pattern resolves a force poorly. Again, Sequential Program provides extreme examples with Speedup since a sequential program allows no speedup² and with Read-to-Write Ratio because multiple readers cannot proceed in parallel in a sequential program.

Question marks indicate that the quality of resolution is quite variable. Programs based on Data Ownership can be extremely complex if CPUs must access each other's data. If no such access is needed, the programs can be as trivial as a script running multiple instances of a sequential program in parallel.

See the individual patterns for more information on how they resolve the forces.

4.3.2 Index to Transformational Patterns

Table 4 shows an index of RCU transformational patterns. The first five columns record whether the pattern resolves the corresponding force, and the last column gives the name of the pattern. This name is italicized for patterns with fewer than three uses, indicating a provisional pattern that has not yet withstood the test of time. The forces, which can also be thought of as attributes, are

Table 4: RCU Transformational Pattern Index

Fresh	Consistent	Atomic	Mutex	MB	Pattern
Y	N	N	Y	Y	Mark Obsolete Objects (4.2.1)
N	Y	Y	N	n	Substitute Copy For Original (4.2.2)
N	Y	Y	N	n	Impose Level Of Indirection (4.2.3)
Y	Y	N	N	Y	Global Version Number (4.2.4)
Y	Y	Y	Y	Y	<i>Stall Updates</i> (4.2.5)

as follows:

Fresh: Since RCU readers do not exclude writers, readers can find themselves referencing old versions of the data, or “stale data”. A pattern that addresses this force therefore transforms the algorithm into a form that rejects stale data, so that only fresh data is actually used. For example, in Mark Obsolete Object, readers reject any object that has been marked as obsolete.

Consistent: Again, RCU readers do not exclude writers, so that readers may see inconsistencies if writers make non-atomic changes or if readers access the same data multiple times. A pattern that addresses this force transforms the algorithm into a form that sees only consistent data, either by making changes atomically (for example, Substitute Copy For Original) or by rejecting inconsistencies (for example, Global Version Number).

Atomic: Atomicity prevents some types of inconsistency. A pattern that addresses this force transforms the algorithm from a form in which readers can see partially completed updates into a form where updates appear atomic to readers.

Mutex: The presence of this force indicates that readers must perform some sort of mutual exclusion, involving something like locking or atomic instructions. This mutual exclusion will be reasonably light weight, for example, Mark Obsolete Objects requires per-element mutual exclusion but no global locks. Nonetheless, the added read-side overhead will cause the resulting program to more poorly resolve the upper-level Read-to-Write Ratio, Speedup, and Overhead forces.

MB (Memory Barrier): The presence of this force indicates that readers must execute a memory-barrier instruction on CPUs with weak memory consistency models. For example, in the Global Version Number pattern, the readers must use memory barriers to ensure that the first snapshot of the version number is taken before any other accesses, and that the last snapshot is taken after any other accesses. Again, the added read-side overhead will cause the resulting program to more poorly resolve the upper-level Read-to-Write Ratio, Speedup, and Overhead forces. In addition, the memory barriers result in additional Complexity.

Entries marked with a lower-case “n” are “no” on all CPU architectures except for DEC Alpha.

These two indices assist in determining when to apply RCU to a given algorithm, and which patterns are appropriate in those cases where use of RCU is appropriate.

5 Selected Applications of RCU Design Patterns

This section presents the application of RCU to the Linux System V IPC implementation in Section 5.1, summarizing that implementation’s use of RCU-related design and transformation patterns. Section 5.2 summarizes the experience and discusses the results of applying RCU. The patterns used by the Linux 2.6 kernel as of November 29, 2004 are summarized in Table 5.

More details on additional Linux 2.6 kernel uses of RCU may be found elsewhere [24].

Table 5: RCU Usage in Linux as of November 29, 2004

Subsystem	RCU Pattern
System V IPC	Mark Obsolete Objects (4.2.1) Impose Level Of Indirection (4.2.3) Reader-Writer-Lock/RCU Analogy (4.1.8) Substitute Copy For Original (4.2.2)
Directory-entry cache	Global Version Number (4.2.4) RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8) Stall Updates (4.2.5)
RAID-related drivers	Reader-Writer-Lock/RCU Analogy (4.1.8)
system-call audit	Reader-Writer-Lock/RCU Analogy (4.1.8)
Module load failure races	Pure RCU (4.1.7)
slab cleanup	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
tasklist patch	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
FD management patch	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8) Substitute Copy For Original (4.2.2)
Dynamic NMI handlers	Pure RCU (4.1.7)
Hotplug HW removal	Pure RCU (4.1.7)
PPC64 PTE free	Pure RCU (4.1.7)
IP route cache	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
DECNET routing	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
SNAP protocol unregister	Pure RCU (4.1.7) RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
802.1Q VLAN	Reader-Writer-Lock/RCU Analogy (4.1.8) Pure RCU (4.1.7)
Ethernet bridge	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
Combined bridging/routing	Pure RCU (4.1.7)
Packet handler	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
IPv4/IPv6 Net filter	Pure RCU (4.1.7) RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
IPv4 and IPv6 protocol switch	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
IPv4 tunneling	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
Raw socket protocol	RCU Existence Locks (4.1.9)
address resolution	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)
traffic scheduling	RCU Existence Locks (4.1.9) Reader-Writer-Lock/RCU Analogy (4.1.8)

5.1 System V IPC

This section describes how RCU was used to break up the global locks used by Linux’s System V IPC primitives, as described in an earlier publication by some of the authors and others [4]. These locks guard the following: (1) the map from IPC identifiers to corresponding `kern_ipc_perm` structures, (2) expansion of the mapping arrays, and (3) individual IPC operations. The role of these locks is shown in a schematic of the Linux 2.4 System semaphore data structures in Figure 19; the message-queue and shared-memory data structures were changed similarly. The global semaphore protects array-grow operations, and the global spinlock protects all other operations. This arrangement does have the advantage of simplicity, but unfortunately also severely restricts throughput and scaling, even on two-CPU systems.

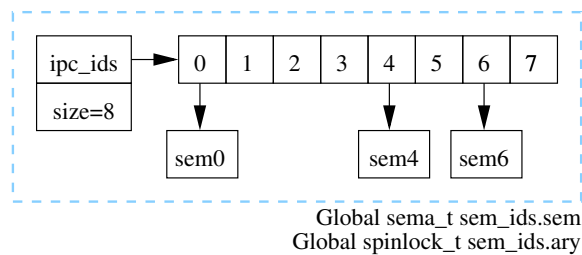


Figure 19: Linux 2.4 Kernel Semaphore Locking

A straightforward modification would replace these global locks with reader-writer locks, using the Reader/Writer Locking pattern described in Section 4.1.6, allowing mapping operations to be performed in parallel.

However, with the assistance of Dipankar Sarma and Maneesh Soni, Mingming Cao took the additional step of following the Reader-Writer-Lock/RCU Analogy described in Section 4.1.8, replacing the global locks with (1) RCU to guard the mapping array and (2) per-`kern_ipc_perm` locks to guard the IPC operations using the Data Locking pattern described in Section 4.1.3, which resulted in significant system-level speedups on database benchmarks. This modification also serves to illustrate use of the Mark Obsolete Objects pattern (described in Section 4.2.1) to prevent access to stale data, with a new per-semaphore `deleted` flag serving as the marking. In addition, the Impose Level Of Indirection pattern (described in Section 4.2.3) was used to simplify array-growth operations. Figure 20 shows a schematic of the new Linux 2.6 System semaphore data structures.

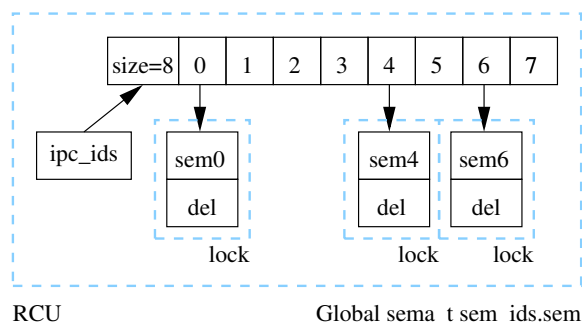


Figure 20: Linux 2.6 Kernel Semaphore Locking

5.1.1 Semaphore Data Structures Changes

There are three basic changes between the semaphore data structures of Linux 2.4 and 2.6, shown in Figures 19 and 20, respectively:

1. A `lock` has been added to each “Sem” data structure
2. The array size has been moved into the array
3. A `deleted` flag has been added to each “Sem” data structure.

The per-semaphore `lock` is necessary because the global lock has been eliminated. Each semaphore’s operations are serialized by its lock, permitting different semaphores to be operated on concurrently.

Moving the array size into the array ensures that synchronization-free RCU readers always see a size that is consistent with the actual array. In absence of such consistency, a reader might see a large size and a small array, which could result in memory corruption due to indexing off the end of the small array. Although it is also possible to ensure consistency by carefully ordering operations, placing the size into the array is both simpler and more efficient.

The `deleted` flag allows synchronization-free readers to reject concurrently deleted semaphores, as is shown in the following sections.

5.1.2 Semaphore Removal

```
1 void
2 ipc_rmid(struct ipc_ids* ids, int id)
3 {
4     struct kern_ipc_perm* p;
5     int lid = id % SEQ_MULTIPLIER;
6
7     down(&ids->sem);
8     p = ids->entries[lid].p;
9     spin_lock(&p->lock);
10    ids->entries[lid].p = NULL;
11    p->deleted = 1;
12    spin_unlock(&p->lock);
13    up(&ids->sem);
14    ipc_rcu_putref(p);
15 }
```

Figure 21: RCU Semaphore Deletion

The deletion process is performed by `ipc_rmid`, as shown by the pseudocode in Figure 21. The actual code is more complex due to details of the implementation, and is publicly available for study [51]. Line 7 acquires `sem` to exclude concurrent array resizing, line 8 picks up a pointer to the semaphore’s data structure, and line 9 acquires the per-semaphore lock to exclude concurrent semaphore operations. Line 10 clears the pointer from the array to the semaphore, and line 11 sets the semaphore’s `deleted` flag. Lines 12 and 13 release locks, and line 14 arranges for the semaphore’s data structure to be freed at the end of a subsequent grace period.

5.1.3 Semaphore Lock Acquisition

The `ipc_lock()` primitive shown in Figure 22 maps a semaphore ID to a pointer to the corresponding locked data structure, or to `NULL` if the ID is invalid. Line 8 marks the beginning of a read-side RCU critical section, disabling preemption in a preemptive kernel. Line 9 picks up a pointer to the mapping array. Lines 10-13 return `NULL` if the ID to be mapped is out of range. Line 14 picks up a pointer to the data structure corresponding to the ID, and lines 15-18 return `NULL` if there is no such structure. Line 19 acquires a lock on the semaphore data structure, and lines 20-24 return `NULL` if the semaphore has been deleted. Note that the lock serializes deletion and lookup of a given semaphore. Line 25 returns a pointer to the semaphore data structure.

```

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     entries = rcu_dereference(ids->entries);
10    if(lid >= entries->size) {
11        rcu_read_unlock();
12        return NULL;
13    }
14    out = entries->p[lid];
15    if(out == NULL) {
16        rcu_read_unlock();
17        return NULL;
18    }
19    spin_lock(&out->lock);
20    if (out->deleted) {
21        spin_unlock(&out->lock);
22        rcu_read_unlock();
23        return NULL;
24    }
25    return out;
26 }

```

Figure 22: RCU Detecting Semaphore Deletion

Note that this function leaves the structure locked upon return. The caller must invoke `ipc_unlock()` to unlock it.

5.1.4 Semaphore Operation

This section presents a graphical demonstration of how mapping-array growth (due to semaphore creation), semaphore deletion, and semaphore lookup proceed concurrently. Figure 23 shows a system with three semaphores allocated out of a maximum of eight that could be accommodated.

The intermediate results of a array growth and semaphore deletion are shown in Figure 24. At this point, concurrent semaphore lookup would see semaphore 4 as being deleted (note the "D" in the diagram), and might or might not see the newly created semaphore 2, depending on the exact timing. Failing to see semaphore 2 is legal, since this semaphore was created *after* semaphore lookup started execution. A subsequent `ipc_lock()` would see semaphores 0, 2, and 6, but would not newly deleted semaphore 4.

Finally, Figure 25 shows the state of the system after a grace period. The old 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.

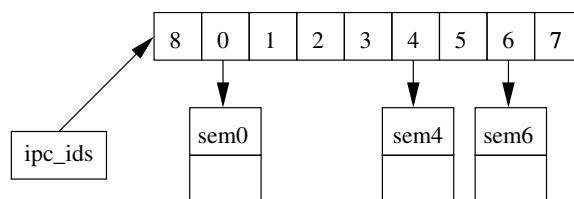


Figure 23: Semaphore Initial State

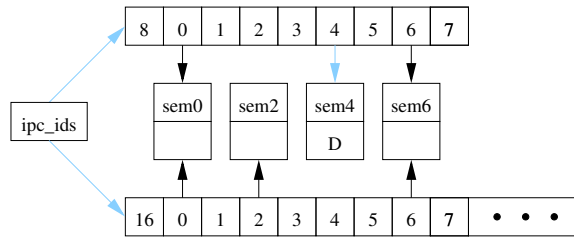


Figure 24: Semaphore Structures After Array Replacement

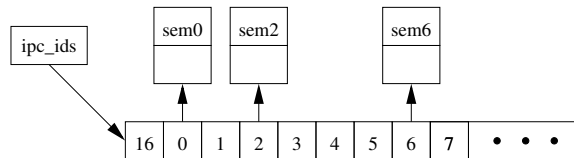


Figure 25: Semaphore Structures After Grace Period

5.1.5 Semaphore Discussion

The RCU changes to Linux’s System V IPC implementation resulted in excellent performance improvements combined with negligible increases in complexity, as is shown in the following two sections. The order-of-magnitude performance improvements make this subsystem the RCU “poster child” for RCU performance improvement, though the more recent RCU version of the audit-validation cache [33] also shows impressive results.

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 memory.

The raw transaction rate for each of the five runs with each kernel are shown in Figure 26, which shows better than a 5% improvement due to RCU. 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 6, 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 constructed a System V semaphore microbenchmark named *semopbench* and ran it on an Intel 8-CPU 700 MHz PIII system. The results in Table 7 clearly show the order-of-magnitude reduction in runtime obtained by applying the reader-writer-locking/RCU analogy to

Table 6: DBT1 Database Benchmark Results (TPS)

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

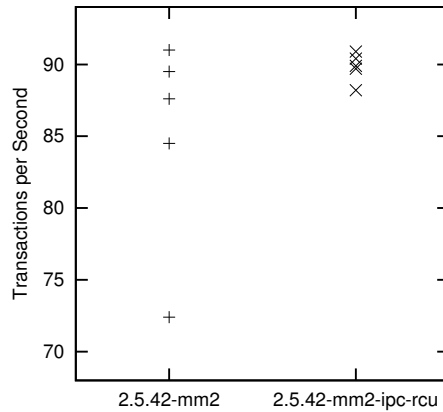


Figure 26: DBT1 Database Benchmark Raw Results

Table 7: semopbench Microbenchmark Results (seconds)

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

System V IPC mechanisms.

Semaphore Complexity The RCU changes to the System V IPC implementations inflicted less than 5% expansion of code size, as shown in Table 8. This change increased the overall code size by only 151 lines. This order-of-magnitude performance benefit is well worth the modest increase in complexity.

Table 8: Semaphore Change in Lines of Code

	Ins/Del/Delta			Total Lines		% Delta
	Ins	Del	Delta	New	Old	
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%

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.

5.2 Discussion

This section presented several case studies on the application of RCU to operating-system kernels, primarily Linux 2.6.0, and also summarized several tens of uses in VM/XA. DYNIX/ptx,

K42/Tornado, and the Linux 2.4 and 2.6 kernels. These uses of RCU typically produced large performance increases, in one case increasing the performance of System V semaphores by more than an order of magnitude, as described in Section 5.1. In some cases, RCU enabled functionality that could be accomplished only with great difficulty using traditional locking schemes, as described by Gamsa et al. [9] and Appavoo et al. [3]. In other cases, use of RCU allowed other locking primitives to be done away with, for example, brlock was eliminated from the Linux 2.5.69 and 2.5.70 kernels.

Each of the case studies used several of the RCU design patterns and transformational patterns described in Section 4, validating their use and structure. Future work includes analyzing RCU-related code as it is produced to identify additional RCU patterns and to refine existing RCU patterns.

6 Comparison of RCU and Locking

Although the analytic results presented in the previous section allow the performance of different synchronization primitives to be compared across a wide variety of hardware platforms and workloads, such results cannot take the place of actual measurements for specific platforms and workloads. To fill this gap, Section 6.1 presents measurements of the hash-table workload described in Section 1, and Section 6.2 presents analytic comparisons of RCU and locking under conditions that are difficult to measure, including large numbers of CPUs and varying memory-latency ratios.

6.1 Measured Comparison to Locking

Given the long history of and deep familiarity with locking, one would expect a new synchronization mechanism such as RCU to be adopted only if it provides some overwhelming advantages over locking. From the discussion in the preceding section, one would expect RCU to have overwhelming performance advantages on read-mostly workloads, but that these advantages would wane rapidly with increasing update intensity.

This section tests these expectations, using the same hash-table mini-benchmark that was presented in Section 1, running on a 4-CPU 700MHz P-III system. Although a variant of Linux RCU devised by Manfred Spraul [47] has run successfully on SMP machines with as many as 512 CPUs [48], however, detailed performance measurements are not yet available. These tests varied the update fraction f and measured the throughput of hash operations (searches and updates) per microsecond for each value of f and for each of the following types of synchronization mechanisms:

ideal: the single-threaded, synchronization-free performance, multiplied by the desired number of CPUs (four in this case).

rcu: the RCU primitive described in this paper, using per-hash-chain spinlocks to guard updates.

smr: Michael's safe memory reclamation variant of NBS, which is the most efficient known NBS algorithm for read-mostly workloads, also developed independently by researchers at Sun [30, 31, 13, 14].

spinbkt: a per-bucket spinlock, placed in a cacheline separate from the hash-chain header pointers.

globalrw: a global reader-writer spinlock.

brlock: the Linux 2.4 kernel's "big reader lock", which provides a lock per CPU, so that reading CPUs acquire only their own lock and writing CPUs acquire all CPUs' locks.

global: a global spinlock.

One of the authors (Paul) has previously presented measurements for this same benchmark taken on other types of CPUs and covering more of the parameter space [25, 24].

The results are displayed in Figure 27, with the graph on the left showing data from a small 32-bucket hash table that fits into CPU on-chip cache, and with the graph on the right showing data from a large 16,384-bucket hash table that overflows CPU on-chip cache. RCU achieves

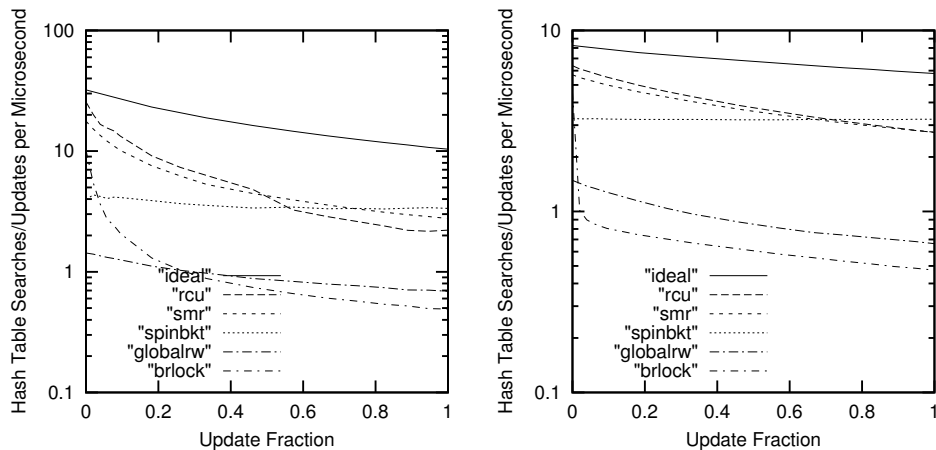


Figure 27: Four-CPU Hash Table Performance for Mixed Workload

ideal performance on a read-only workload, but is also the best synchronization mechanism up to an update fraction of about 0.5, despite failing to achieve ideal performance. It uniformly beats brlock by more than a factor of two, and manages at least 50% of the performance of the best synchronization mechanism on an update-only workload. Note that achieving ideal performance on an update-only workload is outside the scope of this paper, which focuses on the read-mostly case.

6.2 Analytic Comparison to Locking

Although actual measurements are normally preferable to analytic results, the latter must suffice in situations where direct measurement is infeasible, for example, for very large numbers of CPUs and variable memory-latency ratios.

Figure 28 shows regions of optimality for each of spinlock, brlock, and RCU. The plot on the left assumes one update per CPU per ten grace periods on average, while the plot on the right assumes ten updates per CPU per grace period on average. For comparison, note that the Linux kernel has been measured at over 1,000 updates per grace period under heavy load on a single-CPU system. For the smaller number of updates per grace period, RCU's region of optimality decreases with increasing number of CPUs up to about 10 CPUs, at which point it increases, @@@. For the larger number of updates per grace period, RCU's region of optimality increases monotonically with increasing numbers of CPUs.

Figure 29 shows regions of optimality of spinlock, brlock, and RCU as the computer system's memory-latency ratio varies. The memory-latency ratio is defined to be the ratio of the latency of the largest on-chip CPU cache to that of main memory. Historically, this ratio has been increasing, as noted in Section 1. Again, the plot on the left shows regions of optimality for one update per CPU per ten grace periods, and the plot on the right shows regions of optimality for ten updates per CPU per grace period. For the low update rate, the workload must be increasingly read-intensive (smaller update fraction) as the memory-latency ratio rises. However, for the more intense update rate, RCU is optimal for increasingly update-intensive workloads as the memory-latency ratio rises.

In both cases, the more heavily RCU is used, the greater its performance advantage in more severe environments.

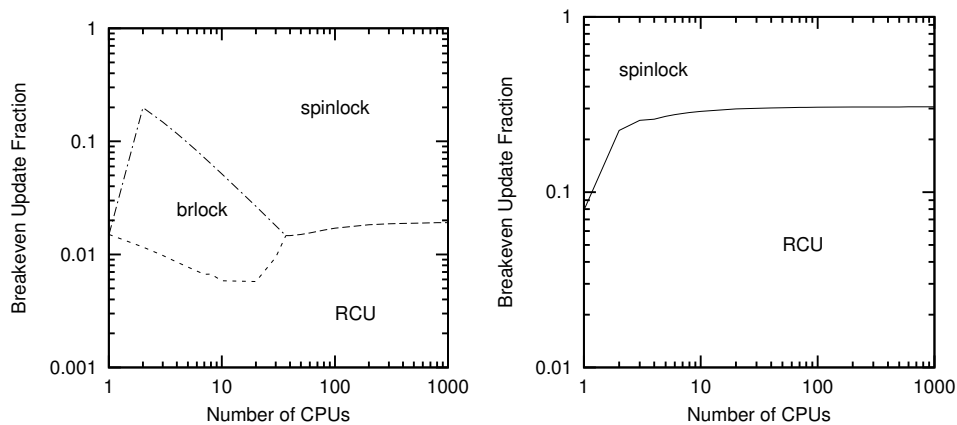


Figure 28: Analytic Breakevens for Large Numbers of CPUs

7 Related Work

This section gives an overview of synchronization mechanisms that permit readers synchronization-free access: the readers need not acquire locks, use any atomic operations beyond normal load and store, or, on most CPU architectures, execute any special memory-barrier instructions.

Section 7.1 reviews the deferred-destruction literature, which extends back a quarter century, Section 7.2 discusses future prospects for RCU given four scenarios of system-architecture evolution, and Section 7.3 summarizes RCU's future prospects.

7.1 Deferred Destruction

Deferred destruction was first described in 1980, when Kung and Lehman [18] recommended use of a garbage collector to defer destruction of nodes in a parallel binary search tree in order to simplify its implementation. This good advice does not help much in environments that lack a garbage collector, such as most operating-system kernels. Even environments possessing garbage collectors typically impose significant overhead on readers, for example to maintain reference counts. In addition, it is not clear how great a performance increase can be obtained from a binary search tree given the frequent updates to the root node, in light of the high cost of pipeline stalls and memory latency.

In 1982, Manber and Ladner [19, 20] recommended deferring destruction until all threads running at that time have terminated. This recommendation is not helpful in operating-system-kernel or server environments, where threads are explicitly designed to *not* terminate. Nonetheless, this approach does eliminate all read-side synchronization overhead. However, it again is not clear how great a performance increase can be obtained from parallel search trees given the high cost of pipeline stalls and memory latency.

In 1986, Hennessy, Osisek, and Seigh [11] introduced passive serialization, which is a deferred-destruction mechanism that relies on the presence of "quiescent states" in the VM/XA hypervisor that are guaranteed not to be referencing the data structure. However, this mechanism was not optimized for the memory latency and pipeline-stall overheads found on modern computer systems, which is not surprising given that these overheads were not so expensive at that time. A similar mechanism was prototyped for the Linux 2.6 kernel [4], which does in fact suffer from these overheads, which may explain why passive serialization was not applied very widely in VM/XA. Nonetheless, passive serialization appears to be the first deferred-destruction mechanism to be used in production.

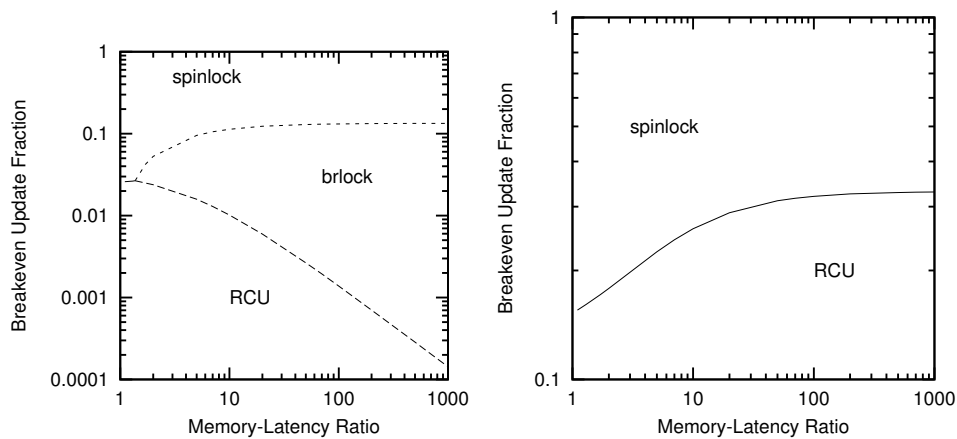


Figure 29: Analytic Breakevens for Varying Memory-Latency Ratio

In 1990, Pugh [35] noted that explicitly tracking which threads were reading a given data structure permitted deferred free to operate in the presence of non-terminating threads. However, this explicit tracking imposes significant read-side overhead, which is anything but desirable in read-mostly situations. Although this algorithm takes pains to avoid write-side contention and parallelize the other write-side overheads by providing a fine-grained locking design, the high costs of pipeline stalls and memory latency make it unclear how much of the performance advantage reported in 1990 remains in 2004.

At about this same time, Adams [1] described “chaotic relaxation”, where the normal barriers between successive iterations of convergent numerical algorithms are relaxed, so that iteration n might use data from iteration $n - 1$ or even $n - 2$. This introduces error, which typically slows convergence and thus increases the number of iterations required. However, this increase is sometimes more than made up for by a reduction in the number of expensive barrier operations, which are otherwise required to synchronize the threads at the end of each iteration. Unfortunately, chaotic relaxation requires highly structured data, such as the matrices used in scientific programs, and is thus inapplicable to most data structures in operating-system kernels. @@@ add reference to Peter Strazdins later work? Hey, he -did- stay awake through my entire 1998 RCU presentation! ;-) @@@

In 1993, Jacobson [16] described what is perhaps the simplest deferred-free technique: simply waiting a fixed amount of time before freeing blocks awaiting deferred free. Note that Jacobson did not describe any write-side changes he might have made in this work using SGI’s Irix kernel. Aju John published a similar technique in 1995 [17], in which the write-side changes were described in more detail. Jacobson’s and John’s approach works well if there is a well-defined upper bound on the length of time that reading threads can hold references, as there might well be in hard real-time systems. However, if this time is exceeded, perhaps due to preemption, excessive interrupts, or larger-than-anticipated load, memory corruption can ensue, with no reasonable means of diagnosis. Jacobson’s technique is therefore inappropriate for use in production operating-system kernels, except for those kernels that can provide hard real-time response guarantees for all operations.

In 1995, Pu et al. [34] applied a technique similar to that of Pugh’s read-side-tracking to permit replugging of algorithms within a commercial Unix operating system. However, this replugging permitted only a single reader at a time. The following year, this same group of researchers extended their technique to allow for multiple readers [6]. Their approach requires memory barriers

(and thus pipeline stalls), but reduces memory latency, contention, and locking overheads.

Finally, in 2002, Michael [30, 31] presented techniques that defer the destruction of data structures to simplify NBS synchronization. In particular, this technique eliminates locking, reduces contention, reduces memory latency for readers, and parallelizes pipeline stalls and memory latency for writers. However, these techniques still impose significant read-side overhead, particularly in the form of memory barriers, as discussed in Section 1. Researchers at Sun worked along similar lines concurrently [13, 14].

These mentions of deferred destruction, though important, did not present an efficient mechanism for determining how long to defer destruction in SMMP environments lacking a garbage collector but with long-running threads. The lack of such a mechanism prevented the benefits of deferred destruction from being realized.

This gap was filled by one of the authors's work on DYNIX/ptx's RCU [29, 41] and by the Tornado and K42 research operating systems' "generations", each of which provides a NUMA-optimized specialized garbage collector to implement deferred destruction in an easy-to-use fashion. In both cases, the key idea is a straightforward API that allows data to be queued for later destruction, so that the programmer need not be concerned with the details of the actual destruction deferral implementation. The properties of the write-side access are dictated by the write-side synchronization mechanism, but typically parallelize pipeline stalls, memory latency, and locking while reducing contention. In all three operating systems, deferred destruction enables synchronization-free read-only access. However, this use was restricted to a few subsystems in the case of DYNIX/ptx and to existence locking in the case of K42 [9]. Widespread use was hindered by the lack of a robust set of design patterns for the typical practitioner to follow, a lack that was addressed in Section 4.

The attributes of these approaches to deferred destruction are summarized in Table 9. The meanings of the entries are as follows:

- A** Avoids the specified type of overhead.
- P** Parallelizes the specified type of overhead.
- Y** Supports the specified attribute.
- N** Includes special support for NUMA machines.
- S** Includes special support for SMMP machines.

A lower-case character indicates partial support. This table shows the evolution of deferred destruction from an academic curiosity to a production-ready mechanism. Although deferred destruction has progressed quite far, there is still much work left to be done, as is outlined in Section 8.

The last two rows of this table show Linux implementations for the 2.4 and 2.6 kernels, respectively. The Linux 2.6 implementation also adds the following features, not present in any of the other deferred-destruction implementations, as of the 2.6.10 kernel:

1. Soft realtime response time [38].
2. Handling of multiple sets of quiescent states and corresponding disjoint grace periods [37].
3. Protection against network-based denial-of-service attacks [37].
4. Incorporating memory barriers into other primitives so that use of RCU no longer requires use of explicit memory barriers.

7.2 Future Scenarios

The main reason that RCU is of interest now is the fundamental change in computer-system architecture that has played out over the past 20 years. As computer-system architecture continues to change, it is reasonable to ask whether RCU will continue to be useful on future computer systems. Predicting the future course of computer-system evolution has proven to be fraught with peril, so in the following sections will examine four possible scenarios: (1) Uniprocessor Über Alles, (2) Multithreaded Mania, (3) More of the Same, and (4) Crash Dummies Slamming into the

Table 9: Attributes of Deferred Destruction Mechanisms

	Year of Publication	Year in Production	Architecture Optimizations	Appropriate for OS Kernel	Independently Invented	Reads:				Writes:			
						Pipeline Flushes	Memory Latency	Contention	Locking	Pipeline Flushes	Memory Latency	Contention	Locking
Kung/Lehman	80				Y			A	A	?	?	?	?
Manber/Ladner	82					A	A	A	A	?	?	?	?
Passive Serialization	89	86?		Y	Y	A	A	A	A	p	p	a	p
Pugh	90			y			?	A	A	p	p	a	p
Chaotic Relaxation	91				y	a	a	a	a	a	a	a	a
Jacobson	93				Y	A	A	A	A	?	?	?	?
John	95				Y	A	A	A	A	?	?	?	?
Pu	95			Y	Y		a	a	a				
Dynix/PTX RCU	98	93	SN	Y	Y	A	A	A	A	p	p	a	p
K42 Generations	99		SN	Y	Y	A	A	A	A	p	p	a	p
NBS & Hazard Pointers	02			y			a	a	A	p	p	a	A
SuSE 7.3 Update RCU	01	02	S	Y		A	A	A	A	p	p	a	p
Linux 2.6 RCU	01	04	S	Y		A	A	A	A	p	p	a	p

Memory Wall.

7.2.1 Uniprocessor Über Alles

In this scenario, the combination of Moore’s-Law increases in CPU clock rate and continued progress in horizontally scaled computing render SMMP systems irrelevant. This scenario is therefore dubbed “Uniprocessor Über Alles”, literally, uniprocessors above all else.

These uniprocessor systems would be subject only to instruction overhead, since memory barriers, cache thrashing, and contention do not affect single-CPU systems. In this scenario, RCU is useful only for niche applications, such as interacting with NMIs. It is not clear that an operating system lacking RCU would see the need to adopt it, although operating systems that already implement RCU might continue to do so.

However, recent progress with multithreaded CPUs indicates that this scenario is quite unlikely outside of very constrained embedded systems.

7.2.2 Multithreaded Mania

A less-extreme variant of Uniprocessor Über Alles still features uniprocessors, but with hardware multithreading. In fact, multithreaded CPUs are now standard for many desktop and laptop computer systems. The most aggressively multithreaded CPUs share all levels of cache hierarchy, thereby eliminating CPU-to-CPU memory latency, in turn greatly reducing the performance penalty for traditional synchronization mechanisms. However, a multithreaded CPU would still incur overhead due to contention and to pipeline stalls caused by memory barriers. Furthermore, because all hardware threads share all levels of cache, the cache available to a given hardware thread is a fraction of what it would be on an equivalent single-threaded CPU, which can degrade performance for applications with large cache footprints. There is also some possibility that the restricted amount of cache available will cause RCU-based algorithms to incur performance penalties due to their grace-period-induced additional memory consumption. Investigating this

possibility is future work.

However, in order to avoid such performance degradation, a number of multithreaded CPUs and multi-CPU chips partition at least some of the levels of cache on a per-hardware-thread basis. This increases the amount of cache available to each hardware thread, but re-introduces memory latency for cachelines that are passed from one hardware thread to another.

In either case, RCU helps to avoid contention and also the pipeline-stall overhead due to memory barriers. An interesting open question is whether future CPUs will feature instruction and memory-access reordering. If they do not, then the benefits of RCU will be greatly reduced, though RCU will still enjoy some performance advantage due to its: (1) reduced need for atomic instructions, (2) elimination of read-side bookkeeping operations, such as hazard pointers, and (3) providing the compiler greater freedom to use optimizations that reorder code.

7.2.3 More of the Same

The More-of-the-Same scenario assumes that the memory-latency ratios will remain roughly where they are today.

This scenario actually represents a change, since to have more of the same, interconnect performance must begin keeping up with the Moore's-Law increases in core CPU performance. In this scenario, overhead due to memory latency, contention, and potentially pipeline stalls remains significant, so that RCU retains the high level of applicability that it enjoys today.

7.2.4 Crash Dummies Slamming into the Memory Wall

If the memory-latency trends shown in Figure 1 continues, then memory latency will continue to grow relative to instruction-execution overhead. Systems such as Linux that have significant use of RCU will find additional use of RCU to be profitable. As can be seen in Figure @@@, if RCU is heavily used, increasing memory-latency ratios give RCU an increasing advantage over other synchronization mechanisms. In contrast, systems with minor use of RCU will require increasingly high degrees of read intensity for use of RCU to pay off. As can be seen in this figure, if RCU is lightly used, increasing memory-latency ratios put RCU at an increasing disadvantage compared to other synchronization mechanisms. Since Linux has been observed with over 1,600 callbacks per grace period under heavy load, it seems safe to say that Linux falls into the former category.

However, if memory latency increases too much relative to instruction-execution overhead, we will likely find ourselves in either the Uniprocessor Über Alles or the Multithreaded Mania scenario, due to the fact that a large body of existing SMMP software would exhibit increasingly poor scalability, causing SMMP systems to exhibit increasingly poor price-performance measures.

7.3 Summary

Low-end systems are already moving in the direction of Multithreaded Mania, as hyperthreading is available even in low-end x86 CPUs, such as those used in desktops and laptops.

Multi-CPU chips are in the offing, and these chips will boast low memory-latency overheads for data fetched out of other CPUs' caches. For these mid-range systems, memory latency is likely to improve as rapidly as instruction-execution rates, so that these systems will fall into the More of the Same scenario.

It is more difficult to predict how high-end systems will progress, since such systems are manufactured in low volumes for a relatively small number of workloads. Historical trends are consistent with Crash Dummies Slamming into the Memory Wall, but it is conceivable that improved interconnect technology might bring high-end systems into the realm of More of the Same.

The outlook is therefore good for RCU, since it provides substantial performance or simplicity benefits to all scenarios except for Uniprocessor Über Alles, which seems unlikely to be the sole scenario due to the advent of multithreading in low-end systems. Some possible exceptions to this outlook are low-end embedded systems where cost and power-consumption constraints still

favor single-threaded uniprocessors, and potentially systems with heavily multithreaded single-issue CPUs and fully shared cache hierarchies.

As always, making predictions in the computing field is fraught with hazards, but it appears that RCU will be with us for the foreseeable future.

8 Conclusions

RCU has been used successfully in a number of kernel environments, including VM/XA, Dynix/PTX, Tornado, K42, and Linux. In these environments, it has provided scalability and performance with very little added complexity, particularly for read-mostly data structures, where RCU eliminates the need for readers to execute synchronization operations. Because the performance of synchronization operations has fallen far behind that of normal instructions over the past two decades, this synchronization-free property enables RCU to provide order-of-magnitude speedups on real systems.

Of course, the fact that readers need not execute synchronization operations means that readers may see stale or inconsistent data in face of concurrent updates. Although this is acceptable in a surprisingly large number of situations, such as routing tables, where the updates necessarily lag changes in external physical state, there are many algorithms that cannot tolerate RCU's staleness and inconsistency properties. The transformational design patterns presented in this paper may be applied to such algorithms in order to transform them into a form that can tolerate such staleness and inconsistency. These patterns were used to apply RCU to the Linux 2.6 kernel's System V IPC implementation, resulting in order-of-magnitude speedups for System V semaphore operations on an 8-CPU system.

Since RCU is designed for read-mostly situations, the question of RCU's area of applicability naturally arises. This paper presents a parametric study showing that RCU provides substantial performance advantages even when update operations constitute several tens of percent of the workload. This result demonstrates RCU's broad applicability.

Use of RCU has proven quite beneficial in a number of environments. However, it was not until RCU was exposed to a large number of users as a part of this work in Linux that the RCU design patterns were fully refined and codified. This activity has opened up a huge number of research directions.

The Linux 2.6 kernel's RCU infrastructure has evolved rapidly over the past few years. However, we expect that additional use will uncover further performance, usability, and real-time response improvements to Linux's RCU infrastructure.

RCU has been applied to a number of areas in the Linux kernel, including networking, directory-entry cache, security auditing, and dynamic interrupt handling. RCU continues to be applied to new areas, and we expect that these new applications will unearth new RCU design patterns, and motivate additional performance studies.

Thus far, production use of RCU has been restricted to low-level environments, such as operating-system kernels and virtual-machine monitors like VM/XA. Additional work is needed to determine whether RCU is applicable to application-level environments, perhaps in conjunction with non-blocking synchronization.

Finally, there is not yet a formal statement of RCU semantics. While this lack has thus far not been an impediment to adoption and use of RCU, it is quite possible that formal semantics would point the way towards tools that automatically validate uses of RCU or that permit RCU algorithms to be automatically generated by a parallel compiler.

Perhaps the most fascinating question is "Why has the uptake of RCU been so slow?" The first RCU-related paper that we are aware of was published in 1980 [18], but even as late as 2000, RCU was obscure at best, used only by a very few small groups of researchers and developers [9, 11, 29]. Therefore, although RCU has been in production use for well over a decade, there is still much work to be done and much to be learned about this still-novel synchronization primitive. We enthusiastically invite you to join us in this exciting exploration.

9 Acknowledgments

@@@ A polite author always includes acknowledgments. You should thank everyone, especially those who funded the work.

References

- [1] ADAMS, G. R. *Concurrent Programming, Principles, and Practices*. Benjamin Cummins, 1991.
- [2] ALEXANDER, C. *The Timeless Way of Building*. Oxford University Press, New York, 1979.
- [3] APPAVOO, J., HUI, K., SOULES, C. A. N., WISNIEWSKI, R. W., DA SILVA, D. M., KRIEGER, O., AUSLANDER, M. A., EDELSON, D. J., GAMSA, B., GANGER, G. R., MCKENNEY, P., OSTROWSKI, M., ROSENBERG, B., STUMM, M., AND XENIDIS, J. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal* 42, 1 (January 2003), 60–76.
- [4] ARCANGELI, A., CAO, M., MCKENNEY, P. E., AND SARMA, D. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)* (June 2003), USENIX Association, pp. 297–310.
- [5] COPLIEN, J. O., AND SCHMIDT, D. C., Eds. *Pattern Languages of Program Design*, vol. 1. Addison Wesley, Reading, Massachusetts, 1995.
- [6] COWAN, C., AUTREY, T., KRASIC, C., PU, C., AND WALPOLE, J. Fast concurrent dynamic linking for an adaptive operating system. In *International Conference on Configurable Distributed Systems (ICCDs'96)* (Annapolis, MD, May 1996), p. 108.
- [7] FRASER, K. A. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, 2003.
- [8] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [9] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 87–100.
- [10] GREENWALD, M., AND CHERITON, D. R. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), USENIX Association, pp. 123–136.
- [11] HENNESSY, J. P., OSISEK, D. L., AND SEIGH II, J. W. Passive serialization in a multitasking environment. Tech. Rep. US Patent 4,809,168 (lapsed), US Patent and Trademark Office, Washington, DC, February 1989.
- [12] HERLIHY, M. Implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* 15, 5 (November 1993), 745–770.
- [13] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing* (October 2002), pp. 339–353.
- [14] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)* (July 2003), pp. 92–101.

- [15] HSIEH, W. C., AND WEIHL, W. E. Scalable reader-writer locks for parallel systems. Tech. Rep. MIT/LCS/TR-521, MIT Laboratory for Computer Science, Cambridge, MA, 1991.
- [16] JACOBSON, V. Avoid read-side locking via delayed free. Verbal discussion, September 1993.
- [17] JOHN, A. Dynamic vnodes – design and implementation. In *USENIX Winter 1995* (New Orleans, LA, January 1995), USENIX Association, pp. 11–23.
- [18] KUNG, H. T., AND LEHMAN, Q. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems* 5, 3 (September 1980), 354–382.
- [19] MANBER, U., AND LADNER, R. E. Concurrency control in a dynamic search structure. Tech. Rep. 82-01-01, Department of Computer Science, University of Washington, Seattle, Washington, January 1982.
- [20] MANBER, U., AND LADNER, R. E. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems* 9, 3 (September 1984), 439–455.
- [21] MCKENNEY, P. E. *Pattern Languages of Program Design*, vol. 2. Addison-Wesley, June 1996, ch. 31: Selecting Locking Designs for Parallel Programs, pp. 501–531.
- [22] MCKENNEY, P. E. Practical performance estimation on shared-memory multiprocessors. In *Parallel and Distributed Computing and Systems* (Boston, MA, November 1999), pp. 125–134.
- [23] MCKENNEY, P. E. Using RCU in the Linux 2.5 kernel. *Linux Journal* 1, 114 (October 2003), 18–26.
- [24] MCKENNEY, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: <http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf> [Viewed October 15, 2004].
- [25] MCKENNEY, P. E. RCU vs. locking performance on different CPUs. In *linux.conf.au* (Adelaide, Australia, January 2004). Available: <http://www.linux.org.au/conf/2004/abstracts.html#90> <http://www.rdrop.com/users/paulmck/RCU/lockperf.2004.01.17a.pdf> [Viewed June 23, 2004].
- [26] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-copy update. In *Ottawa Linux Symposium* (July 2001). Available: <http://www.linuxsymposium.org/2001/abstracts/readcopy.php> http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf [Viewed June 23, 2004].
- [27] MCKENNEY, P. E., SARMA, D., ARCANGELI, A., KLEEN, A., KRIEGER, O., AND RUSSELL, R. Read-copy update. In *Ottawa Linux Symposium* (June 2002), pp. 338–367. Available: http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz [Viewed June 23, 2004].
- [28] MCKENNEY, P. E., AND SLINGWINE, J. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings* (Berkeley CA, February 1993), USENIX Association, pp. 295–306.
- [29] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.
- [30] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architecture* (August 2002), pp. 73–82.

- [31] MICHAEL, M. M. 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* (August 2002), pp. 21–30.
- [32] MOLNAR, I., AND MILLER, D. S. brlock. Available: http://www.tm.kernel.org/pub/linux/kernel/v2.3/patch-html/patch-2.3.49/%linux_include_linux_brlock.h.html [Viewed September 3, 2004], March 2000.
- [33] MORRIS, J. Recent developments in SELinux kernel performance. Available: http://www.livejournal.com/users/james_morris/2153.html [Viewed December 10, 2004], December 2004.
- [34] PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. Optimistic incremental specialization: Streamlining a commercial operating system. In *15th ACM Symposium on Operating Systems Principles (SOSP'95)* (Copper Mountain, CO, December 1995), pp. 314–321.
- [35] PUGH, W. Concurrent maintenance of skip lists. Tech. Rep. CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.
- [36] SARMA, D. [patch] rcu for low latency (experimental). <http://marc.theaimsgroup.com/?l=linux-kernel&m=108003746402892&w=2>, March 2004.
- [37] SARMA, D. Re: [patch] rcu for low latency (experimental). <http://marc.theaimsgroup.com/?l=linux-kernel&m=108016474829546&w=2>, March 2004.
- [38] SARMA, D., AND MCKENNEY, P. E. Making rcu safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)* (June 2004), USENIX Association, pp. 182–191.
- [39] SATYANARAYANAN, M. Scalable, secure, and highly available distributed file access. *IEEE Computer* 23, 5 (1990), 9–21.
- [40] SEIGH II, J. W. Read copy update. email correspondence, March 2003.
- [41] SLINGWINE, J. D., AND MCKENNEY, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Tech. Rep. US Patent 5,442,758, US Patent and Trademark Office, Washington, DC, August 1995.
- [42] SLINGWINE, J. D., AND MCKENNEY, P. E. Method for maintaining data coherency using thread activity summaries in a multicomputer system. Tech. Rep. US Patent 5,608,893, US Patent and Trademark Office, Washington, DC, March 1997.
- [43] SLINGWINE, J. D., AND MCKENNEY, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Tech. Rep. US Patent 5,727,209, US Patent and Trademark Office, Washington, DC, March 1998.
- [44] SLINGWINE, J. D., AND MCKENNEY, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Tech. Rep. US Patent 6,219,690, US Patent and Trademark Office, Washington, DC, April 2001.
- [45] SNAMAN, W. E., AND THIEL, D. W. The VAX/VMS distributed lock manager. *Digital Technical Journal* 5 (September 1987), 29–44.

- [46] SOULES, C. A. N., APPAVOO, J., HUI, K., DA SILVA, D., GANGER, G. R., KRIEGER, O., STUMM, M., WISNIEWSKI, R. W., AUSLANDER, M., OSTROWSKI, M., ROSENBERG, B., AND XENIDIS, J. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003), USENIX Association, pp. 141–154.
- [47] SPRAUL, M. [rfc] 0/5 rcu lock update. Available: <http://marc.theaimsgroup.com/?l=linux-kernel&m=108546407726602&w=2> [Viewed June 23, 2004], May 2004.
- [48] STEINER, J. Re: [lse-tech] [rfc, patch] 1/5 rcu lock update: Add per-cpu batch counter. Available: <http://marc.theaimsgroup.com/?l=linux-kernel&m=108551764515332&w=2> [Viewed June 23, 2004], May 2004.
- [49] TAY, Y. C. *Locking Performance in Centralized Databases*. Academic Press, 1987.
- [50] TORVALDS, L. Linux 2.5.43. Available: <http://marc.theaimsgroup.com/?l=linux-kernel&m=103474006226829&w=2> [Viewed June 23, 2004], October 2002.
- [51] TORVALDS, L. Linux 2.6. Available: <ftp://kernel.org/pub/linux/kernel/v2.6> [Viewed June 23, 2004], August 2003.