Read-Copy Update:
Using Execution History to Solve Concurrency Problems

P.E. McKenney, *Senior Member, IEEE,* and J.D. Slingwine

Abstract—The problems of synchronization overhead, contention, and deadlock can pose serious challenges to those designing and implementing parallel programs. Therefore, many researchers have proposed parallel update disciplines that greatly reduce these problems in restricted but commonly occurring situations, for example, read-mostly data structures [7, 10, 11, 13, 18]. However, these proposals rely either on garbage collectors [10, 11], termination of all processes currently using the data structure [13], or expensive explicit tracking of all processes accessing the data structure [7, 18]. These mechanisms are inappropriate in many cases, such as within many operating-system kernels and server applications. This paper proposes a novel and extremely efficient mechanism, called read-copy update, and compares its performance to that of conventional locking primitives under conditions of both low and high contention in read-intensive data structures.

Index Terms—Shared memory, mutual exclusion, reader-writer locking, performance, contention.

1 Introduction

Reducing lock contention and synchronization overhead will continue to be important in parallel design and implementation because increases in CPU-core instruction-execution rate are expected to continue to outstrip reductions in global latency for large-scale multiprocessors [3, 6, 22]. This trend will cause global lock and synchronization operations to continue becoming more costly relative to instructions that manipulate local data. Expensive global lock operations are particularly troublesome when the locks are used to guard read-mostly data structures. In this common special case, reading processes pay a heavy penalty to guard against very rare events. To see how rare these events can be, consider the following two examples.

The first example is a routing table for a system connected to the Internet. Many Internet routing protocols process routing changes at most every minute or so. Therefore, a system transmitting at the low rate of 100 packets per second would need to perform a routing-table update at most once per 6,000 packets, for an update fraction f of less than 10^{-3}.

The second example is a system with 100 mirrored disks, each of which has an MTBF

of 100,000 hours.[1] A transaction-processing system performing 10,000 disk I/Os per second would perform in excess of 10^{10} I/Os on the average before having to update the internal tables tracking which disk contains which data. This yields a value below 10^{-10} for *f*.

Both these examples demonstrate the performance benefits to be gained by decreasing the overhead incurred by reading threads and CPUs, even at the cost of great increases in that incurred by writing CPUs and threads. These examples motivate specialized locking designs that make fewer read-side references to globally shared, often-updated variables.

Deadlock avoidance will also continue to be increasingly important in parallel design and implementation. To see this, consider queuing models in which the locks act as servers and the CPUs or threads act as arrivals. As the number of CPUs or threads increases, one or more of the locks will saturate, so that adding CPUs or threads will fail to increase system performance. An effective remedy to this problem of limited system performance is to partition algorithms and data structures so that many CPUs or threads may be operating on different partitions in parallel. However, this remedy brings its own problem—more locks can render the system more prone to deadlock.

This paper describes a novel approach that addresses the lock-contention, synchronization-overhead, and deadlock problems for a specialized but commonly occurring set of situations.

2 Existing Solutions

The problems outlined in the previous section have been attacked by many researchers over a period spanning several decades. This section outlines representative solutions.

Reader-writer spinlocks [14] allow reading processes to proceed concurrently. However, updating processes may *not* run concurrently with each other or with reading processes. In addition, reader-writer spin-locks exact significant synchronization overhead from reading processes, although this overhead can be reduced in exchanged for greatly increased overhead exacted from writing processes [8]. On the other hand, reader-writer spin-locks allow writers to block readers and vice versa, thereby avoiding stale data. This tradeoff is shown in Figure 1—exclusion between readers and writers imposes lock-contention

[1] For purposes of comparison, disks with rated MTBFs in excess of 450,000 hours are readily available.

costs and increases the time required to become aware of an external event.

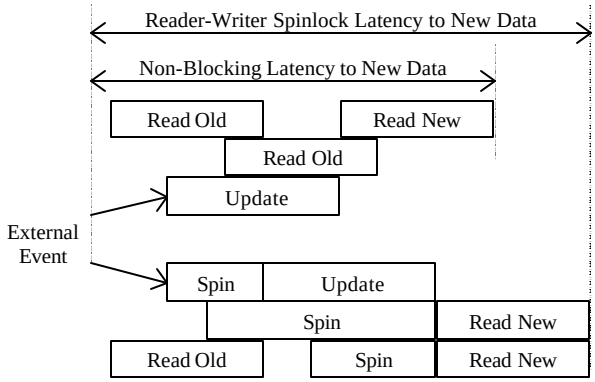

Figure 1: Latency Of Non-Blocking Update Compared To Reader-Writer Spin-Lock

Wait-free synchronization [7] allows reading and updating processes to run concurrently, but again exacts significant synchronization overhead. It also requires that memory used for a given type of data structure never be subsequently used for any other type of data structure, and that reading threads write to shared storage. On parallel computers, these writes will result in high-latency cache misses. On the other hand, wait-free synchronization provides wait-free processing to updates as well as to reads, and in addition avoids stale data.

Timestamping and versioning concurrency-control allow reading and updating processes to proceed concurrently, but impose synchronization overhead on reading processes [2]. Chaotic relaxation [1] accepts stale data to reduce locking overhead, but requires highly structured data.

Manber and Ladner [13] describe an algorithm that defers freeing a given node until all processes running at the time the node was removed have terminated. This allows reading processes to run concurrently with updating processes, but does not handle non-terminating processes such as those found in OSs and server applications. In addition, they do not describe an efficient mechanism for tracking blocks awaiting deferred free or for determining when the relevant processes have terminated.

Pugh [18] uses a technique similar to that of Manber and Ladner, but notes that (expensive) read-side state updates permit non-terminating processes. However, Pugh leaves to the reader the mechanism for efficiently tracking blocks awaiting deferred free.

Kung and Lea [10, 11] describe use of a garbage collector to manage the list of blocks awaiting deferred free, again, allowing reading processes to run concurrently with updating processes. 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. 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 in a timely manner after all reading processes have dropped references to the blocks awaiting deferred free.

Jacobson [9] describes perhaps the simplest possible deferred-free technique: simply waiting a fixed amount of time before freeing blocks awaiting deferred free. This works if there is a well-defined upper bound on the length of time that reading processes can hold references. However, if processes hold their references longer than expected (perhaps due to greater-than-expected load or data-structure size), memory corruption can ensue, with no reasonable means of diagnosis.

3 Suggested Solution

This paper describes a novel approach in which updating CPUs and threads refer to a *summary of thread activity* in order to determine when update operations may be safely carried out. In many cases, this summary may be derived from counters and statistics that must be maintained for other purposes. In these cases, the summary of thread activity imposes zero net overhead on reading processes and CPUs, which in turn provides dramatic speedups for access to read-intensive data structures.

The form, implementation, and use of a summary of thread activity are described in later sections. Summaries of thread activities may be constructed for both tightly coupled and distributed systems.

4 Conditions and Assumptions

Use of read-copy update is most likely to be helpful with read-intensive data structures, where a modest amount of memory may be spared for a list of deferred actions, where stale data may be either tolerated or suppressed, and where there are frequently occurring natural or artificial quiescent states.

By “read intensive”, we mean that the update fraction f is much smaller than the reciprocal of the number of CPUs. In some special cases, read-copy update can provide performance benefits even though f exceeds 0.9. As noted in the introduction, f can be as small as 10^{-10}.

Ever-increasing memory sizes tend to make the space needed for the list of deferred actions a non-problem, but the need for tolerance of stale data cannot always be so easily dismissed. However, any reading thread that *starts* its access *after* an update completes is guaranteed to see the new data. This guarantee is sufficient in many cases. In addition, data structures that track state of components external to the computer system

(e.g., network connectivity or positions and velocities of physical objects) must tolerate old data because of communication delays. In other cases, old data may be flagged so that the reading threads may detect it and take explicit steps to obtain up-to-date data, if required [13, 18].

Quiescent states and periods are the basis of read-copy update, and are discussed in detail in the next section.

5 Details of Solution

The following sections give definitions of terms, an outline of how read and update algorithms use read-copy update, examples of quiescent states, example uses of read-copy update, several possible designs for the summary of thread activity, an outline of an implementation of the summary of thread activity, and considerations for CPUs that cannot deterministically invalidate variables from other CPUs' caches.

5.1 Definitions

Temporary variable: A short-life-span data item used to cache intermediate results, such as results of searches of a particular data structure. Note that "variable" is used in a general sense that includes elements of arrays and fields of structures.

Permanent variable: A long-life-span data item used to hold persistent state, such as list headers. Note that the difference between a temporary and permanent variable is a design decision rather than a fundamental algorithmic property. Again, "variable" is used in a general sense.

Live variable: A variable that will be referenced before being overwritten.

Dead variable: A variable that either will never again be referenced or will be overwritten before being referenced.

Quiescent state: A state at which all temporary variables belonging to the CPU or thread passing through the quiescent state are dead. A quiescent state may be defined to be with respect to a particular data structure or with respect to all data structures in the system. Unless otherwise specified, the paper will use "quiescent state" in this latter universal sense. Examples of quiescent states in various software systems appear in a later section.

Quiescent period: A time interval during which each thread passes through at least one quiescent state. Note that any time interval that encloses a quiescent period is itself a quiescent period.

Summary of thread activity: A set of data structures that are used to identify quiescent periods.

5.2 Read Algorithm

Reading threads may access the read-copy updated data structure as if there was only one CPU with no possibility of preemption or interrupts of any sort (though there may be restrictions on how data is traversed). Reading threads therefore enjoy complete freedom from synchronization overhead.

5.3 Update Algorithm

Updating CPUs or threads perform a read-copy update as follows:

1. Perform any synchronization operations needed to guard against other updating CPUs or threads.
2. Perform the update so that any reading CPUs or threads are guaranteed to see consistent (though possibly stale) data. This includes any special operations required to enforce memory-access ordering.
3. Modify any permanent variables to reflect the update.
4. Wait for a quiescent period to elapse.
5. Perform any needed cleanup operations (for example, freeing up memory occupied by the old versions of the data).
6. Perform any synchronization operations needed to allow other updating CPUs or threads to proceed.

Any sort of update discipline may be used to accomplish the synchronization in steps 1 and 6, including explicit locking, atomic instructions, or techniques taken from wait-free synchronization [7]. However, if only one CPU or thread is allowed to update the data, *all* locking may be omitted.

In many cases, it is possible for step 6 to precede step 5 and even step 4. When step 6 uses explicit locking, this decreases lock contention.

Note that only step 4 is novel. Algorithms incorporating the other five steps have appeared previously [7, 10, 11, 13, 18].

5.4 Quiescent State Examples

Many applications and systems have natural *universal* quiescent states, which, as noted earlier, are quiescent states that apply to *all* data structures in the application or system.

For example, within an operating system (OS) with non-preemptive kernel threads, there is a direct mapping from "thread" to CPU. Any CPU that is in the idle loop, executing in user mode, offline (halted), or performing a context switch[2] cannot be

[2] Operating systems with preemptive kernels either must take explicit action to suppress context switches or must restrict use of the read algorithm to code in which context

holding any references to *any* kernel data structure. Therefore, each of these four states is a universal quiescent state.

Similarly, many parallel user applications drop all references to data structures while waiting for user input. Many transaction-processing systems drop all references to data structures at the completion of a transaction. Interrupt-driven real-time control systems often drop all references to data when running at base priority level. Reactive systems [23] often drop all references to application-level data structures upon completion of processing for a given event. Discrete-event simulation systems often drop all references to simulation data structures at the end of processing for each discrete event. These applications can therefore also possess natural universal quiescent states.

Such systems will normally maintain statistics that track the number of times that they pass through their natural quiescent states. For example, most OSs will maintain counts of context switches and most transaction-processing systems maintain counts of the number of transactions complete. These counts, kept for performance-monitoring purposes, can be used to greatly reduce the cost of tracking quiescent periods, as will be shown in later sections.

Note that a distributed system can be said to complete a quiescent period once each of its components has completed a quiescent period.

Systems without naturally occurring quiescent states can often accommodate the insertion of artificial quiescent states.

5.5 Relation to Other Locking Primitives

Locking primitives generally indicate ownership of a lock by possession of some resource. For example, any thread that has read-acquired a given lock, but not yet read-released it, can be said to read-hold that lock.

Read-copy update is analogous to reader-writer locks. The read-side algorithm in read-copy update can be thought of as owning a CPU or thread that is not currently passing through a quiescent state. This accounts for the low overhead of read-side read-copy update. The CPU or thread must be acquired in any case, so the read-side algorithm imposes no incremental cost or added risk of deadlock.

---

switches have already been disabled (e.g., interrupt handlers).

5.6 Usage Examples

The following two examples show how read-copy update may be used to modify a linked list and to flush a log buffer.

5.6.1 Lock-Free Linked-List Access

If a thread removes all references to a given data structure, it may safely free up the memory comprising that data structure after the end of the next quiescent period. Note that threads traversing the data structure need not acquire any locks. The required synchronization is achieved implicitly through the quiescent states—the quiescent period guarantees that no threads reference the data structure. Eliminating read-side locking can greatly increase speedups in the many cases where updates are rare. This same effect can be achieved using a garbage collector (in environments possessing them), but at greater cost. This greater cost stems from the need to modify otherwise-read-only data to indicate that a reference is held.

For a concrete example, consider a singly linked list with Thread 0 updating Element B while Thread 1 is doing a lock-free traversal.

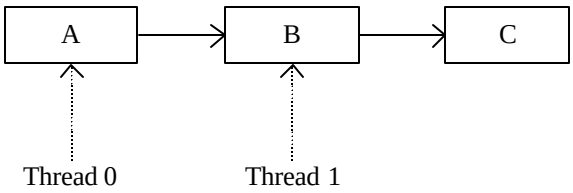

Figure 2: List Initial State

Suppose that Thread 0 needs to make a change to Element B that cannot be done atomically. Thread 0 cannot simply modify Element B in place, as this would interfere with Thread 1. Thread 0 instead copies Element B into a new Element B', modifies B', issues a memory-barrier operation, then points A's next pointer to B', as shown in Figure 3. This does not harm Thread 1 as long as B still points to C, and as long as Thread 0 waits until Thread 1 stops referencing B before freeing it.[3]

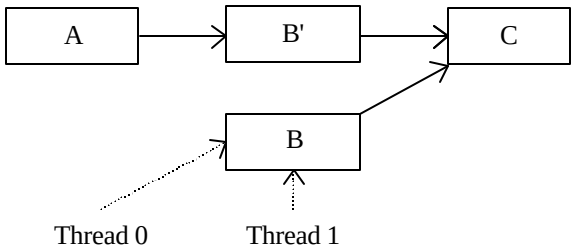

Figure 3: List Deferred Deletion

Thread 1 can no longer obtain a reference to B, so Thread 0 waits for a quiescent period (see Figure 1) before deleting it. After the quiescent period, all of Thread 1's temporary variables, including the pointer to B, are

[3] A pointer from B to B' may be used to allow Thread 1 to avoid stale data. Explicit locking may be used [15] to guarantee forward progress in cases where many updates are running concurrently with the reader.

dead. Thread 0 can then safely delete B, as shown in Figure 5.

This idiom of updating a copy of an element while allowing concurrent reads gives "read-copy update" its name. This idiom may be easily extended to handle arbitrarily linked multi-lists, although some care is required. For example a CPU or thread cannot expect to traverse back-pointers and find exactly the same elements it previously encountered.

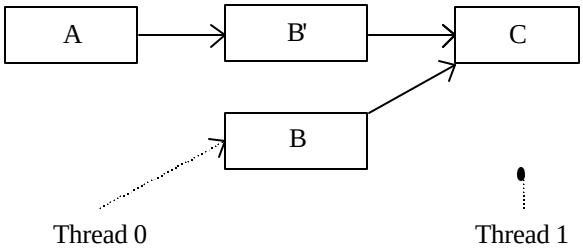

Figure 4: List After Quiescent Period

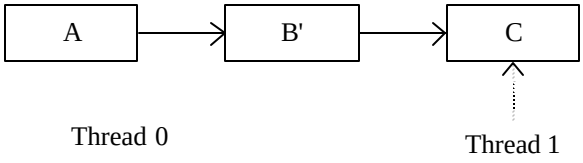

Figure 5: List After Deletion

5.6.2 Lock-Free Buffer Flushing

For another example, suppose that a parallel program creates log buffers that must be flushed to disk, but only after all log records have been completed. One approach is to maintain a global lock so that only one process at a time could create log records. However, this could result in a bottleneck under heavy load. Another approach is to use a global lock only to allocate space for the log records, and then create the actual records themselves in parallel. If the creation of a log record does not involve quiescent states, a flush may be safely initiated upon completion of a quiescent period starting after the last log record has been allocated.

Consider an initially empty two-entry log buffer:

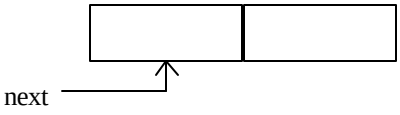

Figure 6: Log Buffer Initial State

If Threads 0 and 1 reserve both available slots, the situation will be as shown in Figure 7.

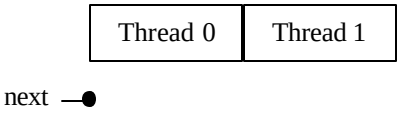

Figure 7: Log Buffer Fully Reserved

Both slots are occupied, and the "next" pointer is NULL. Therefore, Thread 2 must wait until Threads 0 and 1 have completed their entries, flush the log buffer, and only then reserve its slot.

Threads 0 and 1 could use explicit synchronization operations to inform Thread 2 when they have completed their log entries. However, this would result in needless synchronization operations for log buffers with large numbers of entries. Instead, Thread 2 waits to flush the buffer until the end of the next synchronization period. This

guarantees that Threads 0 and 1 have completed their entries (their temporary variables referencing the buffer are dead) without requiring synchronization operations.

Note that a garbage collector is not an appropriate solution for this second example, because we need to write out the log buffer instead of freeing it.[4]

5.7 SUMMARY OF THREAD ACTIVITY

Using a summary of thread activity to identify quiescent periods is beneficial only if it is coded very efficiently, otherwise, it is cheaper just to use locks. An efficient summary of thread activity is relatively complex, therefore, this section moves from simpler (but slower) implementations to more complex implementations suitable for large-scale shared-memory processing (SMP) and cache-coherent non-uniform memory-access (CC-NUMA) architectures.

For concreteness, we focus on a parallel non-preemptive OS kernel. In this case, the in-kernel threads map directly to CPUs, and the implementations focus on CPUs rather than threads.

The following sections describe the following implementations: (1) locking-primitive summary, (2) enforced quiescent states, (3) quiescent-state bitmask, and (4) quiescent-state counters.

5.7.1 Locking-Primitive Summary

Perhaps the most straightforward way of identifying quiescent states is to maintain count of the number of locks held by each CPU. When this number drops to zero on a given CPU, that CPU records the fact that it has entered a quiescent state by clearing a its bit in a global bitmask. When the value of the bitmask becomes zero, the end of a quiescent period has been reached. Any subsystem wishing to wait for a quiescent period sets each CPU's bit in the global bitmask.

Although this approach is simple, it is fatally flawed. First, it is slow, needing to update a global variable each time that a CPU releases its last lock.[5] Second, update disciplines not using locks would have their critical sections violated by this sort of summary of thread

[4] In some languages, it is possible to define finalization functions that are invoked at garbage-collection time. However, there is no guarantee that garbage collection will be performed in a timely manner.

[5] Updates to shared global variables are much more expensive than are updates to local per-CPU variables.

activity. Finally, a CPU that ran for an extended period without acquiring any locks (e.g., a CPU in the idle loop) would never clear its bit, despite being in an extended quiescent state.

Therefore, a different approach is required.

5.7.2 Enforced Quiescent States

Another simple approach is to *force* quiescent states, for example, via a daemon that handles quiescent-period requests. The daemon responds to a request by running on each CPU in turn, then announcing the end of the quiescent period, as shown in Figure 8.

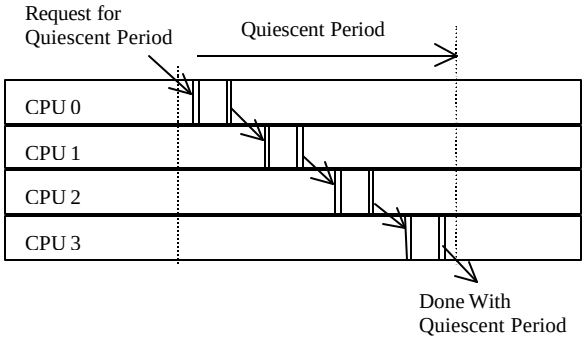

Figure 8: Enforced Quiescent States

Each CPU that the daemon runs on must do two context switches, one to switch to the daemon, and the other to switch away. A context switch is a quiescent state, so this set of context switches is a quiescent period, as desired. In this case, the summary of thread activity is maintained as part of the local state of the daemon itself.

This approach works well, and entered production on Sequent machines in 1993. Context switches are usually from one to three orders of magnitude more expensive than locking primitives, but for read-intensive data structures, the expense is justified. In addition, eliminating locks can greatly simplify deadlock avoidance. Furthermore, batching allows a single quiescent period to satisfy many requests.

Nevertheless, it is possible to do much better.

5.7.3 Quiescent-State Bitmask

Another approach is to instrument the quiescent states themselves. Each time a given CPU reaches a quiescent state, it clears its bit in a global bitmask. When the bitmask becomes zero, the quiescent state has ended. Any subsystem wishing to wait for a quiescent period sets each CPU's bit in the global bitmask.

A quiescent period measured in this manner is shown in Figure 9, with the rightmost bit in the four-bit mask corresponding to CPU 0. The individual memory transactions are shown because they dominate the performance characteristics of the summary of thread activity.

In Figure 9, CPU 1 requests a quiescent period. The bitmask initially resides only in CPU 2's cache, so CPU 1 must first obtain a

copy, as shown by the "Req" and "Rsp" arrows.

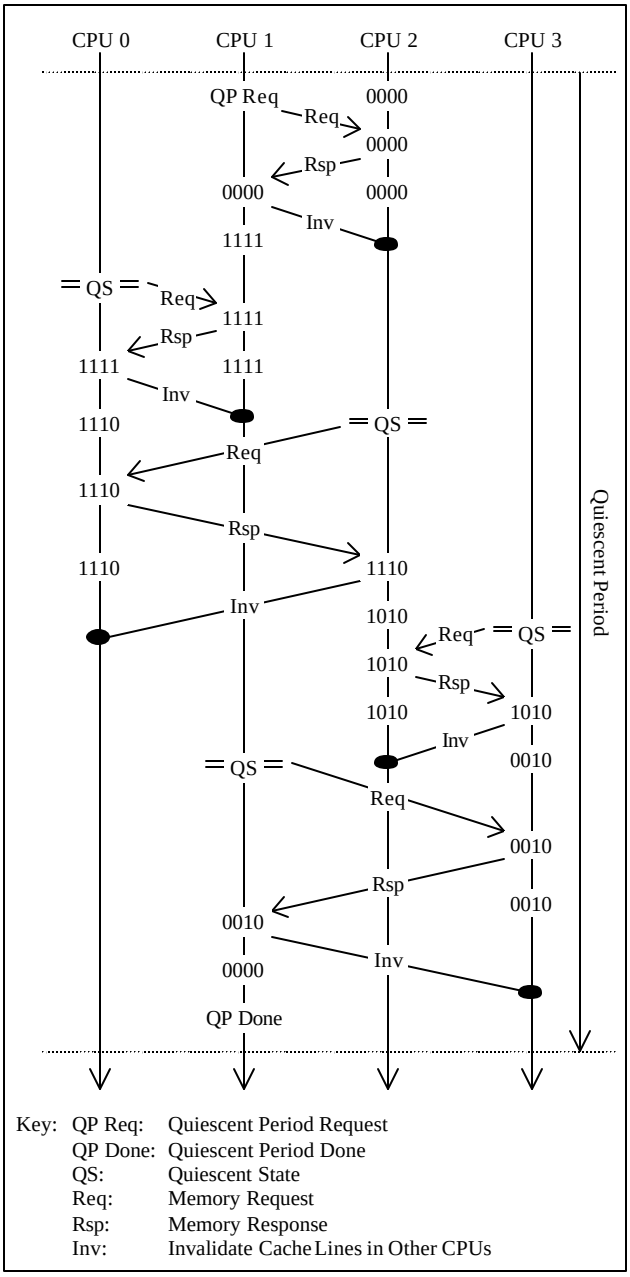

Figure 9: Quiescent-State Bitmask

CPU 1 then writes all one-bits to the bitmask, invalidating the copy in CPU 2's cache, as shown by the "Inv" line ending in a circle. CPU 0 is the first to pass through a quiescent state, so it gets a copy from CPU 1 in order to clear its bit, which invalidates the copy in CPU 1's cache. CPU 2 and CPU 3 pass through their quiescent states in a similar manner. Finally, when CPU 1 clears its bit, the bitmask becomes zero, indicating the end of the quiescent period.

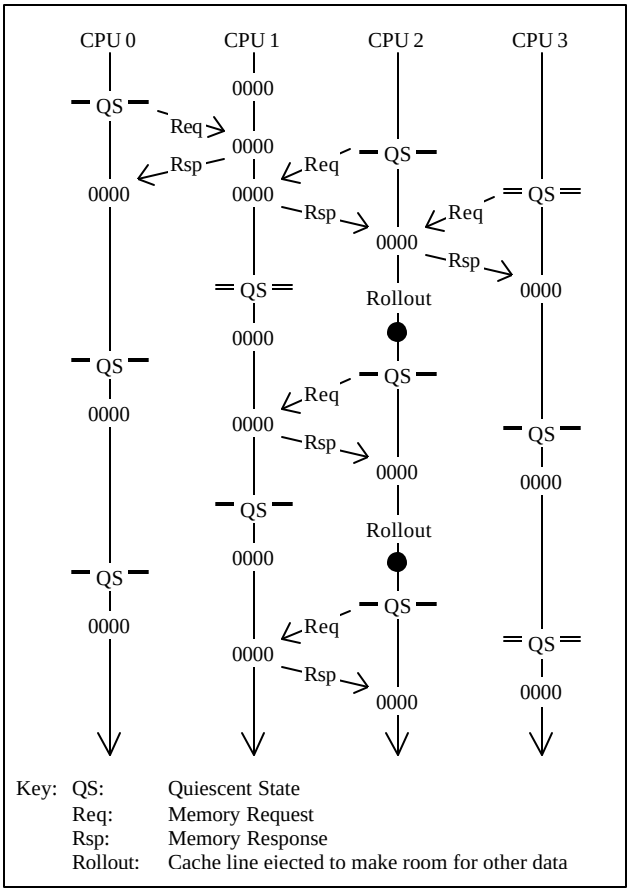

Figure 10: Bitmask Cache Thrashing

To prevent long-running user-level processes and idle CPUs from indefinitely extending a quiescent period, the scheduling-clock interrupt handler records a quiescent state any time that it interrupts either user-mode execution or the idle loop.

This approach can be faster than enforced quiescent states, but the frequent accesses to the shared global bitmask can be quite expensive, as shown in Figure 10.

CPU 2 is frequently rolling the bitmask out of its cache in order to make room for other data, thereby incurring expensive cache misses each time it passes through a quiescent state.

5.7.4 Quiescent-State Counters

More-efficient implementations isolate measurement from callback processing. Quiescent states are counted per-CPU and subsystems wait for quiescent periods by registering callbacks on per-CPU callback lists.

An OS kernel's quiescent states either are counted anyway or occur when the CPU is not doing anything useful. Examples of the former include system calls, traps, and context switches. Examples of the latter include the idle loop and removal of CPUs from service. The pre-existing counts of these events are used to implement a quiescent-period-detection algorithm that incurs almost no added cost.

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.
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 a quiescent period has ended, it executes any of its callbacks that were waiting for the end of that quiescent period.

Steps 2, 3, 4, and 6 all involve time delays that must be tuned to balance CPU consumption against the wall-clock time required to identify a quiescent period. This is a classic CPU-memory tradeoff: decreasing the quiescent-period-identification interval increases CPU consumption, while increasing

it increases the number of callbacks queued up waiting for a quiescent period.

An actual implementation faces these issues:

1. Proper handling of callbacks that are enqueued while a quiescent period is in progress. These callbacks must wait for a subsequent quiescent period to complete.
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.

5.8 Implementation

Our currently shipping implementation of read-copy update uses quiescent-state counters. An SMP version has been in production in Sequent Dynix/ptx since 1994. The CC-NUMA version went into production in 1996 on a hierarchical-bus architecture with four CPUs per local bus. Each local unit is called a *quad*.

The four issues listed in the previous section are handled as follows:

1. Each CPU maintains a separate queue of callbacks awaiting the end of a later quiescent period (nxtlist) as well as the queue of callbacks awaiting the end of the current quiescent period (curlist). Each quiescent period is identified by a *generation number*. Each CPU tracks the generation number corresponding to the callbacks in its curlist. 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. The implementation checks for new quiescent states from within an existing scheduling-interrupt handler, and uses software interrupts to dispatch callbacks whose quiescent period has ended. This incurs minimal overhead and 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. These variables are combined in a manner similar to Scott's and Mellor-Crummey's combining-tree barriers [19].
4. Callbacks are accumulated in nxtlist while the current quiescent period is in progress. The heavier the read-copy update load, the larger the batches and the smaller the per-callback overhead.

The following sections describe the quiescent-periods algorithm. More details are available [16, 21].

5.8.1 State Variables

The state variables for the quad-aware implementation of read-copy update are grouped into generation numbers, bitmasks, statistics, statistics snapshots, and callback lists.

Each quiescent period is identified by a generation number. Since the algorithm maintains loosely coupled state, there are several state variables tracking different generation numbers. The highest generation requested thus far is tracked by rcc_maxgen. The generation currently being serviced is tracked by rcc_curgen, which is replicated per-quad in pq_rcc_curgen. The earliest generation that a particular CPU needs to be completed is tracked by the per-CPU variable rclockgen.

The bitmasks track which CPUs and quads need to pass through a quiescent state in order for the current generation to complete. The set of quads that contain CPUs needing to pass through a quiescent state is tracked by rcc_needctxtmask, and the set of CPUs on a given quad needing to pass through a quiescent state is tracked by the per-quad variable pq_rcc_needctxtmask.

Each CPU tracks the number of context switches in the per-CPU variable cswtchctr. Each CPU tracks the number of system calls and traps from user mode in the per-CPU variables v_syscall and usertrap, respectively. Each CPU tracks the sum of the number of passes through the idle loop and the number of times a process to yielded that CPU in the per-CPU variable syncpoint.

As soon as a given CPU notes the start of a new generation, it snapshots its statistics: cswtchctr into rclockcswtchctr, v_syscall into rclocksyscall, usertrap into rclockusertrap, and syncpoint into rclocksyncpoint.

Read-copy callbacks advance through per-CPU callback lists nxtlist, curlist, and intrlist when quiescent periods are detected, as shown in Figure 11.

5.8.2 Pseudo-Code Overview

The pseudo-code call tree and function descriptions are as follows:

- hardclock()
  - rc_chk_callbacks()
    - rc_adv_callbacks()
      - rc_intr() (via software interrupt)
      - rc_reg_gen()
    - rc_cleanup()
      - rc_reg_gen()
      - rc_adv_callbacks()
        - rc_intr() (via software interrupt)
        - rc_reg_gen()

1. hardclock(): This scheduling interrupt is invoked by a per-CPU clock. It invokes rc_chk_callbacks() when this CPU's callbacks might advance. Advancing is possible when pq_rcc_needctxtmask indicates that this CPU needs to pass through a quiescent state, when pq_rcc_curgen indicates that the quiescent period for any callbacks in this CPU's curlist has ended, or when this CPU's curlist is empty and its nxtlist is nonempty.

2. rc_adv_callbacks(): Advances callbacks from this CPU's nxtlist to its curlist and from its curlist to its intrlist as quiescent periods complete. Also calls rc_intr() via software interrupt to invoke callbacks placed into its intrlist and calls rc_reg_gen() to register the presence of a new set of callbacks in its curlist.

3. rc_callback(): Registers a new read-copy callback by adding it to this CPU's nxtlist. Callbacks arriving during a given quiescent period are thus batched, greatly improving performance, as shown in Section 6.

4. rc_chk_callbacks(): Calls rc_adv_callbacks() in order to advance callbacks. Snapshots the statistics variables when it notes that a new quiescent period has started. Checks the current statistics against the snapshot in order to determine if this CPU has passed through a quiescent state, and, if so, calls rc_cleanup().

5. rc_cleanup(): At quiescent-period end, rc_cleanup() updates the generation numbers, and 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 the callbacks in intrlist, which 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.

5.8.3 Flow of Callbacks

New callbacks are injected into the system by rc_callback(). While the callbacks are awaiting invocation by rc_intr(), they are kept on per-CPU linked lists, and flow through the system as shown in Figure 11. A rc_onoff() function (not shown) moves callbacks to a global list when a CPU is taken out of service.

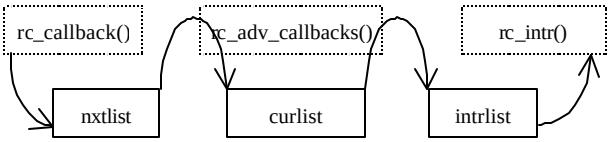

Figure 11: Flow Of Callbacks

The actual implementation also includes functions to check for CPUs taking too long to reach a quiescent state. This pinpoints areas that are impacting real-time response.

5.9 Memory-Consistency Models

Memory-consistency models [5] can have a significant effect on read-copy update. Although it is not difficult to construct read-copy update algorithms that are compatible with all memory-consistency models, simpler algorithms can be constructed for the stronger models.[6]

The different algorithms for the different memory-consistency models are illustrated by a linked-list insertion example.

5.9.1 Sequential Consistency

To insert an element into a linked list, given a sequentially consistent memory model:

1. Perform any synchronization operation needed to prevent others from concurrently modifying the list.
2. Allocate and initialize the new element, including the pointer to its successor-to-be.
3. Update the new element's predecessor's pointer.
4. Perform any synchronization operation needed to allow others to modify the array.

Reading processes simply traverse the linked list, as noted earlier. Of course, if only one CPU is permitted to add to the linked list, then the synchronization operations in steps 1 and 4 may be omitted.

This implementation also works on systems where writes from a given CPU are seen by other CPUs in the order written.

5.9.2 Release Consistency

If writes may be reordered, but a special memory barrier operation is available that segregates writes issued by the CPU executing the memory barrier, then it is necessary to execute the memory barrier operation before updating the pointers. By "segregates writes", we mean that all writes preceding the memory barrier are visible to

[6] Compilers can introduce code-reordering complications, which can be dealt with via C/C++ "volatile" declarations or equivalents.

all CPUs before any writes following the memory barrier.

To insert into a linked list under release consistency:

1. Perform any synchronization operation needed to prevent others from concurrently modifying the list.
2. Allocate and initialize the new element, including the pointer to its successor-to-be.
3. Execute a memory barrier, in order to make the contents of the new element visible to all CPUs.
4. Update the new element's predecessor's pointer.
5. Perform any synchronization operation needed to allow others to modify the array.

Again, reading processes simply traverse the linked list, as noted earlier. And again, if only one CPU is permitted to add to the linked list, then the synchronization operations in steps 1 and 4 may be omitted.

5.9.3 Weaker Memory-Consistency Models

Some CPUs, such as the Alpha AXP [20] lack a memory barrier instruction that fully segregates writes. Instead, weaker memory-barrier instructions are executed by both the reading and the updating CPUs. In this model, CPUs inserting into a linked list could still use the procedure shown in Section 5.9.2. However, CPUs traversing the list would execute a memory-barrier instruction after fetching each pointer, but before dereferencing it.

Although this scheme executes correctly, the internal synchronization and state-flushing operations performed by memory-barrier instructions are orders of magnitude more expensive than normal instructions. Furthermore, weak-memory-consistency CPUs other than Alpha require other arrangements of special instructions. Fortunately, there are a number of ways to safely eliminate read-side use of these expensive operations in read-copy update algorithms.

5.9.3.1 Freelist of Invalid Values

One method for CPUs similar to the Alpha AXP assumes that there is at least one field in each cacheline of each element with a distinct *invalid* value [4]. A reading process would use the following procedure to traverse each element of the list:

1. Dereference the pointer to the element.
2. If any of the fields in the element contain invalid values, execute a memory-barrier instruction.

3. Perform any needed operation on the current element.
4. Repeat from step 1 with the pointer-to-next field.

An updating process uses the procedure outlined in Section 5.9.2, except that Step 2 allocates the element from a special freelist whose elements have all their fields set to invalid values. Elements are freed onto this special freelist using the following procedure:

1. Set all fields to the corresponding invalid value.
2. Wait for a quiescent period to elapse.
3. Add the element to the freelist.

The algorithm for detecting quiescent periods guarantees that each CPU will acquire and release at least one lock. The lock primitives contain a memory-barrier instruction, which ensures that each CPU will see the invalid values when the element is removed from the freelist.

This method works well on the Alpha AXP, and requires only a small amount of read-side overhead. However, it is possible to eliminate even this modest amount of read-side overhead, as shown in the next section.

5.9.3.2 Quiescent Periods

The general method is to wait for a quiescent period after filling in the structure, but before installing pointers to the structure. Since each CPU must execute at least one memory-barrier instruction during a quiescent period as a side effect of maintaining the summary of thread activity, this procedure suffices to ensure memory consistency. The update procedure for a singly-linked list is as follows:

1. Perform any synchronization operation needed to prevent others from concurrently modifying the list.
2. Allocate and initialize the new element, including the pointer to its successor-to-be.
3. Wait for a quiescent period in order to make the contents of the new element visible to all CPUs.
4. Update the new element's predecessor's pointer.
5. Perform any synchronization operation needed to allow others to modify the array.

This update procedure allows reading processes to simply traverse the list, without any need for synchronization operations. However, this approach potentially requires the updating process to hold a lock for the full duration of a quiescent period, which in some cases results in an unacceptably low update rate.

There are a number of well-known methods for handling this update-rate bottleneck. One is to hash the list, with each bucket having its own lock, so that updates to multiple buckets may be performed in parallel. A second is to batch the updates, so that multiple elements are added to the list over a single quiescent period.

6 Analytical Comparison

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.
2. per-generation costs. These are incurred during each read-copy generation.
3. per-batch costs. These are incurred during each read-copy batch. Per-batch costs are incurred only by CPUs that have a batch during a given generation. These costs are amortized over callbacks making up that batch.
4. per-callback costs. These are incurred for every read-copy callback.

Details of the derivations may be found in companion technical reports [15, 16]. The symbols are defined as follows: f is the fraction of lock acquisitions that do updates; m is the number of CPUs per quad; n is the number of quads, t_c is the time required to access the fine-grained hardware clock; t_f is the latency of a fast access that hits the CPU's cache; t_m is the latency of a medium-speed access that hits memory or cache shared among a subset of the CPUs; t_s is the latency of a slow access that misses all caches, and r is the ratio of t_s to t_f.

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:

$$C_h = nmt_c + 3nmt_f \qquad \textbf{Equation 1}$$

$$C_g = \left(3n + 2nm - m\right)t_s + \left(2nm + m - 1\right)t_m + 2nmt_f \qquad \textbf{Equation 2}$$

$$C_b = 3t_s \qquad \textbf{Equation 3}$$

$$C_c = 7t_f \qquad \textbf{Equation 4}$$

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, or $7t_f$. Note that this time period is shorter than may be measured accurately on modern speculative and multi-issue CPUs.

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 5:

$$C_{wc} = \begin{pmatrix} 3n + 2nm - m + 3 \\ 12nm + m - 16 \\ 2nm^2 + 7nm + 8 \\ nm^2 \end{pmatrix} \begin{pmatrix} t_s + \\ t_n + \\ t_f + \\ t_c \end{pmatrix}$$

Equation 5

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. In typical commercial workloads, CPUs will pass through several quiescent states per period.

Typical costs may be computed assuming a system-wide Poisson-distributed inter-arrival rate of l per generation, as shown in Equation 6.

$$C_{typ} = \frac{\sum_{k=1}^{\infty} \frac{l^k e^{-l}}{k!} C_k}{1 - e^{-l}}$$

Equation 6

Here $(l^k e^{-l})/k!$ is the Poisson-distributed probability that k callbacks are registered during a given generation if on average l of them arrive per generation. Note that the 0th term of the Poisson distribution is omitted, since there is no read-copy overhead if there are no read-copy arrivals. The division by $1-e^{-l}$ normalizes for this omission. The quantity C_k is defined as follows:

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

Equation 7

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 solution to the occupancy problem:

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

Equation 8

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

Substituting Equation 7 and Equation 8 into Equation 6 and substituting Equation 1, Equation 2, Equation 3, and Equation 4 into the result, then multiplying by the update fraction f yields the desired expression for the typical cost:

$$\frac{f}{e^l - 1} \sum_{k=1}^{\infty} \frac{l^k \begin{pmatrix} 3n + 5nm - m \\ 3nm\left(1 - \frac{1}{nm}\right)^k \\ 10nm + 7k + 1 \end{pmatrix} \begin{pmatrix} r - \\ r + \\ nmt_c \end{pmatrix}}{k!k}$$

Equation 9

These results are displayed in the following figures. The traces are labeled as follows: "drw" is per-CPU distributed reader-writer spin-lock; "qrw" is per-quad distributed reader-writer spin-lock; "sl" is simple spin-

lock; "rcb" is best-case read-copy update; "rcp", "rcz", and "rcn" are read-copy update with Poisson-distributed arrivals with l equal to 10, 1, and 0.1, respectively; and "rcw" is worst-case read-copy update.

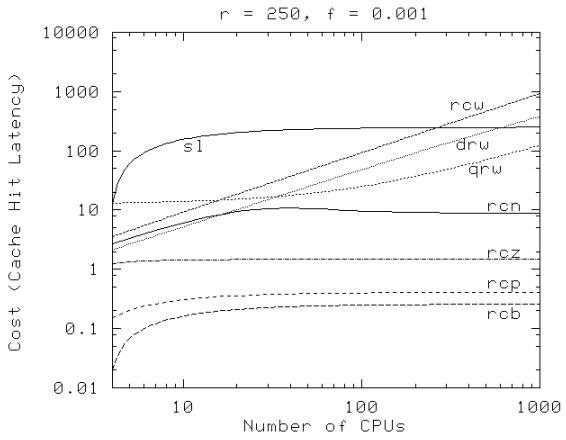

Figure 12: Overhead vs. Number of CPUs

Figure 12 displays read-copy update overhead as a function of the number of CPUs. 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, due to the batching capability of read-copy update. Although simple spin-lock also shows good scaling, this good behavior is restricted to low contention.

Figure 13 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.

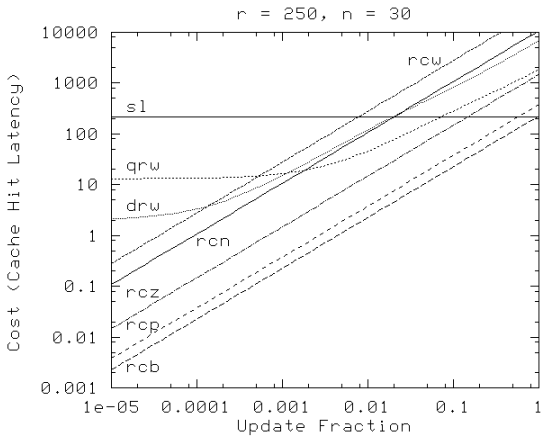

Figure 13: Overhead vs. Update Fraction

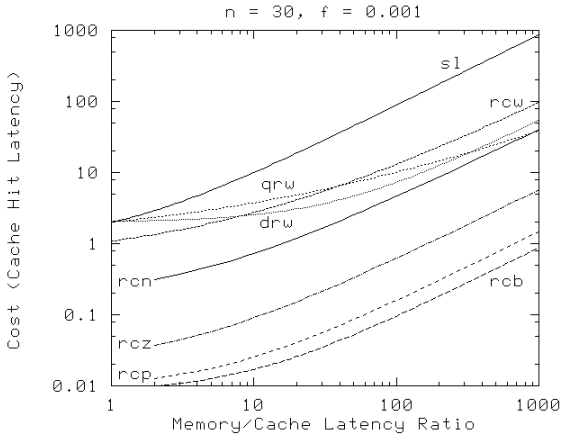

Figure 14: Overhead vs. Latency Ratio

Figure 14 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 high contention, larger numbers of CPUs, or by lower update fractions, as shown in Figure 15**.**

The situation shown in Figure 15 is far from extreme. As noted earlier, common situations can result in update fractions below 10^{-10}.

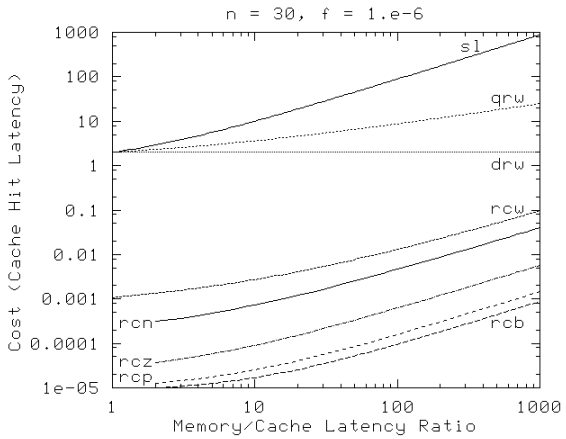

Figure 15: Overhead vs. Latency Ratio for Low *f*

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

7 Measured Comparison

Data shown in this section was collected on a Sequent NUMA-Q machine with 32 Xeon CPUs, each running at 450 MHz.

The results for simple spin-lock shown in Figure 16 show good agreement between the analytic model and measurements taken on real hardware. Note that there are small but measurable deviations both for smaller and for larger numbers of quads. The deviations at the low end are due in part to hardware optimizations and speculative execution, neither of which are accounted for in the model. The deviations at the high end are due in part to longer sharing list It is possible to design more complex locking primitives that do not suffer from these deviations, but such designs are beyond the scope of this paper.

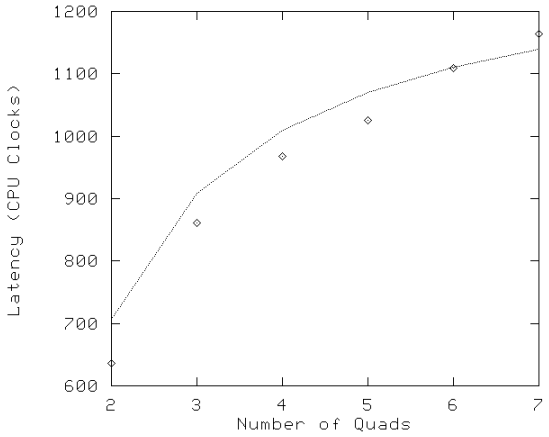

Figure 16: Measured vs. Analytic Latencies for Simple Spin-Lock

The results for distributed reader-writer spin-lock shown in Figure 17 also show good agreement between the analytic model and real hardware. Deviations are less apparent than for simple spin-lock because of the log-scale latency axis used to accommodate the wide range of latencies measured for reader-writer spin-lock.

Measured and analytic results for read-copy update also show good agreement, but only for older Pentium CPUs that do not feature speculative and out-of-order execution. The very short and low-overhead code segments implementing read-copy update make it impossible to accurately measure read-copy

update overhead at low levels of contention on modern speculative multi-issue CPUs.

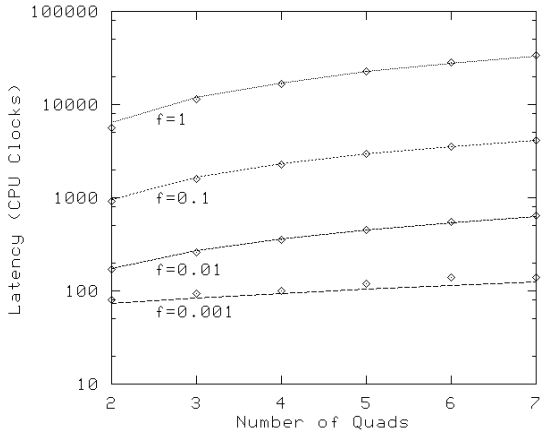

Figure 17: Measured vs. Analytic Latencies for Reader-Writer Spin-Lock

Instead, we measured the locking overhead at high levels of contention for simple spin-lock, reader-writer spin-lock, and read-copy update. This approach allows accurate bulk-measurement techniques to be applied. However, since the analytic results assume low contention, these measurements can only be compared to each other, not to the analytic results.

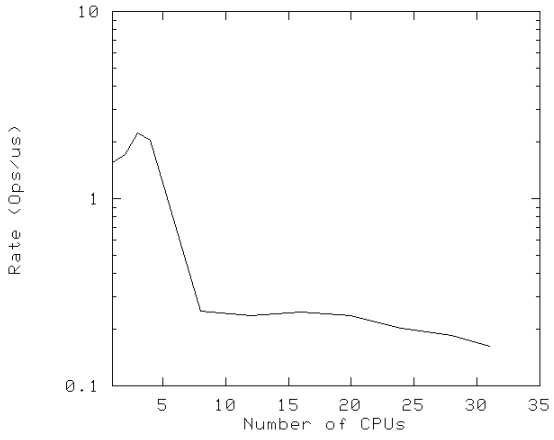

Figure 18: Simple Spin-Lock at High Contention

Figure 18 shows the expected results for simple spin-lock at high contention. The sharp drop in system-wide critical sections per microsecond between four and eight CPUs is due to the greater latency of remote-memory accesses. It is possible to create spin-locks that behave much better under high contention, but these are beyond the scope of this paper.

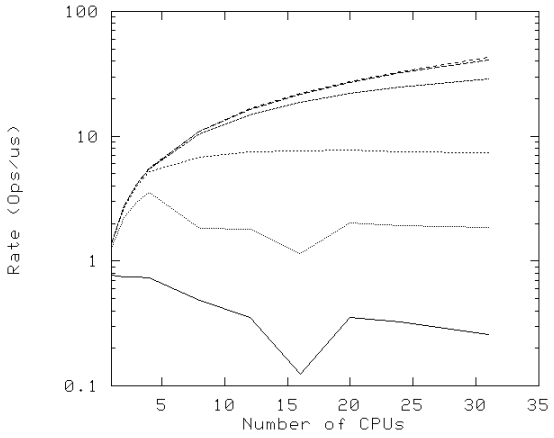

Figure 19: Reader-Writer Spin-Lock at High Contention

Figure 19 shows that distributed reader-writer spin-locks produce much greater levels of critical-section throughput under high contention, but only if read-side acquisitions dominate. The uppermost trace in this figure corresponds to $f=10^{-6}$, and each lower trace corresponds to an order-of-magnitude increase in f, up to $f=0.1$ in the lowermost trace. Note that the traces for 10^{-6} and 10^{-5} are almost overlapping. The erratic nature of the $f=0.1$ and $f=0.01$ traces is due to extreme write-side contention, which results in

interactions between the locking algorithm and the cache-coherence protocol.

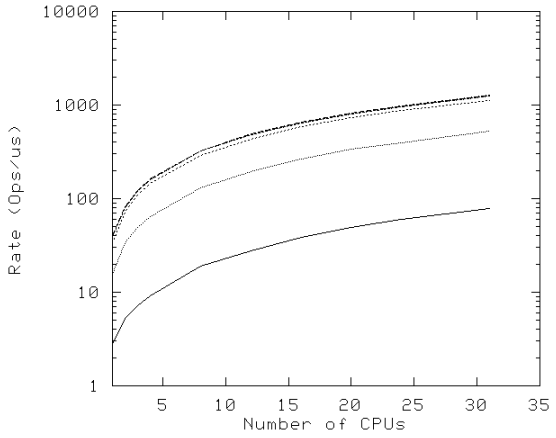

Figure 20: Read-Copy Update at High Contention

Figure 20 shows that read-copy update enjoys linear scaling with increasing numbers of CPUs even under high contention, independent of the value of *f*. Again, the lowermost trace corresponds to *f=0.1*, with each higher trace corresponding to an order-of-magnitude decrease in *f*. The traces for *f=0.001* through $f=10^{-6}$ are nearly overlapping in the figure. At moderately low values of *f*, read-copy update achieves more than one billion critical sections per second, more than an order of magnitude greater than reader-writer spin-lock and many orders of magnitude greater than simple spin-lock.

However, it must be noted that the high-contention case results in high values of λ, which allows the overhead of quiescent-period detection to be amortized over many requests. Nevertheless, there are applications of read-copy update that exhibit this high level of λ.

8 Conclusion

We have presented a novel update discipline, named read-copy update, which provides great reductions in synchronization overhead, tolerates non-terminating threads and reduces deadlock-avoidance complexity. Read-copy update generally gives the best performance improvement for read-mostly algorithms or under high levels of contention. In some cases, the need for synchronization operations is completely eliminated.

We have delineated read-copy update's area of applicability: Data structures that are often accessed and seldom updated, where a modest amount of memory may be spared for structures waiting on a quiescent period, and where stale data may be tolerated or can be suppressed.

We have provided a firm theoretical basis for read-copy update, along with a very efficient implementation. This implementation, which uses a summary of thread activity, fills an important gap in earlier work with concurrent update algorithms. The implementation comprises about 1,500 lines of C code, and has run in production on Sequent machines since 1994.

We have presented measurements that demonstrate multiple orders-of-magnitude reductions in overhead compared to simple spin-lock at low contention.

Future work includes comparison with other methods of maintaining concurrent data structures and analysis of behavior under high contention.

9 Acknowledgements

We 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. We are indebted to Kevin Closson for use of machines used to collect measured data. We are greatful to Dale Goebel, Dave Stewart, John Cherry, and Hans Tannenberger for their support of this work.

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.

10 References

[1] G. R. Adams. Concurrent Programming, Principles, and Practices, Benjamin Cummins, 1991.

[2] N. S. Barghouti and G. E. Kaiser. “Concurrency control in advanced database applications”, ACM Computing Surveys, 23(3), pp. 269-318, September, 1991.

[3] D. Burger, J. R. Goodman, and A. Kägi. “Memory bandwidth limitations of future microprocessors”, Proceedings of the 23rd International Symposium on Computer Architecture, pp. 78-89, May, 1996.

[4] W. Cardoza, private communication.

[5] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors, technical report CSL-TR-95-685, Stanford University, December, 1995.

[6] J. L. Hennessy and Norman P. Jouppi. Computer technology and architecture:

An evolving interaction. IEEE Computer, 24(9), pp. 18-28, September, 1991.

[7] M. Herlihy. Implementing highly concurrent data objects, ACM Transactions on Programming Languages and Systems, 15(5), pp. 745-770, November, 1993.

[8] W. C. Hsieh & W. E. Weihl, "Scalable Reader-Writer Locks for Parallel Systems", Tech report MIT/LCS/TR-521, November, 1991

[9] V. Jacobson. “Avoid read-side locking via delayed free”, private communication, September, 1993.

[10] H. T. Kung and Q. Lehman. “Concurrent manipulation of binary search trees”, ACM Trans. on Database Systems, 5(3), pp. 354-382, September, 1980.

[11] D. Lea. Concurrent Programming in Java, Addison-Wesley, 1997.

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

[13] U. Manber and R. E. Ladner. “Concurrency control in a dynamic search structure”, ACM Trans. on Database Systems, 9(3), pp. 439-455, September, 1984.

[14] P. E. McKenney. “Selecting locking primitives for parallel programs”, Communications of the ACM, 39(10), October, 1996.

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

[16] P. E. McKenney. “Implementation and performance of read-copy update”, Sequent TR-SQNT-98-PEM-4.0, March 1998.

[17] J. M. Mellor-Crummey and M. L. Scott. “Scalable reader-writer synchronization for shared-memory multiprocessors”, Proceedings of the Third PPOPP, Williamsburg, VA, pp. 106-113, April, 1991.

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

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

[20] R. L. Sites and R. T. Witek. Alpha AXP Architecture Reference Manual, Digital Press, 1995.

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

[22] H. S. Stone and J. Cocke. "Computer architecture in the 1990s". IEEE Computer, 24(9), pages 30-38, Sept. 1991.

[23] J. M. Vlissides, J. O. Coplien, and N. L. Kerth. Pattern Languages of Program Design, Volume 2, Addison-Wesley, 1995.