Using Thread History to Implement Low-Overhead Solutions to Concurrency Problems

Paul E. McKenney and John D. Slingwine
Sequent Computer Systems, Inc.

TR-SQNT-97-PEM-2.1

1.Abstract

Sooner or later, every parallel programmer experiences the frustration of a particularly elegant-seeming design suffering from deadlock, livelock, or unacceptable synchronization overhead. Resolving these problems can greatly increase the design's complexity, thereby imposing unacceptable development and maintenance costs.

However, a novel synchronization technique allows the programmer to circumvent these issues in restricted yet commonly occurring circumstances. This technique, named *read-copy update*, uses a thread-activity summary to determine when potentially dangerous operations may be safely carried out. For some common algorithms, these techniques can provide near-zero synchronization overhead.

2.Introduction

Read-copy update addresses a restricted but commonly occurring class of applications in which data is accessed much more frequently than it is updated, where use of stale data may be tolerated, and where memory size is not the limiting factor. The prototypical application in this class is the network routing table. Updates are infrequent, and, in the event of a change, stale routing data has been in use for quite some time before the update is even received. Therefore, it is acceptable to use stale routing data for a short time after the update is received. In return, read-copy update allows reading processes to run concurrently with updating processes, provides wait-free processing to reading processes, and provides extremely low synchronization overheads.

There are several categories of 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 and 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 both readers and updaters 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 that is similar to read-copy update. Their algorithm defers freeing nodes until all processes running at the time that 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 the non-terminating processes such as those found in operating systems and server applications. In addition, they do not describe an efficient mechanism for maintaining this list.

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, however, such avoidance requires use of explicit and costly synchronization operations.

3.Read-Copy Update

The following linked-list example will be used to illustrate read-copy update:

```
/*
 * Search for the element with the specified key.  Return a pointer
 * to it, or NULL if no such key is found.  Set the location pointed to by pred
 * to point to the element preceding the one found, or to the last element if
 * there is no such element.  In either case, leave the list locked so that the
 * caller may safely manipulate the element found.
 */
el_t *search(el_t *hdr, key_t key, el_t **pred)
{
        el_t *p;

        P(mutex);
        *pred = hdr;
        p = hdr->next;
        while (p != NULL) {
                if (match(p, key)) {
                        return (p);
                }
                *pred = p;
                p = p->next;
        }
        return (NULL);
}

/*
 * Insert a new element at the end of the list, or return FALSE if an element with
 * the specified key is already a member of the list.
 */
bool_t insert(el_t *hdr, el_t *new)
{
        el_t *p;
        el_t *pred;

        p = search(hdr, new->key, &pred);
        if (p != NULL) {
                V(mutex);
                return (FALSE);
        }
        new->next = pred-> next;        /* A */
        pred->next = new;               /* B */
        V(mutex);
        return (TRUE);
}

/*
 * Delete the specified element from the list, or return FALSE if no such element
 * is a member of the list.
 */
bool_t delete(el_t *hdr, key_t key)
{
        el_t *p;
        el_t *pred;

        p = search(hdr, key, &pred);
        if (p == NULL) {
                V(mutex);
                return (FALSE);
        }
        pred->next = p->next;
        V(mutex);
        free(p);
        return (TRUE);
}
```

```
/*
 * Update the element with the specified key, or return FALSE if no such element
 * is a member of the list.
 */
bool_t update(el_t *hdr, key_t *key)
{
        el_t *p;
        el_t *pred;

        p = search(hdr, key, &pred);
        if (p == NULL) {
                V(mutex);
                return (FALSE);
        }
        p->count = p->count + 1;
        V(mutex);
        return (TRUE);
}
```

These C functions use simple spinlocks, denoted by the P and V operations, to protect a singly-linked list. If elements are inserted, but never deleted, then the locking operations in `search()` may be eliminated, due to the careful ordering of operations in `insert()`.[1] However, this optimization is not safe if `delete()` is used, because a CPU in `search()` might suddenly find that the element pointed to by `p` had been freed. As Manber and Ladner noted, if we could efficiently determine when there are no more processes holding references to the newly deleted element, we could remove this danger by deferring the free until that time.

Waiting until all currently active processes terminate is not feasible in a program with non-terminating processes. However, such programs frequently pass through *quiescent states* where they do not reference any protected data structures. Example quiescent states include the idle loop in an operating system, user state in an operating system, a context-switch point in an operating system, a wait for user input in an interactive application, a wait for new messages in a message-passing software system, a wait for new transactions in a transaction-processing system, a wait for control input in an event-driven real-time control system, the base priority level in a completely interrupt-driven real-time system, and the event-queue processor for a discrete-event simulation system. In such programs, we can simply wait until all processes have passed through at least one quiescent state before freeing deleted data structures.

If we prohibit blocked processes from holding references to protected data, as is normally the case with spinlock-protected data, then we need only track the activities of the individual CPUs rather than of all processes.[2] In a kernel, this restriction allows the normally collected per-CPU counts of context switches, system calls, and traps to be used to determine when the CPUs have passed through a quiescent state. The overhead of checking for quiescent states may be greatly reduced by batching deferred frees into *generations* and by maintaining a periodically updated *summary of thread activity*.

A `deferred_free()` primitive allows the lock acquisition to be moved from the `search()` function to the `insert()`, `delete()`, and `update()` functions. This means that the performance of the `search()` function now scales with CPU core clock frequency rather than with the global system interconnect, greatly improving the performance of programs that are dominated by searches. Note that systems with weak memory-consistency models may require a memory-commit primitive between the lines marked A and B in `insert()`. Weak memory-consistency models are becoming increasingly common due to their performance benefits.

The name "read-copy update" is derived from the a usage idiom where the structure to be updated is copied, and then the copy is updated and inserted in the place of the original, all concurrent with reading processes.

[1] The `insert()` function would then need to begin with a `P(mutex)`.

[2] Enforcing this restriction in user-level code requires primitives that disable preemption. In a pre-emptable kernel, disabling interrupts normally suffices.

3.1Read-Copy Update Data Structures

The read-copy-update primitives, such as deferred_free(), use the following per-CPU variables:

Variable	Definition
cswtchctr*	Counter that is incremented on each context switch that occurs on the corresponding CPU.
rclockcswtchctr*	Copy of the cswtchctr variable taken after the beginning of each generation.
rclockgen*	The generation number that blocks in rclockcurlist are waiting on.
rclocknxtlist	List of memory blocks that are to be freed at the end of the next generation.
rclockcurlist	List of memory blocks that are to be freed at the end of the current generation.
rclockintrlist	List of memory blocks that are ready to be freed. These blocks are freed by a software interrupt routine that runs at a lower priority level.
me	Index of the current CPU, starting at zero.

Variables corresponding to other quiescent states (system call, trap, idle loop, user-mode execution) have been omitted in interest of simplicity. The overhead of checking them *is* included in the performance comparisons. In addition, the semantics shown are restricted to deferred freeing of memory. The actual implementation allows arbitrary functions to be executed, which provides the surprisingly powerful semantic "execute this only after all CPUs are done with whatever they are currently doing".

The primitives use the following global variables:

Variable	Definition
rcc_mutex	Simple spinlock that serializes access to the other global variables.
rcc_curgen*	The number of the current batch, or generation, of deferred frees.
rcc_maxgen*	The number of the largest requested generation. This will be one less than rcc_curgen when there are no outstanding requests.
rcc_olmsk	Bitmask whose bits indicate which CPUs are currently operational.
rcc_needctxtmask*	Bitmask whose bits indicate which CPUs have not yet passed through a quiescent state during the current generation. This would be implemented as a hierarchical bitmap on hierarchical-bus systems.

Additional variables used to gather statistics, perform consistency checks, and to handle CPUs being taken out of service have been omitted in interest of simplicity. The overhead of checking and updating these variables *are* included in the performance comparisons. The full implementation is described in [Slingwine95].

The asterisked variables comprise the thread activity summary.

3.2Read-Copy Update Primitives

The primitives that handle update-side processing for read-copy update are:

1. `deferred_free()`. This primitive registers a block of memory for freeing after all CPUs have passed through at least one quiescent state.
2. `rc_chk_callbacks()`. This primitive is invoked from the clock interrupt handler to advance the memory blocks through rclocknxtlist, rclockcurlist, and rclockintrlist as the read-copy generations advance, but only if at least one of the following conditions hold:
    a) rclocknxtlist is nonempty and rclockcurlist is empty, in other words, if the recently registered memory blocks may now be moved to join a generation.
    b) rclockcurlist is nonempty and the corresponding generation has completed, in other words, if the memory blocks are ready to be freed.
    c) The current CPU's bit in rcc_needctxtmask is set, in other words, if the current CPU must pass through a quiescent state before the current generation can complete.
3. `rc_intr()`. This primitive is invoked at low priority to actually free the memory blocks that have advanced all the way through rclocknxtlist, rclockcurlist, and rclockintrlist.

4. `rc_reg_gen()`. This primitive is invoked to announce a requested generation. The first invocation of this primitive for a given generation marks a boundary between batches of memory blocks.

These primitives work with lists of memory blocks type-named `dfree_t`.

The implementation of `deferred_free()` is as follows:

```
void deferred_free(void *p)
{
        dfree_t *dfp = (dfree_t *)p;

        DISABLE();
        *rclocknxtlist.tail = dfp;
        rclocknxtlist.tail = &(dfp->next);
        ENABLE();
}
```

The `ENABLE()` and `DISABLE()` primitives enable and disable interrupts, respectively. This primitive simply places the memory to be deferred-freed onto a per-CPU list. The per-CPU nature of this list ensures that cache lines containing the list and its elements remain local to the CPU, in turn ensuring high performance.

The implementation of `rc_chk_callbacks()`, which is similar to a lazy barrier computation [Scott92], is as follows:

```
void rc_chk_callbacks()
{
        if (!isempty(rclockcurlist) &&
            (rclockgen < rcc_curgen)) {

                /* Move ready-to-free blocks to intr list. */

                *rclockintrlist.tail = rclockcurlist.head;
                rclockintrlist.tail = rclockcurlist.tail;
                rclockcurlist.head = NULL;
                rclockcurlist.tail = &rclockcurlist.head;
                send_interrupt();
        }
        if (isempty(rclockcurlist) &&
            !isempty(rclocknxtlist)) {

                /* Move recently-registered blocks to cur list. */

                rclockcurlist.head = rclocknxtlist.head;
                rclockcurlist.tail = rclocknxtlist.tail;
                rclocknxtlist.head = NULL;
                rclocknxtlist.tail = &rclocknxtlist.head;
                rclockgen = rcc_curgen + 1;
                P(rcc_mutex);
                rc_reg_gen(rclockgen);
                V(rcc_mutex);
        }
        if ((rcc_needctxtmask & (1 << me)) == 0) {

                /* Already did our part for this generation. */

                return;
        }
        if (isinvalid(rclockcswtchctr)) {

                /* Need to track quiescent state for this generation. */

                rclockcswtchctr = cswtchctr;
        }
        if (rclockcswtchctr == cswtchctr) {

                /* Still have not passed through a quiescent state. */

                return;
        }

        /* Passed through quiescent state. */

        P(rcc_mutex);
        rcc_needctxtmask &= ~(1 << me);
        rclockcswtchctr = INVALID_VALUE;
        if (rcc_needctxtmask != 0) {

                /* Still waiting for others to pass through quiescent state. */

                V(rcc_mutex);
                return;
        }

        /* Last CPU, finish current generation and start new one if needed. */

        rcc_curgen++;
        rc_reg_gen(rcc_maxgen);
        V(rcc_mutex);
}
```

The implementation of `rc_intr()` is as follows:

```
void rc_intr()
{
        dfree_t *dfp;

        while (!isempty(rclockintrlist)) {

                /* Free next memory block. */

                DISABLE();
                dfp = rclockintrlist.head;
                rclockintrlist.head = dfp->next;
                if (rclockintrlist.head == NULL) {
                        rclockintrlist.tail = &rclockintrlist.head;
                }
                ENABLE();
                free(dfp);
        }
}
```

The implementation of `rc_reg_gen()` is as follows:

```
void rc_reg_gen(int newgen)
{
        if (newgen > rcc_maxgen) {

                /* First time we have seen newgen. */

                rcc_maxgen = newgen;
        }
        if ((rcc_needctxtmask != 0) ||
            (rcc_maxgen < rcc_curgen)) {

                /* Generation in progress or done with newgen. */

                return;
        }

        /* Start a new generation! */

        rcc_needctxtmask = rcc_olmsk;
}
```

4.Performance

These sections compare the performance of read-copy update to the following locking mechanisms under conditions of low and high contention:

1. Simple spinlock.
2. Centralized reader-writer spinlock.
3. Distributed (cache-friendly) reader-writer spinlock.

Simple spinlock can be implemented as follows:

```
#define LOCKED   1
#define UNLOCKED 0

void P(char *lock)
{
        while (test_and_set(lock) == LOCKED) {
                while (*lock == LOCKED) {
                        continue;
                }
        }
}

void V(char *lock)
{

        *lock = UNLOCKED;
}
```

The `test_and_set()` primitive sets the low-order bit of the byte pointed to by its argument and returns the initial setting of that bit. There are many variations on the simple spinlock theme with different advantages and disadvantages over varying ranges of contention. Systems with weak memory-consistency models may need to use special primitives to ensure that the assignment in `V()` is committed to memory in a timely fashion.

Centralized reader-writer spinlock may be implemented as follows:

```
typedef struct {
        lock_t srw_lock;
        int srw_rdrq;   /* Read requests. */
        int srw_wrrq;   /* Write requests. */
        int srw_rdcp;   /* Read completions. */
        int srw_wrcp;   /* Write completions. */
} srwlock_t;

void PR(srwlock_t *l)
{
        int rdrq, wrrq;

        /*
         * Record new reader request and
         * capture writer request number.
         */

        P(&(l->srw_lock));
        (l->srw_rdrq)++;
        wrrq = l->srw_wrrq;
        V(&(l->srw_lock));

        /* Wait for any preceding writers to finish. */

        while (l->srw_wrcp != wrrq);
}

void
VR(srwlock_t *l)
{
        /* Record another read completion. */

        P(&(l->srw_lock));
        (l->srw_rdcp)++;
        V(&(l->srw_lock));
}

void PW(srwlock_t *l)
```

```
{
        int rdrq, wrrq;

        /*
         * Record new writer request and
         * capture both reader and writer
         * request number.
         */

        P(&(l->srw_lock));
        rdrq = l->srw_rdrq;
        wrrq = (l->srw_wrrq)++;
        V(&(l->srw_lock));

        /*
         * Wait for any preceding readers
         * and writers to finish.
         */
        while ((l->srw_rdcp != rdrq) ||
               (l->srw_wrcp != wrrq));
}

void VW(srwlock_t *l)
{
        /* Record another write completion. */

        P(&(l->srw_lock));
        (l->srw_wrcp)++;
        V(&(l->srw_lock));
}
```

The implementation of distributed reader-writer spinlock uses one simple spinlock for each CPU, and an additional simple spinlock for the writers. A CPU acquiring a read-side lock simply acquires its own lock. A CPU acquiring a write-side lock acquires the writer lock followed by each of the CPU's locks in CPU-number order.

4.1Low Contention

This section determines what conditions favor which locking primitives under low contention, that is, when the probability of two CPUs attempting to acquire the same lock at the same time is vanishingly small. The conditions examined are number of CPUs, ratio of cache-miss to cache-hit latencies, and fraction of critical sections requiring exclusive lock. This section uses the following nomenclature:

Symbol	Definition
f	Fraction of lock acquisitions that require exclusive access to the critical section.
n	Number of CPUs.
r	Ratio of t_s to t_f.
t_s	Time required to complete a "slow" access that misses the CPU's local cache
t_f	Time required to complete a "fast" access that hits the CPU's local cache

This analysis assumes modern microprocessor architecture, where the processor core and level-1 caches have insignificant latencies compared to that of level -2 cache hits and fetches from memory. Therefore, the overhead of instruction execution and of data references that hit the level-1 cache may be ignored. The slow accesses measured by t_f are level-2 cache hits, and the fast accesses measured by t_s are misses to main memory. Instruction fetches and references to the stack are assumed to always hit the level-1 cache.

4.1.1Overhead of Simple Spinlock

Under low contention, a simple spinlock requires one atomic operation to acquire and another to release[3]. If there are n CPUs, the chances are one out of n that this same CPU will have been the last to previously acquire the lock, and therefore have the locking data structure already present in its cache. Since contention is low, the lock data structure will always be in the CPU's cache upon release. Therefore, the overhead is:

[3] On CPUs with strong memory consistency, the release can be implemented as a simple store.

$$\frac{(n-1)t_s + (n+1)t_f}{n}$$ **Equation 1**

4.1.2 Overhead of Centralized Reader-Writer Spinlock

A centralized reader-writer spinlock acquisition consists of two back-to-back spinlock acquisition-release pairs. The first acquisition has the same probability of finding the data structure in the CPU's cache as does the simple spinlock, while the second acquisition is certain to hit the CPU's cache. The first critical section will check one counter (two if the is a write-acquisition) and update one counter. The second critical section will update one counter. Therefore, the overhead is:

$$\frac{(n-1)t_s + ((5+f)n+1)t_f}{n}$$ **Equation 2**

4.1.3 Overhead of Distributed Reader-Writer Spinlock

A distributed reader-writer spinlock read-side acquisition consists of a simple spinlock acquisition. However, there is a much greater chance that the relevant portion of the data structure is in the CPU's cache: for every lock access by this CPU, there are *(n-1)f* write accesses from other CPUs that affect this CPU's portion of the data structure. The probability that this CPU's portion of the data structure remains in the cache is thus:

$$\frac{1}{(n-1)f+1}$$ **Equation 3**

instead of the *1/n* probability for a simple spinlock. Analogously, the probability that some other CPU's portion of the data structure will already be in our cache when doing a write-side acquisition is given by:

$$\frac{f}{(n-1)f+1}$$ **Equation 4**

The expected time for a given CPU to do a read-side acquisition is then:

$$E_3 t_f + (1-E_3)t_s$$ **Equation 5**

where E_3 is Equation 3. This expands to:

$$\frac{(n-1)f t_s + ((n-1)f+2)t_f}{(n-1)f+1}$$ **Equation 6**

The expected time for a CPU to to write-side acquisition is the time required to acquire and release the writer-guard lock, plus the time to acquire this CPU's lock, plus the time to acquire the *(n-1)* other CPU's locks, plus the time to release all *n* CPU's locks:

$$E_1 + (E_3 t_f + (1-E_3)t_s) + (n-1)(E_4 t_f + (1-E_4)t_s) + n\,t_f$$ **Equation 7**

This expands to:

$$\frac{(n^2-1)t_s + (n+1)^2 t_f}{n}$$ **Equation 8**

The weighted average of Equations 5 and 7 gives the overall overhead for distributed reader-writer spinlocks:

$$\frac{\begin{pmatrix}\left(f^2n^3+2\left(f-f^2\right)n^2-f\,n+f^2-f\right)t_s\\+\left(f^2n^3+2f\,n^2+\left(2-f\right)n-f^2+f\right)t_f\end{pmatrix}}{f\,n^2+\left(1-f\right)n}$$ **Equation 9**

4.1.4Overhead of Read-Copy Update (Worst Case)

The overhead of read-copy update is zero on the read side, so the overall overhead is simply worst-case overhead plus the overhead of the underlying update mutex, all multiplied by the update fraction. The worst-case overhead is *n(n-1)* slow accesses[4] plus *(65n+28)* fast accesses. The worst case occurs when updates are so infrequent that they are handled separately:

$$\left(\left(n^2+1\right)\left(n-1\right)t_s+\left(65n^2+29n+1\right)t_f\right)f$$ **Equation 10**

4.1.5Overhead of Read-Copy Update (Best Case)

Read-copy updates are batched so that a second update adds minimal overhead when running concurrently with a previous update. The incremental cost of this second update is 11 more data references that are very likely to hit the cache. Therefore, the best-case overhead is just *(11t_ff)*. However, this overhead must be added to that of the mutual-exclusion primitive that synchronizes the updates. If this primitive is simple spinlock, the best-case overhead of read-copy update is:

$$\frac{\left(\left(n-1\right)t_s+\left(12n+1\right)t_f\right)f}{n}$$ **Equation 11**

It is possible to have many concurrent updates without danger of update-side lock contention if many different data structures, protected by many different locks, are being updated concurrently.

4.1.6Comparison of Overheads

Since the number of CPUs and memory latencies are fixed by the machine being used, the question that will be answered is "what values of *f* favor which locking primitive?". Therefore, this section finds expressions for the value of *f* that causes pairs of Equations 1, 2, 9, and 10 to have equal values.

Since Equation 2 is strictly greater than Equation 1, simple spinlocks are always superior to centralized reader-writer spinlocks under low contention.

Simple spinlock and distributed reader-writer spinlock are compared by setting Equations 1 and 9 equal and solving for *f*, which gives:

$$f=\frac{n\sqrt{\left(n-1\right)r+n+1}\sqrt{\left(5n-5\right)r-3n+5}-\left(n^2+n-2\right)r-n^2+n-2}{\left(2n^3-4n^2+2\right)r+2n^3-2}$$ **Equation 12**

The breakeven value of *f* varies roughly inversely with the number of CPUs *n*. Distributed reader-writer spinlock is faster than simple spinlock for sufficiently small *f*. For any update fraction *f*, there is a number of CPUs *n* above which simple spinlock will outperform distributed reader-writer spinlock, but only if contention remains low.

Similarly, for simple spinlock and worst-case read-copy update (Equations 1 and 10):

$$f=\frac{\left(r+1\right)n-r+1}{r\,n^3+\left(65-r\right)n^2+\left(29+r\right)n+1-r}$$ **Equation 13**

[4] This can be reduced to *O(n)* accesses on NUMA systems, but the exposition and analysis is beyond the scope of this paper. A similar optimization may be performed for SMP systems, but this optimization is ineffective for small *n*.

The breakeven value of *f* varies roughly as the inverse square of the number of CPUs *n*. Worst-case read-copy update is faster than simple spinlock for sufficiently small *f*.

Best-case read-copy update does much better compared to simple spinlock (Equations 1 and 11):

$$f = \frac{(r+1)n - r + 1}{(r+12)n + 1 - r}$$ **Equation 14**

The breakeven value of *f* approaches *((r+1)/(r+12))* as the number of CPUs *n* increases. For the relatively large values of *r* found on large-scale multiprocessors, best-case read-copy update outperforms simple spinlock unless the workload consists almost entirely of updates.

For centralized and distributed reader-writer spinlock (Equations 2 and 9):

$$f = \frac{\left(\begin{array}{c} n\sqrt{\left((5n^2 - 10n + 5)r^2 + (10n^2 - 30n + 20)r + 21n^2 - 20n\right)} \\ -(n^2 + n - 2)r + 3n^2 - 2n - 2 \end{array} \right)}{(2n^3 - 4n^2 + 2)r + 2n^3 - 2n^2 + 2n - 2}$$ **Equation 15**

This varies inversely with the number of CPUs. Distributed reader-writer spinlock is faster than centralized for sufficiently small *f*. Similarly, for any update fraction *f*, there is a number of CPUs *n* above which centralized reader-writer spinlock will be superior to distributed reader-writer spinlock.

For centralized reader-writer spinlock and worst-case read-copy update (Equations 2 and 10):

$$f = \frac{(r+5)n - r + 1}{r\,n^3 + (65 - r)n^2 + (28 + r)n + 1 - r}$$ **Equation 16**

The breakeven value of *f* varies roughly as the inverse square of the number of CPUs *n*. Worst-case read-copy update is faster than centralized reader-writer spinlock for sufficiently small *f*.

Again, best-case read-copy update does much better (Equations 2 and 11):

$$f = \frac{(r+5)n - r + 1}{(r+11)n + 1 - r}$$ **Equation 17**

The breakeven value of *f* approaches *((r+5)/(r+11))* as the number of CPUs *n* increases. For the relatively large values of *r* found on large-scale multiprocessors, best-case read-copy update is better than centralized reader-writer spinlock unless the workload consists almost entirely of updates.

For distributed reader-writer spinlock and worst-case read-copy update (Equations 9 and 10):

$$\frac{\sqrt{\left(\begin{array}{c} (n^4 - 6n^3 + 13n^2 - 12n + 4)r^2 \\ +(134n^3 - 342n^2 + 104n + 104)r \\ +4481n^2 + 3492n + 676 \end{array} \right)} - (n^2 - 3n + 2)r + 63n + 30}{(2n^3 - 6n^2 + 8n - 4)r + 128n^2 - 72n - 56}$$ **Equation**

18

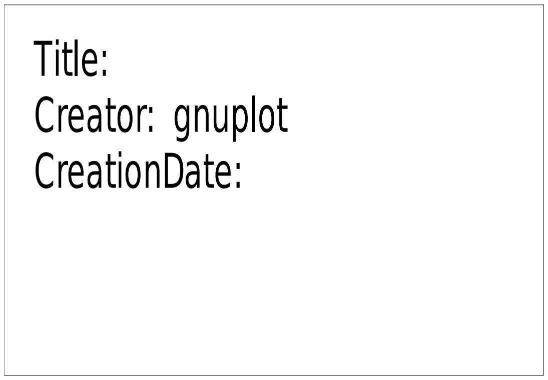
The breakeven value of *f* varies roughly as the inverse square of the number of CPUs *n*. Worst-case read-copy update is faster than distributed reader-writer spinlock for sufficiently small *f*.

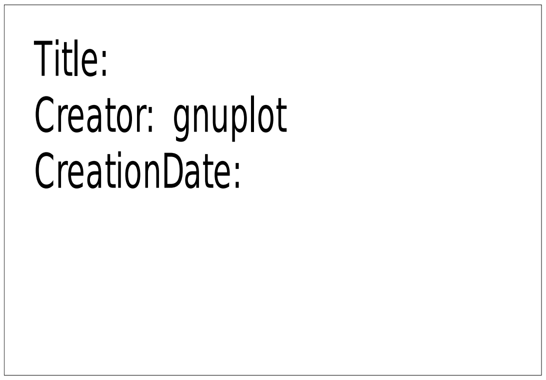
Finally, distributed reader-writer spinlock is always slower than best-case read-copy update, as can be seen by comparing Equations 9 and 11. The only exception to this is for the few cases where:

$$r \le \frac{11 - n}{n - 2}$$ **Equation 19**

in which case, large values of *f* favor distributed reader-writer spinlock. However, in these cases, simple spinlock is better than both distributed reader-writer spinlock and best-case read-copy update.

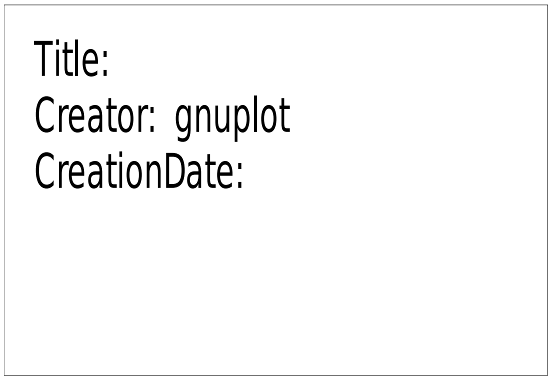
The relationships between traditional primitives and worst-case read-copy update is shown in these plots:

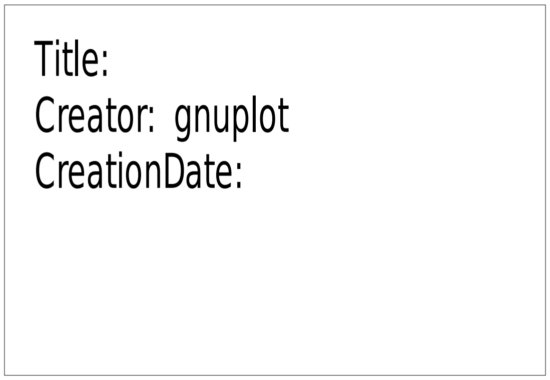




In the regions labelled "A", simple spinlock and centralized reader-writer lock are faster than either distributed reader-writer lock or worst-case read-copy update. In the regions labelled "B", distributed reader-writer lock is fastest. In the regions labelled "C", worst-case read-copy update is fastest

The relationship between traditional primitives and best-case read-copy update is shown in these plots:





As noted earlier, best-case read-copy update is almost always faster than distributed reader-writer lock. In the regions labelled "A", simple spinlock and centralized reader-writer spinlock are faster than best-case read-copy update. In the regions labelled "C", best-case read-copy update is faster. Note that if a program's update frequency rises with the number of CPUs, then at some point, the best-case behavior of read-copy update will dominate. The more heavily read-copy update is used, the better it performs.

Also, as noted earlier, simple spinlock is always faster than centralized reader-writer lock. However, centralized reader-writer lock may be chosen because it allows concurrent readers. The upper of each of the pairs of lines correspond to centralized reader-writer lock, and the lower of the pair to simple spinlock.

4.2High Contention--Throughput

This section examines the effects of update overhead on programs using locking primitives under high read-side contention. High read-side contention means that each CPU is waiting to enter a read-side critical section, unless it is already in a read critical section, in an update critical section, or waiting to enter an update critical section. The parameters varied include number of CPUs, ratio of read critical-section length to update critical-section length, and fraction of critical sections performing updates. Update overhead is made up of the time required for CPUs to leave their read-side critical sections and the time required for the update itself. During this time period, conventional locking primitives will prevent CPUs from entering read-side critical sections.

This section uses the following nomenclature:

Symbol	Definition
f	Fraction of lock acquisitions that require exclusive access to the critical section.
n	Number of CPUs.

r	Ratio of $_u$ to $_r$.
$_r$	Mean read-critical-section service rate.
$_u$	Mean update-critical-section service rate.

This section looks at two read-service-time distributions: deterministic and exponential. Deterministic service times can be a reasonable approximation for short critical sections that have consistent cache-miss rates. Exponential service times can be a reasonable approximation when the critical sections are subject to cache misses, interrupts, and preemption. The read service-time distribution is important because the time required for *n* CPUs to leave their read-side critical sections will be the maximum of *n* random variables taken from the corresponding distribution.

The primitives are compared by comparing the rates at which they complete read-side critical sections. These rates are expressed as a fraction of the ideal rate that would be achieved if updates did not block read-side critical sections. The update fraction *f* is assumed to be small enough that two CPUs are almost never attempting to do concurrent updates.[5]

The number of read-critical-sections per update-critical-sections in terms of the update fraction *f* is *(1-f)/f.* On an *n*-CPU machine, *n* of these read-critical-sections will be started as soon as the update completes. The remaining:

$$\frac{1-f-f\,n}{f} \qquad \textbf{Equation 20}$$

read-side critical sections will have all been at least started, on average, in time:

$$\frac{1-f-f\,n}{f\,n\,\mu_r} \qquad \textbf{Equation 21}$$

If the time required for the read-side critical sections to complete once a CPU starts attempting to do an update is *T*, then the time required to complete both the read-side critical sections and the update is:

$$\frac{1-f-f\,n}{f\,n\,\mu_r}+T+\frac{1}{\mu_u}=\frac{(1-f-f\,n)\mu_u+T\,f\,n\,\mu_r\mu_u+f\,n\,\mu_r}{f\,n\,\mu_r\mu_u} \qquad \textbf{Equation 22}$$

In the ideal case, the read-side critical-section completion delay would be zero and the update would proceed in parallel with the reads:

$$\frac{1-f-f\,n}{f\,n\,\mu_r}+\frac{1}{n\mu_u}=\frac{(1-f-f\,n)\mu_u+f\,\mu_r}{f\,n\,\mu_r\mu_u} \qquad \textbf{Equation 23}$$

Dividing these gives a measure of efficiency:

$$\frac{(1-f-f\,n)\mu_u+f\,\mu_r}{(1-f-f\,n)\mu_u+T\,f\,n\,\mu_r\mu_u+f\,n\,\mu_r} \qquad \textbf{Equation 24}$$

4.2.1Deterministic Read Service-Time Distribution

For a perfectly deterministic read service-time distribution, the time required for the *(n-1)* non-updating CPUs to leave their critical sections is just the read-size service time *1/* $_r$ if the lock primitive favors readers (as the distributed reader-writer spinlock does).[6] Its efficiency is thus:

$$\frac{(1-f-f\,n)\mu_u+f\,\mu_r}{(1-f)\mu_u+f\,n\,\mu_r} \qquad \textbf{Equation 25}$$

[5] The general solution for large *f* involves an M/G/1 queuing system where the service-time distribution is a function of queue length. Solution of such a system is beyond the scope of this paper.

[6] Note that the read-side critical sections would execute in lockstep in this case.

A primitive that favors neither readers or writers (such as the centralized reader-writer spinlock) will see zero time with probability *1/n* and *1/* $_r$ time with probability *(n-1)/n*. Its efficiency is thus:

$$\frac{(1-f-f\,n)\mu_u+f\,\mu_r}{\mu_u+f\,n\,\mu_r} \qquad \textbf{Equation 26}$$

This is better than the efficiency of distributed reader-writer spinlock, however, note that this holds only when *1/* $_r$ is large compared to the lock-acquisition overhead.

Read-copy update allows updaters to run in parallel with readers, and therefore achieves the ideal throughput. Its efficiency is thus exactly 1.

4.2.2Exponential Read Service-Time Distribution

The time required for each of *(n-1)* read-side critical sections to complete is just the maximum of *(n-1)* samples taken from the corresponding service-time distribution, due to the memoryless property of the exponential distribution. This maximum is given by:

$$\frac{1}{(n-1)!\,\mu_r}\sum_{k=0}^{n-2}\frac{-1^k}{(k+1)^2}\binom{n-1}{k} \qquad \textbf{Equation 27}$$

Centralized and distributed reader-writer spinlock behave similarly in this case, with efficiency given by:

$$\frac{(1-f-f\,n)\mu_u+f\,\mu_r}{(1-f-f\,n)\mu_u+\frac{f\,\mu_u}{(n-1)!}\sum_{k=0}^{n-2}\frac{-1^k}{(k+1)}\binom{n}{k+1}+f\,n\,\mu_r} \qquad \textbf{Equation 28}$$

Read-copy update again achieves an efficiency of 1.

4.3High Contention--Read-Side Latency

The following table compares the search time of a three-element linked list given various synchronization schemes on 80486 CPUs:

Mechanism	Time (seconds)	Savings
Unix Signal	51.00	98%
Atomic Instruction	2.00	53%
Fast No-Preempt	1.46	36%
Disable Interrupts	1.08	13%
Read-Copy Update	0.94	0%

The “Savings” column lists the read-side savings that gained by switching to read-copy update. The “Atomic Instruction” row is equivalent to the overhead of distributed reader-writer spinlock.

This table illustrates one of read-copy update’s strengths: that it imposes no overhead on reading processes. To appreciate this strength, consider a protocol routing module designed for single-CPU use. Such a module will often disable interrupts during routing-table searches in order to avoid conflicts with routing-table updates. A multi-CPU routing protocol module that uses read-copy update does not need to disable interrupts, and thus can be said to exhibit *negative* 13% synchronization overhead compared to its single-CPU counterpart.

5.Conclusions

Read-copy update can provide significant speedups where reads are more frequent than updates, where stale data can be tolerated, and where memory size is not a limiting factor. Since read-copy updates are batched, heavy use increases their efficiency. The space of numbers of CPUs, ratio of overheads of cache hits and misses, and update fraction divides into well-defined regions in which one of simple spinlock, distributed reader-writer lock, or read-copy update dominates.

When read-copy update is heavily used, it is significantly faster than traditional locking primitives under almost all conditions.

6.Acknowledgements

We owe thanks to Brent Kingsbury, Jan-Simon Pendry, Dave Wolfe, Stuart Friedberg, Sekhar Chandrasekhar, Doug Miller, and Ken Dove for many helpful discussions. 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.

7.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.

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

[Manber82] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure, Department of Computer Science TR#82-01-01, University of Washington, Seattle, WA, January 1982.

[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.

[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.