Implementation and Performance of Read-Copy Update

Paul E. McKenney
Sequent Computer Systems, Inc.
March 28, 1998

1 Abstract

Read-copy update is an update discipline that uses execution history in order to provide extremely low-overhead algorithms for concurrent access to read-mostly data structures [McKenney98a]. In particular, read-copy update is able to efficiently determine that a data structure can no longer be referenced by any other CPU or thread. This determination makes possible an efficient and effective implementation for a key step of several previously proposed concurrent-update algorithms [Kung80, Manber84, Pugh90]. This paper gives a detailed description of an implementation of read-copy update, and analyzes the performance of that implementation on both uniform and non-uniform memory-access architectures.

2 Introduction

Increases in CPU-core instruction-execution rate are expected to continue to outstrip reductions in global latency for large-scale multiprocessors [Hennessy91, Stone91, Burger96]. This trend will cause global lock and synchronization operations to continue becoming more costly relative to instructions that manipulate local data. This situation will continue to motivate the use of more specialized but less expensive locking designs.

Read-copy update forms the basis for locking designs for specialized, but commonly occurring, situations. The key observation behind read-copy update is that many software applications and systems contain *quiescent states* where the program is not accessing any data structures. Any access made after a quiescent state makes no assumptions about the state of the data preceding the quiescent state.

Of course, a parallel program must consider *all* CPUs or threads. Therefore, a time period during which all CPUs or threads have passed through at least one quiescent state is a *quiescent period*. Any change made before the start of a quiescent period is guaranteed to be visible to all CPUs or threads after the end of that quiescent period. This property makes it possible to design concurrent algorithms using many fewer explicit synchronization operations. In some cases, it is possible to eliminate synchronization operations altogether.

However, if read-copy update is to be useful, it must be able to identify quiescent periods extremely efficiently. Read-copy update accomplishes this by maintaining a carefully chosen *summary of thread activity*. This paper describes an extremely low-overhead implementation of the summary of thread activity that has been used in production in Sequent's Dynix/ptx since 1994, and that was updated for cache-coherent non-uniform memory access (CC-NUMA) architectures in 1996. Other possible implementations, along with a more detailed discussion of read-copy update concepts, may be found in a companion paper [McKenney98a].

Section 3 introduces a methodology for analyzing performance of algorithms running on CC-NUMA architectures. Section presents pseudo-code for the implementation of read-copy update. Section 5 presents performance analysis, Section 6 presents related work, and Section 7 presents summary and conclusions.

3 CC-NUMA Memory-Latency Model

Memory latency is the dominating factor in low-contention lock-overhead analysis due to the high cost of memory accesses compared to instruction execution times. Caches can cause memory latency to vary dynamically depending on order of accesses and updates.

Figure 1 shows the different locations that a cacheline can reside in relative to a given CPU in a CC-NUMA system. As shown in the figure, a CC-NUMA system is composed of several modules, called *quads*, that contain both CPUs and memory. Data residing nearer to a given CPU will have shorter access latencies. As the figure shows, data that is already in a given CPU's L2 cache may be accessed with latency t_f. Data located elsewhere on the quad may be accessed with latency t_m, while data located on other quads may be accessed with latency t_s.

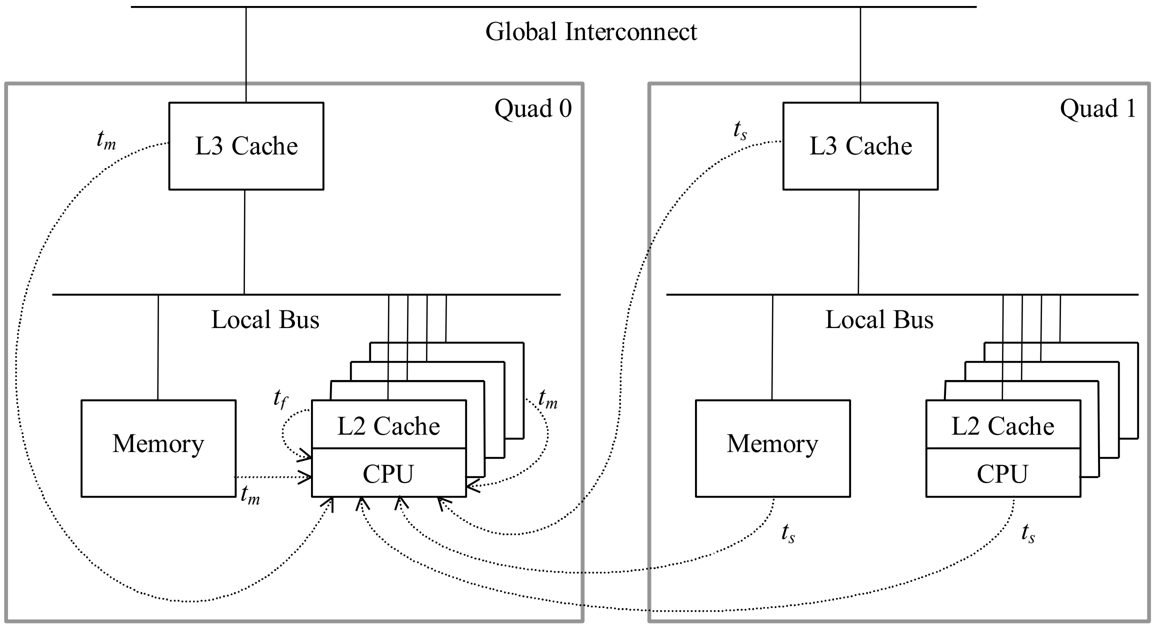

Figure 1: CC-NUMA Memory Latency

Once a given data item has been accessed by a CPU, it is cached in that CPU's L2 cache. If the data's *home* is in some other quad's memory, then it will also be cached in the accessing CPU's quad's L3 cache. In both cases, the caching allows subsequent accesses from the same CPU to proceed with much lower latency. Data that has been previously accessed by a given CPU is assumed to reside in either that CPU's L2 cache (with access latency t_f) or in its L1 cache (with access latency approximated by zero). In other words, at low contention, we assume that there is insufficient memory pressure to force data out of a CPU's L2 cache. However, data is assumed to remain in the much smaller L1 caches only if that data was accessed very recently, for example, during the execution of a single function.

A CC-NUMA system contains n quads and m CPUs per quad (two and four, respectively, in the example shown in the figure). To simplify analysis, each quad is assumed to contain the same number of CPUs. In addition, the analysis makes the following assumptions:

1) Contention is low and lock-hold times are vanishingly small. This means that the probability of two CPUs attempting to acquire the same lock at the same time is vanishingly small, as is probability of one CPU attempting to acquire a lock that another CPU already holds.

2) CPUs acquire locks at random intervals. This means that when a given CPU acquires a lock, that lock was last held, with equal probability, by any of the CPUs. Updates and read-only accesses are assumed to occur randomly with probability f and $1-f$, respectively

3) The overhead of instructions executed wholly within the microprocessor core is insignificant compared to the overhead of data references that miss the cache.

4) The CPU is assumed to have a single-cycle-access on-chip cache. This cache is considered to be part of the CPU core, and for purposes of these derivations is called the "L1 cache". Instruction fetches and stack references (function calls and returns, accesses to auto variables) are assumed to hit this L1 cache.

5) Cache pressure is assumed low, so that a variable that resides in a given cache remains there until it is invalidated by a write from some other CPU.

6) CC-NUMA systems with multiple levels of off-chip cache are assumed to have geometric memory latencies so that $t_s/t_m=t_m/t_f$. This assumption is generally very nearly true in practice, and greatly simplifies the equations. However, substitution of r for t_f, t_m, and t_s is done as a last step so that the equations may be easily adapted to any particular real-world situation.

Note that the actual behavior is depends critically on cache state, so actual results can deviate significantly from the analytic results presented in this paper. For example, if the L2 cache was fully utilized, the added memory pressure resulting from the larger size of higher-performance locking primitives might well overwhelm their performance benefits. Therefore, analytic results should be used as guidelines or rules of thumb, and should be double-checked by measuring the actual performance of the running program. Nevertheless, results obtained from this model have proven quite useful in practice.

Error: Reference source not found summarizes the symbols used in the derivations.

Symbol	Definition
f	Fraction of lock acquisitions that require exclusive access to the critical section.
m	Number of CPUs per quad in NUMA systems, and number of CPUs in system for SMP systems.
n	Number of quadsin NUMA systems, set to 1 in SMP systems.
r	Ratio of t_s to t_f.
t_c	Time required to access fine-grained hardware clock.
t_f	Time required to complete a "fast" access that hits the CPU's L2 cache.
t_m	Time required to complete a "medium" access that hits memory or a cache shared among a subset of the CPUs. This would be the latency of local memory or of the remote cache in CC-NUMA systems.
t_s	Time required to complete a "slow" access that misses all local caches.

Table 1: Nomenclature for Lock Cost Derivation

4 Read-Copy Update Implementation

This implementation considers a CPU to be in a quiescent state if that CPU is in the idle loop, executing in user mode, offline (halted), or performing a context switch. The summary of thread activity counts these quiescent states on a per-CPU basis.[1] At the beginning of a quiescent period, the CPU snapshots its counters. Later, when a CPU observes that at least one of its counters differs from its snapshot, it can conclude that it has passed through at least one quiescent state. The summary of thread activity also uses a per-quad bitmask to track which CPUs on the corresponding quad have not yet passed through a quiescent state, as well as a global bitmask to track which quads contain CPUs that have not yet passed through a quiescent state.

Note that the quiescent states are events that must be counted in any case, or that occur when the CPU is not doing anything useful. Examples of the former include system calls, traps, and context switches, which must be counted in order to support performance-analysis and debug activities. Examples of the latter include passes through the idle loop and taking CPUs out of service. The pre-existing counts of these events are used to implement a quiescent-period-detection algorithm that incurs little cost beyond that which is required just to count the events.

The basic outline of this algorithm is as follows:

1. An entity needing to wait for a quiescent period enqueues a callback onto a per-CPU list.
2. Some time later, this CPU informs all other CPUs of the beginning of a quiescent period.

[1] Because this implementation runs within a non-preemptive kernel, and because one of the quiescent states is a context switch, the implementation may track CPUs instead of threads or processes. Preemptive kernels or applications either would need a cheap way of suppressing preemption, or would choose a different set of quiescent states and track threads or processes instead of CPUs. The former choice may be attractive in environments preemption may be suppressed by setting a bit in a machine register or in a memory location, as is the case in a number of commercial operating systems.

3. As each CPU learns of the new quiescent period, it takes a snapshot of its quiescent-state counters.
4. Each CPU periodically compares its snapshot against the current values of its quiescent-state counters. As soon as any of the counters differ from the snapshot, the CPU records the fact that it has passed through a quiescent state.
5. The last CPU to record that it has passed through a quiescent state also records the fact that the quiescent period has ended.
6. As each CPU learns that the quiescent period has ended, it executes any of its callbacks that were waiting for the end of this particular quiescent period.

Steps 2, 3, 4, and 6 all involve time delays. These time delays may be tuned to trade off overhead against the wall-clock time required to identify a quiescent period. This is a classic CPU-memory tradeoff: shortening the quiescent-period-identification interval too much will consume excessive CPU time, while lengthening it too much will increase the amount of memory that cannot be used until the end of the current quiescent period.

The actual implementation is complicated by the following issues:

1. Proper handling of callbacks that are enqueued while a quiescent period is in progress. These callbacks must wait for the completion of a subsequent quiescent period.
2. Efficient notification of the beginning and ending of a quiescent period.
3. Efficient placement and use of state variables in a CC-NUMA environment.
4. Batching of callbacks in order to make best use of each quiescent period.

These four issues are handled as follows:

1. Each CPU maintains a queue of callbacks awaiting the end of the current quiescent period (rclockcurlist) as well as a separate queue of callbacks awaiting the end of a later quiescent period (rclocknxtlist). Each quiescent period is identified by a *generation number*. Each CPU tracks the generation number corresponding to the callbacks in its rclockcurlist. Since one CPU can start a new quiescent period before another CPU is aware that the previous period has ended, different CPUs can be tracking different generation numbers.
2. CPUs could use interrupts or polling to wait for the beginning and end of quiescent periods, but either approach would waste CPU time. Instead, our implementation checks for state changes from within an existing periodic interrupt that is used to accumulate CPU utilization statistics, collect profiling data, and handle timeouts. This approach incurs minimal overhead and imposes acceptably small delays.
3. In order to promote locality in a CC-NUMA environment, certain state variables are replicated on a per-CPU and a per-quad basis, as shown in Table 2.
4. Batching of callbacks is accomplished by accumulating callbacks while the current quiescent period is in progress. The heavier the read-copy update load, the larger the batches will be, and the smaller the per-callback overhead will be.

Section 4.1 presents state variables used by the implementation, Section 4.2 presents an overview of the pseudo-code, Section 4.3 shows the flow of callbacks through the read-copy subsystem, Section 4.4 presents the pseudo-code itself, and Sections 4.5 and 4.6 each walk the pseudo-code through an example.

4.1 Pseudo-Code State Variables

The state variables for the quad-aware implementation of read-copy update are shown in Table 2.

State Variable Name	Description
rcc_mutex ‡	Lock that guards global and per-quad state.
Generation Numbers	Each quiescent period is identified by a “generation number”.
rcc_maxgen ‡	Largest rclock generation requested thus far.
rcc_curgen ‡	Generation that is currently being serviced. If rcc_curgen is greater than rcc_maxgen, then no generation is being serviced and the rclock subsystem is idle.
pq_rcc_curgen †	Per-quad copy of rcc_curgen, updated in lock-step.
rclockgen	Earliest generation that this CPU needs to be completed.
Bitmasks	These bitmasks track which CPUs and quads need to pass through a quiescent state in order for the current rclock generation to complete. These masks are zero when the rclock subsystem is idle.
rcc_needctxtmask ‡	The set of quads that have CPUs that still need to pass through a quiescent state.
pq_rcc_needctxtmask †	The set of CPUs on this quad that still need to pass through a quiescent state.
Statistics	System statistics that are monitored in order to detect when a given CPU has passed through a quiescent state. Each of these variables is replicated on a per-CPU basis.
cswtchctr	Number of context switches by this CPU.
v_syscall	Number of system calls by this CPU.
usertrap	Number of traps from user state by this CPU.
syncpoint	Number of passes through the idle loop plus the number of times that the CPU was yielded by kernel code (whether or not there was some other process to take the CPU).
Statistics Snapshots	Per-CPU copies of the per-CPU statistics. A snapshot is taken on each CPU as soon as that CPU becomes aware that there is another rclock generation in progress.
rclockcswtchctr	Number of context switches by this CPU.
rclocksyscall	Number of system calls by this CPU.
rclockusertrap	Number of traps from user state by this CPU.
rclocksyncpoint	Number of passes through the idle loop plus the number of voluntary in-kernel yields.
Callback Lists	These per-CPU lists hold rclock callbacks, one list for each state that a particular callback can be in. There is also a set of global lists to hold callbacks for CPUs that have recently been taken offline, but these do not enter into the main flow of the algorithm. There is also a global set of lists that holds the callbacks for CPUs that were offlined with outstanding callbacks.
rclocknxtlist	Callbacks waiting for the previous generation to end so that they may be moved onto the current list. This list is a staging area.
rclockcurlist	Callbacks waiting for the current generation to end, at which point they will be ready to process.
rclockintrlist	Callbacks that are ready to process. These are placed on a list and processed by a software interrupt handler (rc_intr()) so that they may be run at low SPL.
rcc_nxtlist	Global equivalent of the per-CPU rclocknxtlist
rcc_curlist	Global equivalent of the per-CPU rclockcurlist
rcc_intrlist	Global equivalent of the per-CPU rclockintrlist

†: Per-quad state variable.
‡: Global state variable.

Table 2: State Variables for rclock Implementation

4.2 Pseudo-Code Overview

The pseudo-code for the quad-aware read-copy update implementation is divided as follows:

1) hardclock(): This function is invoked by a per-CPU clock interrupt. It contains a code fragment that invokes rc_chk_callbacks() when there is a possibility that callbacks could advance or that a CPU might be stalled. This possibility is indicated either by pq_rcc_needctxtmask indicating that this CPU needs to pass through a quiescent state, by pq_rcc_curgen indicating that the quiescent period for any callbacks in rclockcurlist has ended, or by rclockcurlist being empty and rclocknxtlist being nonempty.

2) rc_adv_callbacks(): Advances callbacks from the rclocknxtlist to the rclockcurlist and from the rclockcurlist to the rclockintrlist as quiescent periods complete. Also calls rc_intr() via software interrupt as needed to invoke callbacks newly placed into rclockintrlist and calls rc_reg_gen() if needed to register the presence of a new set of callbacks in rclockcurlist.

3) rc_callback(): Registers a new read-copy callback by adding it to the rclocknxtlist. Note that this function automatically batches any callbacks that arrive during a particular quiescent period. This batching greatly improves read-copy-update performance.

4) rc_chk_callbacks(): Calls rc_adv_callbacks() in order to advance callbacks from rclocknxtlist to rclockcurlist and from rclockcurlist to rclockintrlist. Snapshots the statistics variables when it notices that a new quiescent period has started. Checks the current statistics against the snapshot in order to determine if the current CPU has passed through a quiescent state since the beginning of the current generation. Calls rc_cleanup() when it determines that the current CPU has passed through a quiescent state.

5) rc_cleanup(): Determines when all CPUs have agreed that the current quiescent period can end, updating the generation numbers if so. Calls rc_reg_gen() and rc_adv_callbacks() to start the next quiescent period, but only if there are callbacks waiting for another quiescent period.

6) rc_intr(): Dispatches callbacks in rclockintrlist. These callbacks have progressed through a full quiescent period.

7) rc_reg_gen(): Tells the read-copy subsystem of a request for a quiescent period. If this is the first request for a given quiescent period, and if there is not currently a quiescent period in progress, initiate one by setting up rcc_maxgen and initializing the bitmasks.

8) rc_onoff(): Moves callbacks from the per-CPU rclocknxtlist, rclockcurlist, and rc_intrlist to the global rcc_nxtlist, rcc_curlist, and rcc_intrlist in preparation for the current CPU's being taken out of service.

The call graph for these functions is as shown in Figure 2.

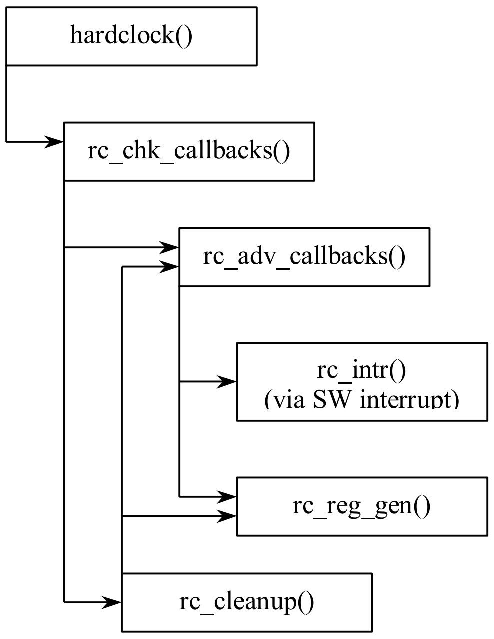

Figure 2: Read-Copy Call Graph

4.3 Flow of Callbacks Through Read-Copy Update Subsystem

New callbacks are injected into the system by rc_callback(). While the callbacks are awaiting invocation by rc_intr(), they are kept on linked lists, and flow through the system as shown in Figure 3. The solid boxes denote lists, while the dotted boxes denote functions. The rcc_nxtlist, rcc_curlist, and rcc_intrlist are global counterparts to the per-CPU rclocknxtlist, rclockcurlist, and rclockintrlist. The three global lists are only used if a CPU is taken out of service with read-copy callbacks outstanding. The rc_onoff() function moves the callbacks from the CPU's lists to the global lists to allow the callbacks to be properly processed after the CPU is taken out of service.

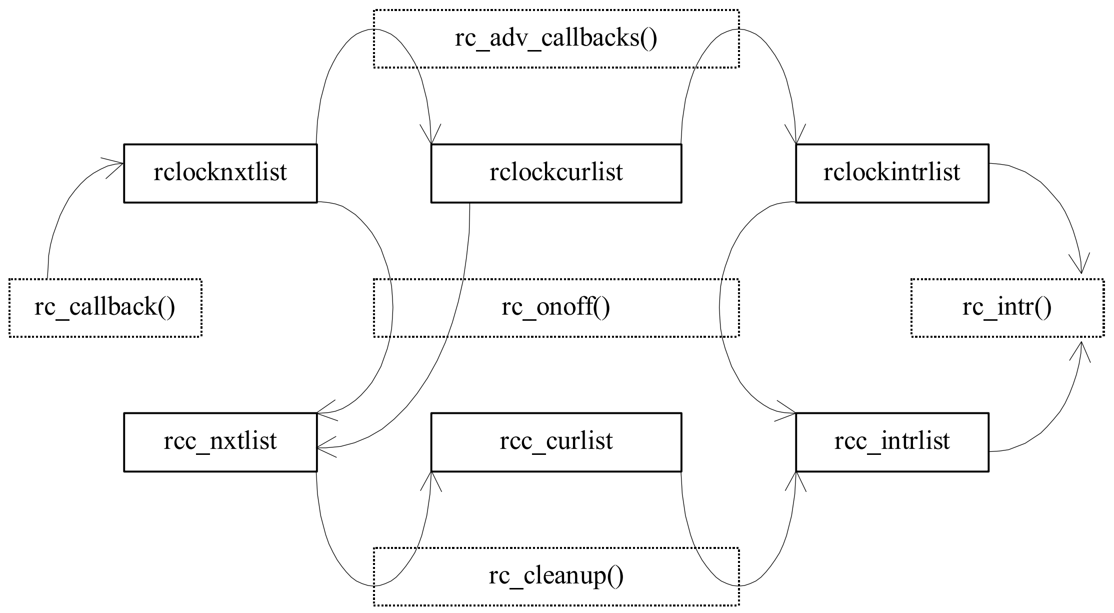

Figure 3: Flow of Callbacks

The actual implementation also includes functions to check for CPUs taking too long to reach a quiescent state. Such checks are needed to bound response times in face of high-priority long-running interrupt handlers or threads. These functions are omitted from this presentation, as they do not come into play in the common case, and as they are easy to understand given a complete understanding the main part of the implementation.

4.4 Pseudocode

The pseudocode includes performance characterizations. These are broken down into the four categories defined in the performance derivation:

1) per-hardclock() costs. These are incurred on every execution of the per-CPU scheduling-clock interrupt, whether or not the read-copy mechanism is actively doing work. An isolated read-copy batch requires two hardclock() invocations on each CPU to propagate through the read-copy subsystem. More invocations would be required if CPUs fail to pass through a quiescent state during one of the intervals between two successive hardclock() invocations. However, this will be a rare event given typical commercial workloads, and will therefore be ignored.
2) per-generation costs. These are incurred during each read-copy generation.
3) per-batch costs. These are incurred during each read-copy batch. Each CPU might or might not have an active batch during a given read-copy generation, and per-batch costs are incurred only by those CPUs that have a batch during a given generation. Note that per-batch costs are amortized over callbacks that are part of a given batch.
4) per-callback costs. These are incurred for each and every read-copy callback.

To review the nomenclature, t_s, t_m, and t_f are slow-, medium-, and fast-latency accesses, respectively. The number of quads is denoted by n and the number of CPUs per quad by m. The number of CPUs in the system is therefore nm.

4.4.1 hardclock() Code Fragment

Invoke rc_chk_callbacks() if any of the following conditions are met:

1) rclocknxtlist is not empty, but rclockcurlist is empty. In this case, the callbacks in rclocknxtlist need to be moved to rclockcurlist, and a generation needs to be registered for them. The cost of this step is t_f because all the variables accessed are local to this CPU and fit into the same cache line.
2) rclockcurlist is not empty, and the generation corresponding to this list (rclockgen) has ended. In this case, the callbacks need to be moved to rclockintrlist, and rc_intr() must be invoked in order to dispatch them. All accesses in this step are cache hits and therefore of insignificant cost.
3) A CPU has stalled. In this case, a stall-warning message must be printed. However, this is outside of the main read-copy update functionality, and therefore will not be considered further, except for the cost, which is t_c to access the clock and either t_f, t_s, or t_m to access the value with which to compare it during a generation. There is a higher cost at the beginning of the generation because the clock value has been updated, either by a CPU on some other quad (for cost t_s), by some other CPU on this quad (for cost t_m), or by this same CPU (for cost t_f). Weighting these costs by the number of CPUs in each of these three situations yields a systemwide per-generation cost of $t_f + t_m(nm+m-1)+t_s m(n-1)$. However, this cost replaces the normal per-hardclock cost of t_f, so the net per-generation cost over and above the normal per-hardclock cost is $(1-nm)t_f + t_m(nm+m-1)+t_s m(n-1)$.
4) The bit in pq_rcc_needctxtmask corresponding to this CPU is set. In this case, rc_chk_callbacks() checks to see if this CPU has passed through a quiescent state yet, thereby ending a generation. All accesses in this step are cache hits and therefore of insignificant cost.

The cost of this code fragment is $nm(2t_f+t_c)$ per hardclock() and of $(1-nm)t_f + t_m(nm+m-1)+t_s m(n-1)$ per generation.

4.4.2 rc_adv_callbacks()

Advance callbacks from the rclocknxtlist to the rclockcurlist and the rclockintrlist as the generation number advances, as follows:

1) Increment the rcc_nchk statistic. The cost of this step is t_f, and is incurred once per hardclock(). Invocations from other locations are counted at the point of call.
2) If rclockcurlist is not empty and rclockgen has completed (i.e., pq_rcc_curgen exceeds it), then the generation has ended for the callbacks on this list, and they need to be prepared to be dispatched:
   a) Append rclockcurlist to rclockintrlist. All accesses in this step hit the cache, and are therefore of insignificant cost.
   b) Initialize rclockcurlist to be empty. All accesses in this step hit the cache, and are therefore of insignificant cost.
   c) If rclockcurlist was initially empty, send this CPU a software interrupt in order to cause rc_intr() to execute at low SPL. The cost of this step is approximated by t_s, and is incurred once per batch.
3) If rclockcurlist is empty, but rclocknxtlist is not, then the callbacks that have been waiting in rclocknxtlist may now be assigned a generation number and moved to rclockcurlist to wait for that generation to end:
   a) Move rclocknxtlist to rclockcurlist. All accesses in this step hit the cache, and are therefore of insignificant cost.
   b) Initialize rclocknxtlist to be empty. All accesses in this step hit the cache, and are therefore of insignificant cost.
   c) Acquire rcc_mutex. The cost of this step is t_s, and is incurred once per batch.
   d) Set rclockgen to rcc_curgen. All accesses in this step hit the cache, and are therefore of insignificant cost.
   e) If the read-copy subsystem is active (i.e., if rcc_curgen is less than or equal to rcc_maxgen), increment rclockgen. All accesses in this step hit the cache, and are therefore of insignificant cost.
   f) Invoke rc_reg_gen(rclockgen) to register this (possibly) new generation. Costs for rc_reg_gen() are accounted for in the description of that function.
   g) Release rcc_mutex. All accesses in this step hit the cache, and are therefore of insignificant cost.

The cost of the rc_adv_callbacks() function is t_f per hardclock() on each CPU, or nmt_f systemwide, and $2t_s$ per batch.

4.4.3 rc_callback()

Registers a new read-copy callback as follows:

1) Set the RCC_REGISTERED bit in the rcc_flags field in order to mark this callback as being registered. The cost for this step is t_f, and is incurred once per callback.
2) Count the new callback in rcc_nreg. The cost for this step is t_f, and is incurred once per callback.
3) Set the callback's rcc_next pointer to NULL. All accesses in this step hit the cache, and are therefore of insignificant cost.
4) If the current CPU is in the process of going offline, the callback must be registered in the global lists. Since this is not the normal course of events, it will not be considered further. However, the cost of checking is t_f, and is incurred once per callback.
5) Suppress interrupts.
6) Append the callback to rclocknxtlist. The cost of this step is $2t_f$, and is incurred once per callback.
7) Restore interrupts.

The cost of the rc_callback() function is $5t_f$ per callback.

4.4.4 rc_chk_callbacks()

Checks per-CPU statistics in order to determine when the current generation has ended from this CPU's viewpoint as follows:

1) Invoke rc_adv_callbacks() to advance callbacks through the lists as needed. The cost of invoking this function is accounted for in its definition.
2) If this CPU's bit in pq_rcc_needctxtmask is already clear, our work is done. Simply return to the caller. All accesses in this step hit the cache, and are therefore of insignificant cost.
3) If this hardclock() interrupt came from a kernel thread and rclockcswtchctr is set to the special "invalid" CSWTCHCTR_INVALID value, then this CPU has just now become aware of a new read-copy update generation. Snapshot the statistics and return. The cost of this step is $8t_f$ per CPU per generation.
4) If this hardclock() interrupt came from a kernel thread and if none of the statistics have changed since the last snapshot, this CPU has not yet passed through a quiescent state for the current read-copy update generation. Simply return to the caller. All accesses in this step hit the cache, and are therefore of insignificant cost.
5) Otherwise, this CPU has just passed through the quiescent state required for the current read-copy update generation.
6) Acquire rcc_mutex. The cost of this step is t_s per CPU per generation.
7) Invoke rc_cleanup() to check for the end of the read-copy update generation. Note that rc_cleanup() releases rcc_mutex. The cost of this step is accounted for in the definition of rc_cleanup().

The cost of rc_chk_callbacks() is $nm(8t_f+t_s)$ per generation.

4.4.5 rc_cleanup()

Determines when all CPUs agree that the current generation can end, and does cleanup actions. Note that rc_cleanup() assumes that the caller has acquired rcc_mutex, and that rc_cleanup() returns after having released rcc_mutex.

1) Clear this CPU's bit in the pq_rcc_needctxtmask. This costs t_m on each CPU per generation.
   a) If this CPU's bit in pq_rcc_needctxtmask was already cleared, we already cleaned up after this CPU:
      (1) Release rcc_mutex. All accesses in this step hit the cache, and are therefore of insignificant cost.
      (2) Return to the caller.
   b) If other CPU's bits in pq_rcc_needctxtmask are still set, then there are other CPUs on this quad that have yet to pass through a quiescent state:
      (1) Set rclockcswtchctr to the special "invalid" CSWTCHCTR_INVALID value. All accesses in this step hit the cache, and are therefore of insignificant cost.

        (2) Release rcc_mutex. All accesses in this step hit the cache, and are therefore of insignificant cost.
        (3) Return to the caller.
    c) If all CPU's bits in pq_rcc_needctxtmask are now cleared, then this is the last CPU on this quad to pass through a quiescent state. Proceed to the global level:
        i) Clear this quad's bit in rcc_needctxtmask. The cost of this step is t_s per quad per generation.
            (1) If all quad's bits in rcc_needctxtmask are now cleared, then this is the last CPU in the system to pass through a quiescent state. Continue with cleanup in top-level step 2 below.
            (2) If there are other quads with CPUs that have yet to pass through a quiescent state for the current generation:
                (a) Set rclockcswtchctr to the special "invalid" CSWTCHCTR_INVALID value. All accesses in this step hit the cache, and are therefore of insignificant cost.
                (b) Release rcc_mutex. All accesses in this step hit the cache, and are therefore of insignificant cost.
                (c) Return to the caller.
            (3) If this quad's bit was already cleared, panic.

2) Get here if this is the last CPU in the system to pass through a quiescent state during the current read-copy update generation. Do the following steps to end the current generation, and, if needed, start a new one.
3) Set rclockcswtchctr to the special "invalid" CSWTCHCTR_INVALID value. All accesses in this step hit the cache, and are therefore of insignificant cost.
4) Increment rcc_curgen to mark the end of the generation. All accesses in this step hit the cache, and are therefore of insignificant cost.
5) Set the pq_rcc_curgen variables for each quad to the new value of rcc_curgen. The cost of this step is $(n-1)t_s$, incurred once per generation.
6) Advance the global lists as appropriate. Since these lists come into play only if a CPU goes offline, they will not be discussed further.
7) Invoke rc_reg_gen(rc_maxgen) to start a new generation if needed (if rc_maxgen is less than rc_curgen, then no more generations are needed, and the read-copy-update subsystem will go idle). The cost of this step is accounted for in the definition of rc_reg_gen().
8) Invoke rc_adv_callbacks() to advance callbacks through the lists as needed. The cost of this step is accounted for in the definition of rc_adv_callbacks().

The cost of rc_cleanup() is $(2n-1)t_s+nmt_m$ per generation.

4.4.6 rc_intr()

Dispatch callbacks that have progressed through a full read-copy generation.

For each callback on the rclockintrlist, do the following:

1) Suppress interrupts.
2) Remove the callback from the list. The cost of this step is t_f, incurred once per callback.
3) Restore interrupts.
4) Clear the RCC_REGISTERED bit from the rcc_flags field. All accesses in this step hit the cache, and are therefore of insignificant cost.
5) Invoke the callback handler function.
6) If the handler returned CALLBACK_DEFER and deferral is permitted for this callback, requeue the callback.
7) Count the handler invocation in the rcc_nprc statistic. The cost of this step is t_f, incurred once per callback.

This function also handles callbacks placed on the global lists due to CPU online and offline, however, onlines and offlines do not occur in the common case.

The cost of rc_intr() is $2t_f$ per callback.

4.4.7 rc_reg_gen(rc_gen_t newgen)

Tell the read-copy subsystem of a request for a read-copy generation. If this is the first request for a given generation, and if there is not currently a generation in progress, initiate one.

1) If newgen is greater than rcc_maxgen, set rcc_maxgen to newgen. The cost of this step is t_s, incurred once per generation, and, in addition, t_s incurred once per batch. The former is the overhead of changing rcc_maxgen once per generation, and the latter is the overhead of checking rcc_maxgen once per batch.
2) If rcc_maxgen is less than rcc_curgen, and if the bits corresponding to each quad in rcc_needctxtmask are all clear, then either no one yet cares about the current read-copy generation, or the current generation has not yet ended. In the former case, there is no need to do anything. In the latter case, the last CPU to clear its bit will do the needed cleanup. So, in either case, simply return to caller! All accesses in this step hit the cache, and are therefore of insignificant cost.
3) Set rcc_needctxtmask to have a bit set for each quad. The cost of this step is t_s, incurred once per generation.
4) Set each quad's pq_rcc_needctxtmask to have a bit set for each CPU on that quad. The cost of this step is $(n-1)t_s$, incurred once per generation.
5) If this CPU's rclockcswtchctr is set to the special "invalid" CSWTCHCTR_INVALID value, snapshot the statistics. The cost of this step would be $8t_f$, incurred once per generation, however, this cost is already accounted for in rc_chk_callbacks().

The cost of rc_reg_gen() is $(n+1)t_s$ per generation plus t_s per batch.

4.5 Read-Copy Update State Walkthrough, Simple Example

Each row of Table 3 corresponds to a state that the read-copy update mechanism passes through. This table depicts this state from CPU 0's viewpoint for a simple case where a single callback is registered while the read-copy subsystem is otherwise idle.

		Generation Numbers				Bitmasks		Statistics				Statistics Snapshots				Callback Lists		
	Event	rcc_maxgen ‡	rcc_curgen ‡	pq_rcc_curgen †	rclockgen	rcc_needctxtmask ‡	pq_rcc_needctxtmask †	cswtchctr	v_syscall	usertrap	syncpoint	rclockcswtchctr	rclocksyscall	rclockusertrap	rclocksyncpoint	rclocknxtlist	rclockcurlist	rclockintrlist
1	Initial State	0	1	1	0	0x0	0x0	0	0	0	0	X	0	0	0			
2	rc_callback()	0	1	1	0	0x0	0x0	0	0	0	0	X	0	0	0	1/?		
3	syncpoint	0	1	1	0	0x0	0x0	0	0	0	1	X	0	0	0	1/?		
4	hardclock()	0	1	1	0	0x0	0x0	0	0	0	1	X	0	0	0	1/?		
5	rc_adv_callbacks()	0	1	1	1	0x0	0x0	0	0	0	1	X	0	0	0		1/1	
6	rc_reg_gen()	1	1	1	1	0x3	0xf	0	0	0	1	0	0	0	1		1/1	
7	syscall()	1	1	1	1	0x3	0xf	0	1	0	1	0	0	0	1		1/1	
8	swtch()	1	1	1	1	0x3	0xf	1	1	0	1	0	0	0	1		1/1	
9	usertrap()	1	1	1	1	0x3	0xf	1	1	1	1	0	0	0	1		1/1	
10	other CPUs…	1	1	1	1	0x1	0x1	1	1	1	1	0	0	0	1		1/1	
11	hardclock()	1	1	1	1	0x1	0x1	1	1	1	1	0	0	0	1		1/1	
12	rc_cleanup()	1	2	2	1	0x0	0x0	1	1	1	1	X	0	0	1		1/1	
13	rc_adv_callbacks()	1	2	2	1	0x0	0x0	1	1	1	1	X	0	0	1			1/1
14	rc_intr()	1	2	2	1	0x0	0x0	1	1	1	1	X	0	0	1			

†: Per-quad state variable.
‡: Global state variable.

Table 3: First Example State Transitions for Quad-Aware Read-Copy Lock Implementation

The first column contains a state number. The "Event" column names the event that caused the transition from the previous state to the current state. The "Generation Numbers", "Bitmasks", Statistics", and "Statistics Snapshots" column groups contain the values of each quantity at each state. Note that the value of rclockcswtchctr is listed as "X" when it contains the special invalid value. Each column in the "Callback Lists" column group is either empty (indicating that the corresponding list is empty) or contains two quantities separated by a "/". The first quantity is the number of callbacks on the corresponding list. The second quantity is the generation number of the callbacks, or "?" if the generation number is not yet known.

Global state variables are marked with a "‡", and per-quad state variables are marked with a "†". Per-CPU state variables are unmarked.

State 1 shows the initial state with the read-copy update subsystem idle (since rcc_curgen is greater than rcc_maxgen).

State 2 shows the effect of rc_callback(), which is simply to add the callback to the per-CPU rclocknxtlist.

State 3 shows the effect of an explicit yield of the CPU: only the corresponding counter is incremented, which has no effect since the read-copy subsystem is idle..

State 4 shows a hardclock(). The hardclock() function invokes rc_chk_callbacks(), which in turn invokes rc_adv_callbacks(), as shown in state 5. In state 5, the rc_adv_callbacks() function observes that rclockcurlist is empty and therefore moves the contents of rclocknxtlist to rclockcurlist, sets rclockgen to rcc_curgen, and invokes rc_reg_gen() to register the new generation, which is shown in state 6. In state 6, rc_reg_gen() sets rcc_maxgen to rclockgen, and, since rcc_maxgen is not less than rclockgen, initializes rcc_needctxtmask and pq_rcc_needctxtmask to have bits for each quad and CPU, respectively. Note that all instances of these state variables are initialized at this point. Finally, since rclockcswttchctr is set to the special "invalid" value, snapshot all the statistics.

States 4 through 6 are the first states that have referred to or modified shared state variables. This is one of the keys to understanding the efficiency of the read-copy lock mechanism under heavy load. Had there been other rc_callback() invocations preceding state 4, all of their callbacks would have moved as a unit, so that the overhead of updating shared state variables would have been amortized over the callbacks.

States 7, 8, and 9 show (redundant) events that increment statistics counters. Only one of these was needed for CPU 0 to agree to end the current generation. Again, these statistics counters are per-CPU state variables, so incrementing them incurs negligible overhead.

State 10 shows the effect of quiescent states and hardclock() processing on other CPUs. Note that the hardclock() processing includes the other functions that update read-copy state. After this processing, all but CPU 0's bits are clear, indicating that only CPU 0 has not yet agreed to the ending of the current generation.

State 11 shows hardclock() on CPU 0. The hardclock() function invokes rc_chk_callbacks(), which observes that the CPU has passed through at least one quiescent state, and therefore invokes rc_cleanup(), which, as shown in state 12, updates the global bitmasks accordingly. Since all bits are now clear, rc_cleanup() also observes that CPU 0 is the last CPU to record passage through the required quiescent state, and is therefore responsible for ending the read-copy generation. The rc_cleanup() function accomplishes this by setting rclockcswtchctr to the special invalid value, incrementing rcc_curgen, setting all instances of the per-quad pq_rcc_curgen state variable to rcc_curgen, and invoking rc_adv_callbacks(). As shown in state 13, rc_adv_callbacks() moves the callbacks from rclockcurlist to rclockintrlist and sends a software interrupt to invoke rc_intr(). Handling of this software interrupt by rc_intr() in state 14 invokes the callback's handler. Again, had there been several callbacks, the overhead of the interrupt (but not that of the callback-handler invocations) would have been amortized over the callbacks.

4.6 Read-Copy Update State Walkthrough, Second Example

The states in Table 4 show a more involved example with multiple callbacks from multiple CPUs in flight simultaneously. This example is again from CPU 0's viewpoint.

		Generation Numbers				Bitmasks		Statistics				Statistics Snapshots				Callback Lists		
	Event	rcc_maxgen ‡	rcc_curgen ‡	pq_rcc_curgen †	rclockgen	rcc_needctxtmask ‡	pq_rcc_needctxtmask †	cswtchctr	v_syscall	usertrap	syncpoint	rclockcswtchctr	rclocksyscall	rclockusertrap	rclocksyncpoint	rclocknxtlist	rclockcurlist	rclockintrlist
1	Initial State (Table 3)	1	2	2	1	0x0	0x0	1	1	1	1	X	0	0	1			
2	CPU 1 rc_callback()	1	2	2	1	0x0	0x0	1	1	1	1	X	0	0	1			
3	hardclock()	1	2	2	1	0x0	0x0	1	1	1	1	X	0	0	1			
4	syscall()	1	2	2	1	0x0	0x0	1	2	1	1	X	0	0	1			
5	CPU 1 hardclock()	2	2	2	1	0x3	0xf	1	2	1	1	X	0	0	1			
6	CPU 1 usertrap	2	2	2	1	0x3	0xf	1	2	1	1	X	0	0	1			
7	rc_callback()	2	2	2	1	0x3	0xf	1	2	1	1	X	0	0	1	1/?		
8	CPU 1 hardclock()	2	2	2	1	0x3	0xd	1	2	1	1	X	0	0	1	1/?		
9	rc_callback()	2	2	2	1	0x3	0xd	1	2	1	1	X	0	0	1	2/?		
10	hardclock()	3	2	2	3	0x3	0xd	1	2	1	1	1	2	1	1		2/3	
11	swtch()	3	2	2	3	0x3	0xd	2	2	1	1	1	2	1	1		2/3	
12	rc_callback()	3	2	2	3	0x3	0xd	2	2	1	1	1	2	1	1	1/?	2/3	
13	other quad-0 CPUs…	3	2	2	3	0x3	0x1	2	2	1	1	1	2	1	1	1/?	2/3	
14	hardclock()	3	2	2	3	0x2	0x0	2	2	1	1	X	2	1	1	1/?	2/3	
15	quad-1 CPUs…	3	3	3	3	0x3	0xf	2	2	1	1	X	2	1	1	1/?	2/3	
16	hardclock()	3	3	3	3	0x3	0xf	2	2	1	1	2	2	1	1	1/?	2/3	
17	swtch()	3	3	3	3	0x3	0xf	3	2	1	1	2	2	1	1	1/?	2/3	
18	Other CPUs…	3	3	3	3	0x1	0x1	3	2	1	1	2	2	1	1	1/?	2/3	
19	hardclock()	4	4	4	4	0x3	0xf	3	2	1	1	3	2	1	1		1/5	2/3
20	rc_intr()	4	4	4	4	0x3	0xf	3	2	1	1	3	2	1	1		1/5	
21	swtch()	4	4	4	4	0x3	0xf	4	2	1	1	3	2	1	1		1/5	
22	hardclock()	4	4	4	4	0x3	0xe	4	2	1	1	X	2	1	1		1/5	
23	Other CPUs…	4	5	5	4	0x0	0x0	4	2	1	1	X	2	1	1		1/5	
24	hardclock()	4	5	5	4	0x0	0x0	5	2	1	1	X	2	1	1			1/5
25	rc_intr()	4	5	5	4	0x0	0x0	5	2	1	1	X	2	1	1			

†: Per-quad state variable.
‡: Global state variable.

Table 4: Second Example State Transitions for Quad-Aware Read-Copy Lock Implementation

State 1 shows the initial state for the second example, which is the final state of the first example.

State 2 shows the effect of an rc_callback() executing on CPU 1. Since the effect of rc_callback() is strictly local to the CPU on which it executes, it results in no state change that is immediately visible to CPU 0.

State 3 shows the effect of the invocation of hardclock() and subordinate functions on CPU 0. Since there is not yet an work to be done from CPU 0's viewpoint, no changes are made to state variables visible to CPU 0.[2]

State 4 shows the effect of a system call on CPU 0, which increments the v_syscall counter.

[2] In reality, there are primitives that allow CPU 0 to access CPU 1's local state, but these primitives are quite expensive and therefore not used in read-copy update.

State 5 shows the effect of hardclock() and subordinate functions executing on CPU 1. This finally produces generation-number and bitmask changes visible to CPU 0.

State 6 shows the effect of a trap from user space on CPU 1. Since the per-CPU statistics are visible only on the CPU incrementing them, no changes are made the state variables visible to CPU 0.

State 7 shows the effect of an rc_callback() executing on CPU 0, which adds the callback to the per-CPU list.

State 8 shows the effect of hardclock() and subordinate functions executing on CPU 1. The only change visible to CPU 0 is CPU 1's clearing its bit from pq_rcc_neeedctxtmask, signaling that CPU 1 has passed through a quiescent state.

State 9 shows the effect of a second rc_callback() executing on CPU 0, which adds another callback to the per-CPU list.

State 10 shows the effect of hardclock() and subordinate functions executing on CPU 0. CPU 0 now becomes aware that it needs to pass through a quiescent state in order to satisfy CPU 1's rc_callback() request, and therefore snapshots its statistics. In addition, it becomes aware of its own rc_callback() requests, which will have to be satisfied by a later quiescent period.

State 11 shows the effect of a context switch on CPU 0, which provides the quiescent state CPU 1 needs.

State 12 shows the effect of another rc_callback() on CPU 0, which adds the callback to the per-CPU rclocknxtlist. Note that this callback cannot be immediately added to rclockcurlist, because the callbacks in the two lists belong to different generations.

State 13 shows the effect of passage through quiescent states followed by hardclock() and subordinate functions executing on the other quad-0 CPUs. The only effect visible to CPU 0 is the clearing of bits in pq_rcc_needctxtmask.

State 14 shows the effect of hardclock() and subordinate functions executing on CPU 0. At this point, CPU 0 becomes aware that it has passed through the quiescent state in state 11, and therefore clears its bit in pq_rcc_needctxtmask. Since it is the last CPU on quad 0 to clear its bit, it also clears the bit in the global rcc_needctxtmask.

State 15 shows the effect of the quad-1 CPUs passing through quiescent states and then executing hardclock() and subordinate functions. This marks the end of the quiescent period needed by CPU 1's callback. Since rcc_maxgen is not less than rcc_curgen, the last CPU realizes that another quiescent period must be started, and therefore sets the bitmasks.

State 16 shows the effect of hardclock() and subordinate functions executing on CPU 0. CPU 0 now notices that a new quiescent period has started, and therefore snapshots its statistics.

State 17 shows the effect of a context switch on CPU 0, while state 18 shows the effect of a quiescent state followed by a hardclock() on the other CPUs.

State 19 shows the effect of a hardclock() on CPU 0. CPU 0 moves rclockcurlist to rclockintrlist and moves rclocknxtlist to rclockcurlist. It also sets up state for a new quiescent period.

State 20 shows the effect of rc_intr() cleaning up the callbacks that just passed through a quiescent period.

States 21 through 25 show the final quiescent period handling CPU 0's last callback.

5 Performance of Read-Copy Update Implementation

There are four components to read-copy-update overhead:

1. per-hardclock() costs. These are incurred on every execution of the per-CPU scheduling-clock interrupt, whether or not the read-copy mechanism is actively doing work. An isolated read-copy batch requires two hardclock() invocations on each CPU to propagate through the read-copy subsystem. More invocations would be required if CPUs fail to pass through a quiescent state during

one of the intervals between two successive hardclock() invocations. However, this will be a rare event given typical commercial workloads, and will therefore be ignored.

2. per-generation costs. These are incurred during each read-copy generation.
3. per-batch costs. These are incurred during each read-copy batch. Each CPU might or might not have an active batch during a given read-copy generation, and per-batch costs are incurred only by those CPUs that have a batch during a given generation. Note that per-batch costs are amortized over callbacks that are part of a given batch.
4. per-callback costs. These are incurred for each and every read-copy callback.

Equation 1, Equation 2, Equation 3, and Equation 4 give the read-copy overhead incurred for each of these four components: per hardclock(), per generation, per batch, and per callback, respectively:

$$C_h = nmt_c + 3nmt_f$$ **Equation 1**

$$C_g = (3n + 2nm - m)t_s + (2nm + m - 1)t_m + (7nm + 1)t_f$$ **Equation 2**

$$C_b = 3t_s$$ **Equation 3**

$$C_c = 7t_f$$ **Equation 4**

These expressions are derived in the detailed pseudocode.

The best-case incremental cost of a read-copy callback, given that at least one other callback is a member of the same batch, is just C_c:

$$C_{bc} = 7t_f$$ **Equation 5**

The worst-case cost of an isolated callback is *m* times the per-hardclock cost plus the sum of the rest of the costs, as shown in Equation 6:

$$C_{wc} = (3n + 2nm - m + 3)t_s + (2nm + m - 1)t_m + (3nm^2 + 7nm + 8)t_f + nm^2t_c$$ **Equation 6**

Note that this worst case assumes that at most one CPU per quad passes through its first quiescent state for the current generation during a given period between hardclock() invocations. Given a more typical commercial workload, CPUs will usually pass through at least one quiescent state per interval between hardclock() invocations. More typical cases will be considered in the following paragraphs.

Typical costs may be computed assuming a system-wide Poisson-distributed inter-arrival rate of λ per generation. This computation is based on the Poisson-weighted costs given the expected number of batches and hardclock() interrupts per generation:

$$C_{typ} = \frac{\sum_{k=1}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} C_k}{1 - e^{-\lambda}}$$ **Equation 7**

where *$((\lambda^k e^{-\lambda})/k!)$* is the probability that k read-copy callbacks will be registered during a given generation if on average λ of them arrive per generation (this is just the Poisson distribution). Note that the *k=0* term of the Poisson distribution is omitted, since there is no read-copy overhead if there are no read-copy arrivals. The division by *$1-e^{-\lambda}$* corrects for the omission of the *k=0* term. The quantity C_k is defined as follows:

$$C_k = \frac{C_h + C_g + N_b(k)C_b + kC_c}{k}$$ **Equation 8**

This definition states that we pay the per-hardclock() and per-generation overhead unconditionally, that we pay the per-batch overhead for each of $N_b(k)$ batches, and that we pay per-callback overhead for each callback.

The expected number of batches $N_b(k)$ is given by the well-known occupancy-problem solution:

$$N_b(k) = nm\left(1-\left(1-\frac{1}{nm}\right)^k\right)$$ **Equation 9**

This is just the number of CPUs expected to have batches given *nm* CPUs and *k* read-copy updates in a given generation.

Substituting Equation 8 and Equation 9 into Equation 7:

$$C_{typ} = \frac{e^{-\lambda}}{1-e^{-\lambda}}\sum_{k=1}^{\infty}\frac{\lambda^k\left(C_h + C_g + nm\left(1-\left(1-\frac{1}{nm}\right)^k\right)C_b + kC_c\right)}{k!k}$$ **Equation 10**

And, finally, substituting Equation 1, Equation 2, Equation 3, and Equation 4 into Equation 10 yields the desired expression for the typical cost:

$$\frac{1}{e^{\lambda}-1}\sum_{k=1}^{\infty}\frac{\lambda^k\left[\begin{array}{c}(3n+5nm-m)r - 3nm\left(1-\frac{1}{nm}\right)^k r + \\ (2nm+m-1)\sqrt{r} + (10nm+7k+1) + nmt_c\end{array}\right]}{k!k}$$ **Equation 11**

These results are displayed in the following figures. The figures on the left are for symmetric multiprocessing (SMP) architectures, and the ones on the right are for CC-NUMA architectures. The SMP results are derived from the CC-NUMA results by considering an SMP system to be a CC-NUMA system with a single large quad, that is, by substituting 1 for *n*, *n* for *m*, t_s for t_m, and the appropriate latency value for t_c.

The traces are labeled as follows:

Label	Description
crw	Centralized reader-writer spinlock
drw	Distributed (cache-friendly) reader-writer spinlock [McKenney96]
qrw	Per-quad distributed reader-writer spinlock
sl	Simple spinlock
rcb	Best-case read-copy update (infinite degree of batching)
rcp	Poisson read-copy update arrivals with $\lambda=10$ updates per CPU per quiescent period
rcz	Poisson read-copy update arrivals with $\lambda=1$ updates per CPU per quiescent period
rcn	Poisson read-copy update arrivals with $\lambda=0.1$ updates per CPU per quiescent period
rcw	Worst-case read-copy update (isolated update)

Table 5: Trace Labels

The performance of the various spinlocks is derived in a companion paper [McKenney98], which also examines high-contention conditions and compares analytic results to measured values.

The first set of figures display read-copy update overhead as a function of the number of CPUs (see Table 5 for definitions of the trace labels). At these typical latency ratios and moderate-to-high update fractions, read-copy update outperforms the other locking primitives. Note particularly that the overhead of the non-worst-case read-copy overheads do not increase with increasing numbers of CPUs. This excellent scaling behavior is due to the batching capability of read-copy update. Although simple spinlock and centralized reader-writer spinlock also show good scaling, this good behavior is restricted to low-contention conditions. The poor behavior of these primitives under conditions of high contention is well known. Use of read-copy update is clearly quite advantageous in systems with large numbers of CPUs.

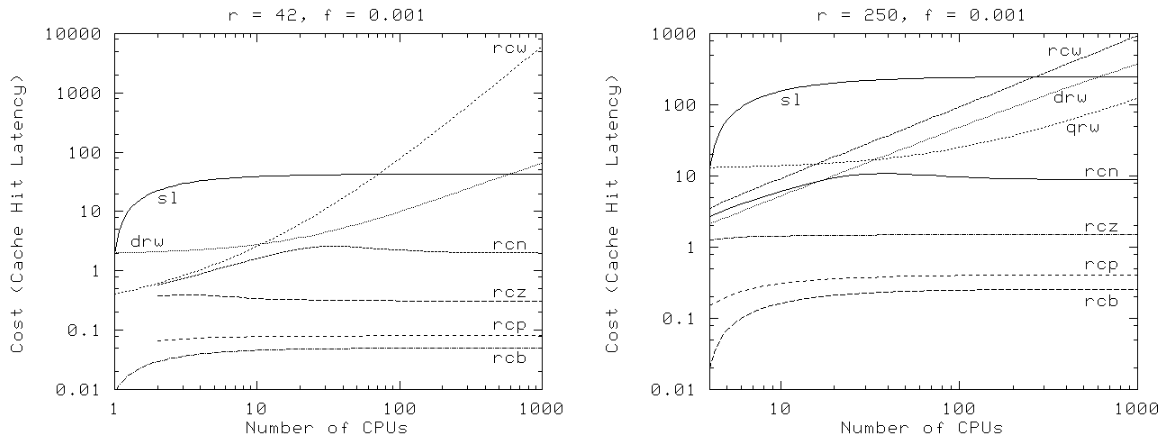

Figure 4: Read-Copy Overhead as Function of Number of CPUs

The next set of figures shows read-copy overhead as a function of the update fraction *f*. As expected, read-copy update performs best when the update fraction is low. Update fractions as low as 10^{-10} are not uncommon [McKenney98a].

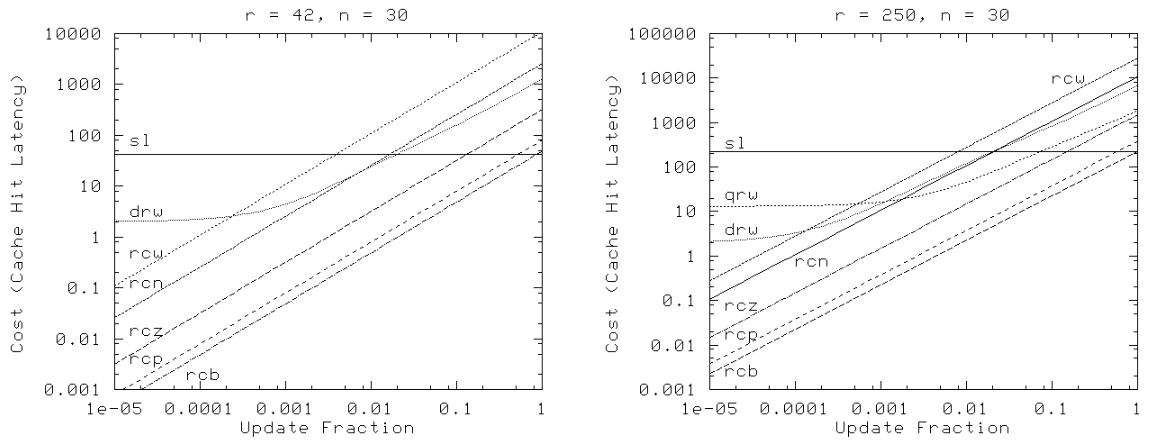

Figure 5: Read-Copy Overhead as Function of Update Fraction

The final set of figures shows read-copy overhead as a function of the memory-latency ratio *r*. The distributed reader-writer primitives have some performance benefit at high latency ratios, but this performance benefit is offset in many cases by high contention, by larger numbers of CPUs, or by lower update fractions, as shown in Figure 7.

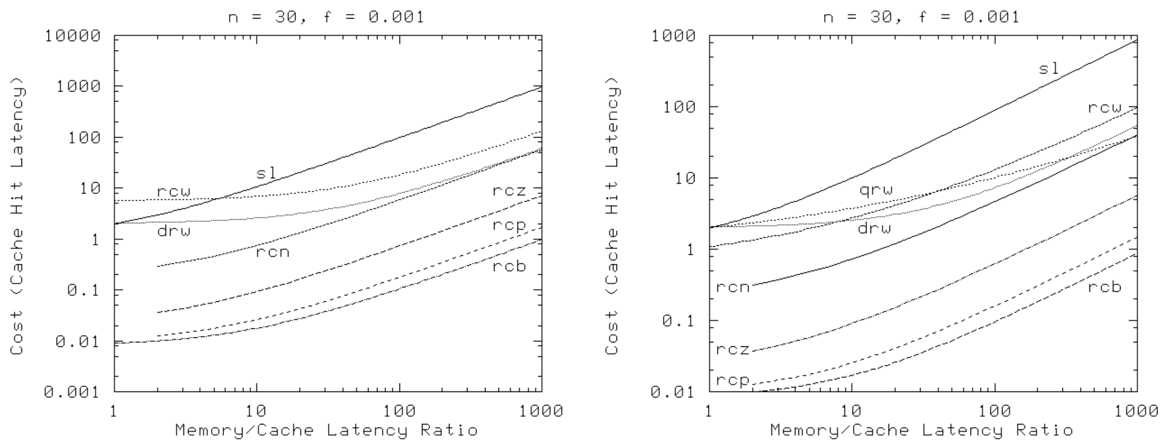

Figure 6: Read-Copy Overhead as Function of Memory-Latency Ratio

The situation shown in Figure 7 is far from extreme. As noted earlier, common situations can result in update fractions below 10^{-10}.

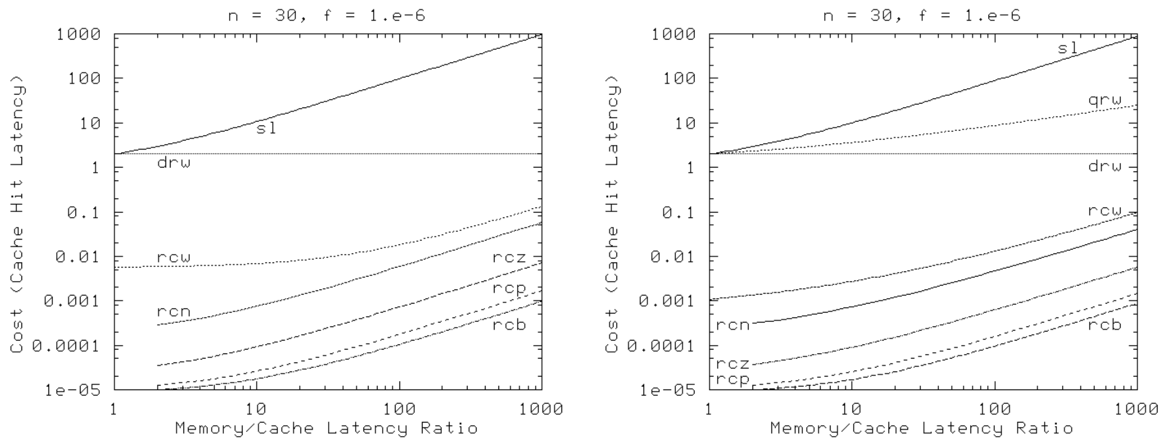

Figure 7: Read-Copy Overhead as Function of Memory-Latency Ratio for Lower Value of *f*

Note finally that all of these costs assume that the update-side processing for read-copy update is guarded by a simple spinlock. In cases where the update-side processing may use a more aggressive locking design (for example, if only one CPU, process, or thread alters the data), read-copy update will have an even greater performance advantage.

6 Related Work

Reader-writer spinlocks [MellorCrummey91] allow reading processes to proceed concurrently. However, updating processes may *not* run concurrently with either each other or with reading processes. In addition, reader-writer spinlocks exact significant synchronization overhead from reading processes. On the other hand, reader-writer spinlocks allow writers to block readers and vice versa, thereby avoiding stale data.

Wait-free synchronization [Herlihy93] allows reading and updating processes to run concurrently, but again exacts significant synchronization overhead. It further requires that memory used for a given type of data structure never be subsequently used for any other type of data structure. In addition, wait-free synchronization requires that reading as well as updating threads write to shared storage. On modern microprocessors, particularly in parallel systems, these writes will result in high-latency cache misses, which will increasingly limit performance as microprocessor clock frequencies rise. On the other hand, wait-free synchronization provides wait-free processing to updates as well as to reads and avoids stale data.

There are a number of timestamping and versioning concurrency-control systems that have some elements in common with read-copy update, however, none of these eliminate synchronization overhead to reading processes [Barghouti91]. So-called "chaotic relaxation" [Adams91] is similar to read-copy update in that it accepts stale data to obtain reduced synchronization overhead, however, chaotic relaxation requires highly structured data.

Manber and Ladner [Manber82] describe an algorithm that uses a technique similar to read-copy update. Their algorithm defers freeing a given node until all processes running at the time the node was removed from the tree have terminated. This does allow reading processes to run concurrently with updating processes, but does not lend itself to non-terminating processes such as those found in operating systems and server applications. In addition, they do not describe an efficient mechanism for maintaining the list of blocks awaiting deferred free.

Pugh [Pugh90] uses a deferred-free technique similar to that of Manber and Ladner, but notes that read-side state update can make it unnecessary to wait for process termination. However, Pugh also does not describe an efficient mechanism to maintain the list of blocks awaiting deferred free.

Kung and Lea [Kung80, Lea97] describe use of a garbage collector to manage the list of blocks awaiting deferred free. However, garbage collectors are often not available, and their overhead renders them infeasible in many situations. In particular, the traditional reference-counting approach incurs expensive memory writes for reading threads. As noted earlier, these writes have unacceptable overhead on modern cached microprocessors. Even when garbage collectors are available and when their overhead is acceptable, they do not address situations where some operation other than freeing memory is to be performed at the end of the quiescent period.

In contrast, read-copy update enables reading processes to avoid all synchronization operations, and in addition may be implemented efficiently even on systems with non-terminating processes. The fact that reading processes do not use synchronization operations reduces the potential for deadlock and livelock. Stale data may be avoided on a case-by-case basis, if needed, although such avoidance can require use of explicit and costly synchronization operations. The algorithm for detecting quiescent periods is described and its performance may be accurately measured and analytically evaluated.

7 Summary and Conclusions

I have presented a novel update discipline, named read-copy update, that can provide great reductions in synchronization overhead and can reduce the complexity of deadlock avoidance. Read-copy update generally gives the best performance improvement for read-mostly algorithms or under high contention. In some cases, the need for synchronization operations is completely eliminated.

I have presented a detailed view of an efficient implementation of read-copy update. This implementation, which uses an efficiently maintained summary of thread activity, fills an important gap in earlier work with concurrent update algorithms. Similar implementation have been used by commercial operating systems running in production on Sequent machines since 1993. The implementation described in this paper went into service in 1996.

I have presented a performance analysis of read-copy update, and have compared the results to those of other locking primitives. Read-copy update outperforms the other primitives in a wide variety of conditions and situations. These comparisons are quite conservative: even greater savings would be realized under conditions of high contention.

8 Acknowledgements

I owe thanks to Stuart Friedberg, Doug Miller, Jan-Simon Pendry, Chandrasekhar Pulamarasetti, Jay Vosburgh and Dave Wolfe for their willingness to try out read-copy update, and to Ken Dove, Brent Kingsbury, and to Phil Krueger and his Base-OS Reading Group for many helpful discussions. I am greatful to Dale Goebel for his support of this work. I owe special thanks to Jack Slingwine for the collaboration that lead to the concept and implementation of read-copy update.

This work was done with the aid of Macsyma, a large symbolic manipulation program developed at the MIT Laboratory for Computer Science and supported from 1975 to 1983 by the National Aeronautics and Space Administration under grant NSG 1323, by the Office of Naval Research under grant N00014-77-C-

0641, by the U. S. Department of Energy under grant ET-78-C-02-4687, and by the U. S. Air Force under grant F49620-79-C-020, between 1982 and 1992 by Symbolics, Inc. of Burlington Mass., and since 1992 by Macsyma, Inc. of Arlington, Mass. Macsyma is a registered trademark of Macsyma, Inc.

9 References

[Adams91] Gregory R. Adams. Concurrent Programming, Principles, and Practices, Benjamin Cummins, 1991.

[Barghouti91] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications, ACM Computing Surveys, September 1991.

[Burger96] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors, *ISCA'96*, (May 1996), pages 78-89.

[Hennessy91] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, page 18-28, Sept. 1991.

[Herlihy93] Maurice Herlihy. Implementing highly concurrent data objects, ACM Transactions on Programming Languages and Systems, vol. 15 #5, November, 1993, pages 745-770.

[Kung80] H. T. Kung and Q. Lehman. Concurrent manipulation of binary search trees, *ACM Trans. on Database Systems*, Vol. 5, No. 3 (Sept. 1980), 354-382.

[Lea97] Doug Lea. Concurrent Programming in Java, Addison-Wesley, 1997.

[Lovett96] T. Lovett and R. Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308-317, May 1996.

[Manber84] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure, *ACM Trans. on Database Systems*, Vol. 9, No. 3 (Sept 1984), 439-455.

[McKenney96] Paul E. McKenney. Selecting locking primitives for parallel programs, *Communications of the ACM*, Vol. 39, No. 10 (1996).

[McKenney98] Paul E. McKenney. Comparing performance of read-copy update and other locking primitives, Sequent TR-SQNT-98-PEM-1, January 1998.

[McKenney98a] Paul E. McKenney. Read-copy update: using execution history to implement low-overhead solutions to concurrency problems, Sequent TR-SQNT-98-2, February 1998.

[MellorCrummey91] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors, Proceedings of the Third PPOPP, Williamsburg, VA, April, 1991, pages 106-113.

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

[Scott92] Michael L. Scott and John M. Mellor-Crummey, Fast, contention-free combining tree barriers, University of Rochester Computer Science Department TR#CS.92.TR429, June 1992.

[Slingwine95] John D. Slingwine and Paul E. McKenney. System and Method for Achieving Reduced Overhead Mutual-Exclusion in a Computer System. *US Patent # 5,442,758*, August 1995.

[Stone91] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *IEEE Computer*, pages 30-38, Sept. 1991.