Selecting Locking Primitives for Parallel Programs

Paul E. McKenney
Sequent Computer Systems, Inc.

Abstract

The only reason to parallelize a program is to gain performance. However, the synchronization primitives needed in parallel programs can consume exececessive memory bandwidths, can be subject to memory latencies, consume excessive memory, and result in unfair access or even starvation. These problems can overwhelm the performance benefits of parallel execution. Therefore, it is necessary to understand these performance implications of synchronization primitives in addition to their correctness, liveness, and safety properties.

This paper presents a pattern language to assist you in selecting synchronization primitives for parallel programs. This pattern language assumes you have already chosen a locking design, perhaps by using a locking design pattern language [McK95b].

1 Overview

A lock-based parallel program uses synchronization primitives to define critical sections of code in which only one CPU at a time may execute concurrently.

For example, Figure 1 presents a fragment of parallel code to search and update a linear list. In this C-code example, the `lt_next` field links the individual elements together, the `lt_key` field contains the search key, and the `lt_data` field contains the data corresponding to that key.

The section of code between the `S_LOCK()` and the `S_UNLOCK()` primitives is a critical section. Only one CPU at a time may be executing in this critical section. If there is not enough work outside of the critical section to keep the rest of the CPUs in the system busy, the excess CPUs will be consumed by contention. These CPUs will wait uselessly to acquire the `looktab_mutex` lock.

The two most useful measures of contention experienced for a particular critical section are: (1) the fraction wall-clock time during which one CPU is executing in that critical section, and (2) the fraction of total CPU or process time spent waiting to enter that critical section. Use the first measure when you want to predict the load level at which contention will limit speedup [McK95a]. Use the second measure to help select locking primitives.

A program's granularity of parallelism is related to the duration of its critical sections. One useful measure of granularity is ratio of the CPU time consumed by the critical sections to that consumed by the locking primitives. If the granularity is too great, speedup will be limited by contention or data skew. If the granularity is too small, speedup will be limited by the overhead of the locking primitives.

The code shown in the figure might simply read the list, or it might modify it. The ratio of the number of reads to the number of modifications (or writes) is called the read-to-write ratio. Programs with high read-to-write ratios can use specialized locking primitives that allow readers to proceed in parallel. The meaning of a "high" read-to-write ratio depends on the context. Common meanings are "much greater than one" or "much greater than the number of CPUs on the system".

The pattern language in this paper will help you to determine what kind of lock the `looktab_mutex` should be. This paper considers a few straightforward test-and-set, queued, and reader/writer locks. This set of locking primitives will handle most typical situations.

Section 2 gives an overview of locking design patterns [McK95b]. Section 3 describes the forces common to all of the patterns. Section 4 overviews contexts in which these patterns are useful. Section 5 presents several indexes to the patterns. Section 6 presents the patterns themselves.

2 Overview of Locking Design Patterns and Forces

Although design and implementation are often treated as separate activities, they are often deeply intertwined, so it is helpful to understand the relationships between them. Therefore, this section presents a brief

```
/* Element of list being searched. */

        typedef struct looktab {
                struct looktab_t *lt_next;
                int     lt_key;
                int     lt_data;
        } looktab_t;

        /* List lock, header and pointer. */

        slock_t looktab_mutex;
        looktab_t *looktab_head = NULL;
        looktab_t *p;

        . . .

        /*
         * Look up a looktab element and
         * examine it.
         */

        S_LOCK(&looktab_mutex);

        p = looktab_head;
        while (p != NULL) {
                if (p->lt_key == key) {
                        break;
                }
                p = p->lt_next;
        }

        if (p != NULL) {

                /*
                 * insert code here to examine or
                 * update the element.
                 */
        }

        S_UNLOCK(&looktab_mutex);
```

Figure 1: Example of a Critical Section

overview of design-level patterns and the forces that act on them. For more details, refer to a locking design pattern language [McK95b].

2.1 Overview of Locking Design Patterns

This paper refers to the following locking design patterns:

Sequential Program: A design with no parallelism, offering none of the benefits or problems associated with parallel programs.

Code Locking: A design where locks are associated with specific sections of code. In object-oriented designs, code-locking locks classes rather than instances of classes.

Data Locking: A design where locks are associated with specific data structures. In object-oriented designs, data-locking locks instances rather than classes.

Data Ownership: A design where each CPU or process "owns" its share of the data. This means that a CPU does not need to use any locking primitives to access its own data, but must use some special communications mechanism to access other CPUs' or processes' data.[1]

Parallel Fastpath: A design that uses an aggressive locking pattern for the majority of its workload (the fastpath), but that must use a more conservative pattern for the remaining small part of its workload.

Reader/Writer Locking: A variant of parallel fastpath allows readers to proceed in parallel (the fastpath), but that forces writers to exclude readers as well as each other.

Hierarchical Locking: A design that uses a global lock for global changes, but that also has a per-element lock for localized changes. In this case, the localized changes are the fastpath.

Allocator Caches: A design that uses per-CPU or per-process caches of memory (or of whatever resource is being allocated)[MS93]. Allocating from

[1] The Active Object pattern [Sch95] describes an object-oriented approach to this sort of communications mechanism. More complex operations that atomicly update data owned by many CPUs must use a more complex approach such as two-phase commit [Tay87].

or deallocating to the cache is the fastpath. Allocations and deallocations that find the cache empty or full, respectively, must leave the fastpath in order to use a more conservatively-locked global allocator.

Critical-Section Fusing: A design that combines critical sections in order to reduce synchronization overhead.

Critical-Section Partitioning: A design that splits critical sections in order to reduce contention.

These design patterns define much of the context in which the locking-primitive patterns must be used.

2.2 Overview of Locking Design Forces

The forces acting at the design level are:

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

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

Overhead: A monoprocessor version of a given parallel program would not need synchronization primitives. Therefore, any time consumed by these primitives is overhead that does not contribute directly to the useful work that the program is intended to accomplish. Note that the important measure is the relationship between the synchronization overhead and the serial overhead–critical sections with greater overhead may tolerate synchronization primitives with greater overhead.

Read-to-Write Ratio: A data structure that is mostly rarely updated may often be protected with lower-overhead synchronization primitives than may a data structure with a high update rate.

Economics: Budgetary constraints can limit the number of CPUs available regardless of the potential speedup.

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

This greater complexity often (but not always!) translates to higher development and maintenance costs. Therefore, budgetary constraints can limit the number and types of modifications made to an existing program – a given degree of speedup is worth only so much time and trouble.

Although the forces acting at the implementation level are closely related to those acting at the design level, there are some important differences noted in Section 3.

3 Forces

When selecting locking primitives, you must consider resource-consumption forces imposed from below by the system on which the program will run and algorithmic forces imposed from above by the program's design.

3.1 Forces from the Machine

If your only goal is squeezing the maximum conceivable performance out of the hardware, then you are faced with size and latency forces for each and every hardware resource making up the machine. This includes memory, caches, data paths, general registers, ALUs, TLBs, branch-prediction logic, and speculation hardware. This overabundance of forces is what makes performance programming so difficult–great quantities of time, effort, and cleverness are needed to balance them optimally.

Fortunately, CPU speed has been increasing exponentially compared to just about everything else [HJ91, SC91].[2] This means that applications are frequently limited by their memory accesses, which in turn means that you can usually ignore the all but the following forces:

Memory Latency: The memory-latency force is analogous to the synchronization-overhead force

[2] One possible exception to this trend has been wide-area-network bandwidth, particularly in areas of the world undergoing telecommunications deregulation. However, high latency usually has a greater impact on locking primitives than does low bandwidth, given the relatively small size of the data required to represent a lock. Latency is very sharply bounded below by the speed of light, at least in the absence of some unexpected breakthroughs in the field of physics, and is therefore frequently the controlling factor.

that affects selection of locking designs. All else being equal, you should use more complex distributed locking primitives on systems whose memory-access latencies are large compared (for example) to their instruction-execution latencies.

Memory Bandwidth: Some locking primitives consume excessive memory bandwidth under high contention. You should use more complex primitives that consume little memory bandwidth if your program is prone to contention.

Memory Size: A program that uses too much memory for a given machine can thrash the caches, resulting in high-latency accesses to main memory. Worse yet, the program's working set might exceed the size of main memory, resulting in extremely slow accesses to secondary storage.

You should make sure that the locking primitives you select don't consume so much memory that they actually degrade overall system performance. Even though the size of the lock data structures are usually quite small, aggressively-parallel designs can require very large numbers of locks.

Note that the minimum size of lock-primitive data structures varies from system to system. If you are concerned about memory consumption, check the actual sizes on your system.

3.2 Forces from the Design

The design forces acting on the selection of locking primitives are read-to-write ratio and fairness:

Read-to-Write Ratio: You can protect an infrequently-changed data structure with lower-overhead locking primitives than can be used for a data structure with a high update rate.

The dividing line between a "high" and a "low" read-to-write ratio depends on the locking primitive. Common dividing lines are: (1) much greater than one (to get any benefit at all from parallel readers), and (2) much greater than the number of CPUs (to get any benefit at all from parallel readers when the write-side cost is linear with the number of CPUs).

Fairness: If your design requires some degree of fair access to critical sections (as would a real-time system), you should use fair primitives.

Note that although the design-space speedup and economics forces are very important, they are controlled more by the overall design than by the specific primitives that you select. This is the reason for the old adage: "Performance must be designed into the system rather than being added on as an afterthought". Poor selection of primitives can cause a good design to perform poorly, but good primitives cannot normally make up for a poor design.

The design-space contention force becomes an element of the implementation-space context. Low contention must be designed into a parallel program.

The design-space synchronization-overhead force is subsumed by the implementation-space memory-latency and memory-bandwidth forces.

The design-space complexity force is a special case. It is true that many of the locking primitives can be difficult to implement. However, this difficulty may be amortized over many uses. This paper assumes that the primitives already exist, and therefore considers problems with their use, not with their implementation.

4 Overview of Context

The major elements of a parallel program's context are the degree of contention and the granularity of parallelism.

If your design has sufficiently-low contention, its granularity of parallelism is irrelevant. Therefore, contention and granularity taken together produce three contexts: (1) low contention, (2) high contention with coarse-grained parallelism, and (3) high contention with fine-grained parallelism. These contexts are discussed in the following sections.

4.1 Low Contention

A critical section enjoys low contention if CPUs and processes almost always gain immediate access to the critical section without spinning or blocking. A low-contention design is very desirable, as it can achieve high speedup. Unfortunately, Amdahl's law states that some part of the resulting program will be inherently serial. This portion will limit speedup, and will cause high contention if too many CPUs are executing the program concurrently.[3]

Nonetheless, real systems have a limited number of CPUs, so your design might have low contention on such a system. In this case, the locking primitives do

[3] "Embarrassingly parallel" algorithms allow the number of CPUs to be increased without bound only if the size of the problem is also increased without bound. For a fixed problem size, even embarrassingly parallel algorithms will exhibit high contention if too many CPUs are applied to them.

not have to do much work, so you have much more latitude when balancing forces–your choice of locking primitive is usually not critical. However, memory size can be critical if you have partitioned your design so finely that you need huge numbers of locks. In addition, memory latency can be critical in hard real-time systems.

Locking design patterns most likely to enjoy low contention include data ownership, allocator caches, parallel fastpath, hierarchical locking, and data locking.

4.2 High Contention, Coarse-Grained Parallelism

If performance is your primary concern, you should redesign a high-contention, coarse-grained program in order to lower the contention and increase the speedup. You can often accomplish this by using the critical-section-partitioning design pattern.

If you choose not to redesign, you must select locking primitives with careful attention to the fairness and read-to-write-ratio forces. If read-to-write ratio is high, you may be able to achieve lower levels of contention without a major redesign through use of the reader/writer locking design pattern.

4.3 High Contention, Fine-Grained Parallelism

Again, if performance is your primary concern, you should redesign to lower both contention and synchronization overhead, perhaps by applying the data-ownership, parallel-fastpath, reader/writer-locking, or allocator-caches design patterns.

If you choose not to redesign, you must carefully balance the memory-latency, fairness, and read-to-write-ratio forces when you select locking primitives. If read-to-write ratio is high compared to the number of CPUs, you may be able to achieve high speedups without a major redesign by using the reader/writer locking design pattern implemented with distributed reader-writer locks.

5 Index to Patterns for Selecting Locking Primitives

This section contains indexes based on relationships between the patterns (Section 5.1), forces resolved by the patterns (Section 5.2), and problems commonly encountered in parallel programs (Section 5.3).

5.1 Pattern Relationships

Section 6 presents the following patterns:

1. Test-and-Set Lock (6.1)
2. Queued Lock (6.2)
3. Queued Reader/Writer Lock (6.3)
4. Counter Reader/Writer Lock (6.4)
5. Distributed Reader/Writer Lock (6.5)

Relationships between these patterns are shown in Figure 2 and are described in the following paragraphs.

The test-and-set lock and the queued lock provide simple mutual exclusion.

The queued lock and the queued reader/writer lock tolerate extreme contention without imposing excessive traffic on the system bus.

The queued reader/writer lock, counter reader/writer lock, and distributed reader/writer lock allow readers to proceed in parallel. The distributed reader/writer lock in addition can operate efficiently in face of high read-side contention and fine-grained parallelism.

5.2 Force Resolution

Table 5.2 compares how each of the patterns resolves each of the forces. Plus signs indicate that a pattern resolves a force well. For example, test-and-set lock resolves memory size perfectly because test-and-set locks can be as small as one bit in size.

Minus signs indicate that a pattern resolves a force poorly. Again, test-and-set lock provides extreme examples with Memory latency since these locks can thrash under high contention, read-to-write ratio because these locks do not allow readers to proceed in parallel, and fairness because these locks provide no fairness mechanism.

Question marks indicate that the quality of resolution is quite variable. Distributed reader/writer lock enjoys extremely low memory latency and memory bandwidth, but only if the read-to-write ratio is very high compared to the number of CPUs in the system. If the read-to-write ratio is low, the memory latency and bandwidth can severely degrade the lock's performance, as well as that of the entire system.

The ratings for memory latency and memory bandwidth assume worst-case workload. Worst-case workload for memory latency is low contention, while the worst-case workload for memory bandwidth is high contention. The reason for this difference is that the

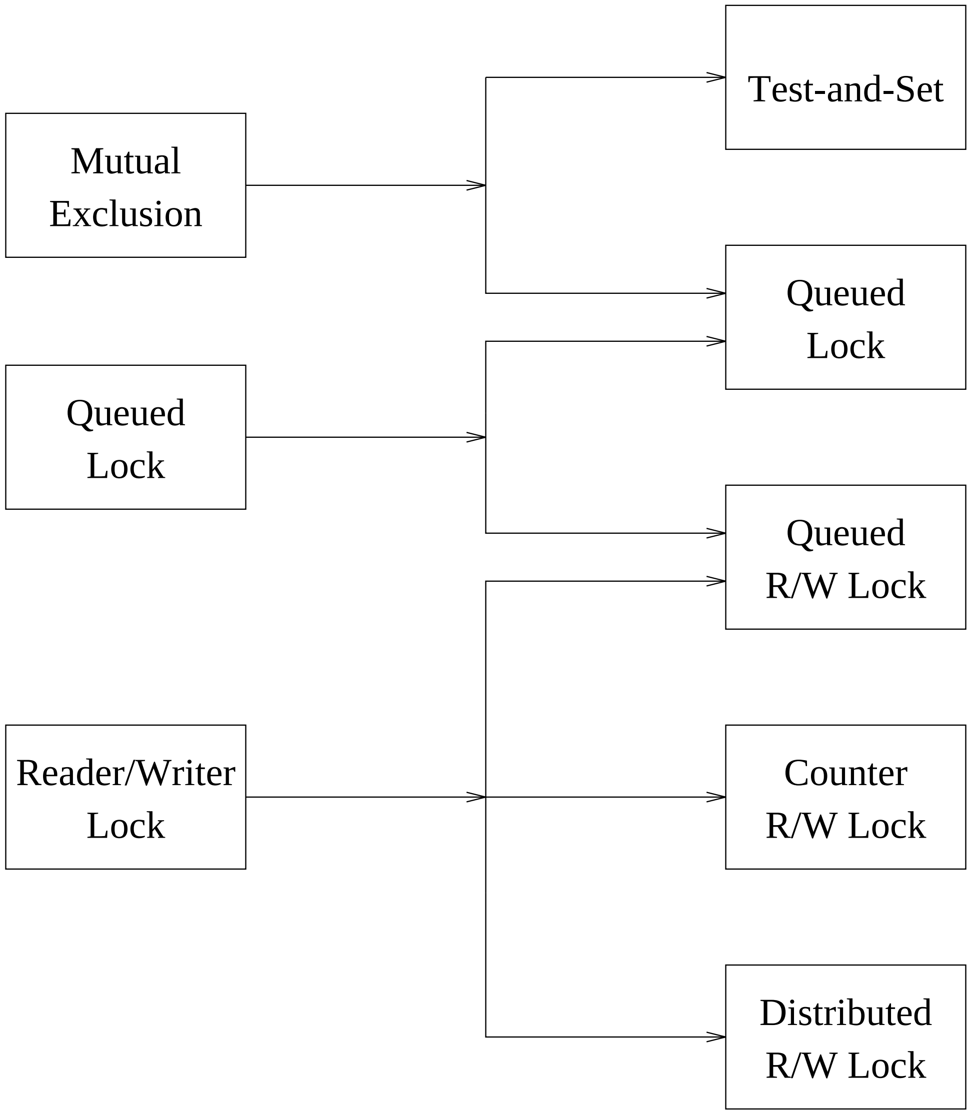

Figure 2: Relationship Between Pattern

Mem. Latency	Mem. B/W	Mem. Size	R/W	Fairness	Pattern
+	− − −	+ + +	− − −	− − −	Test-and-Set Lock (6.1)
−	+	+	− − −	+ + +	Queued Lock (6.2)
−−	+	0	++	+ + +	Queued Reader/Writer Lock (6.3)
0	− − −	++	++	+ + +	Counter Reader/Writer Lock (6.4)
++?	++?	− − −	+ + +	+	Distributed Reader/Writer Lock (6.5)

excessive memory-latency degradations that are often observed at high levels of contention are usually due to excessive memory-bandwidth consumption.

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

5.3 Fault Table

Use Table 5.3 to locate a replacement pattern for a pattern that is causing more problems than it is solving.

6 Patterns for Selecting Locking Primitives

6.1 Test-and-Set Locks

Problem What locking primitive should you use if memory is scarce and you do not need the functionality of more complex locking primitives?

Context A parallel program where contention is low or where fairness and performance is not crucial.

Forces

- Memory Size (+ + +): The size of the lock can be as small as a single bit.

- Memory Latency (+): Successful acquisition uses a single memory-reference instruction. On the other hand, the lock referenced by this instruction is likely to have been most-recently referenced by some other CPU. Therefore, this single reference is likely to be quite expensive on systems with lots of caching.

- Memory Bandwidth (− − −): If a large number of CPUs try to acquire the lock at the same time, each of the losing CPUs will have issued a memory transaction whose only effect is to inform the CPU that it lost. This useless memory traffic can cause severe performance problems if there are a large number of CPUs.

- Read-to-Write Ratio (− − −): A simple test-and-set lock will never allow readers to proceed in parallel.

- Fairness (− − −): A simple test-and-set lock has no provision for fairness. It is quite possible that an extremely unlucky CPU or process might repeatedly fail to acquire the lock.

Solution Use a locking primitive based on atomic test-and-set machine instructions as shown in Figure 3. 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.

```
#define LOCKED   1
#define UNLOCKED 0

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

void
S_UNLOCK(char *lock)
{
        *lock = UNLOCKED;
}
```

Figure 3: Test-and-Set Lock Implementation

If contention is low, this implementation of `S_LOCK()` will execute a single test-and-set instruction.

A number of researchers have worked on variants of test-and-set locks that better tolerate high contention, including test-and-test-and-set [SR84] and randomized exponential backoff [And90]. Although these can be useful, you should not add them to your toolbox until well after you have added a fast and simple test-and-set lock and a robust queued lock.

Resulting Context A program with locking primitives that consume little memory and cause little memory traffic under light load. However, this program will be prone to unfairness and excessive memory bandwidth, and will be unable to allow readers to proceed in parallel.

Design Rationale The simple implementation pays off with low memory and CPU overhead when contention is low. If your design enforces low contention (as might be enforced by data locking, data ownership, parallel fastpath, hierarchical locking, and allocator caches), but does not have a concept of readers that can proceed in parallel, then simple test-and-set locks can be the primitive of choice.

Old Pattern	Problem	Pattern to use
Test-and-Set Lock (6.1)	Locking primitives start consuming excessive memory bandwidths under heavy load.	Queued Lock (6.2)
Test-and-Set Lock (6.1) Queued Lock (6.2)	Program suffers from contention under heavy load, its read-to-write ratio is high, and the program uses fine-grained parallelism.	Distributed Reader/Writer Lock (6.5)
Test-and-Set Lock (6.1) Queued Lock (6.2)	Program suffers from contention under heavy load, its read-to-write ratio is high, and program uses coarse-grained parallelism.	Queued Reader/Writer Lock (6.3) Counter Reader/Writer Lock (6.4) Distributed Reader/Writer Lock (6.5)
Queued Lock (6.2) Queued Reader/Writer Lock (6.3) Counter Reader/Writer Lock (6.4)	Program's locking primitives suffer from memory latency under low load, but the program never suffers from high contention even under heavy load.	Test-and-Set Lock (6.1) Distributed Reader/Writer Lock (6.5)
Distributed Reader/Writer Lock (6.5)	Program's write-side locking primitives are too slow even under light load, and the program uses coarse-grained parallelism.	Queued Reader/Writer Lock (6.3) Counter Reader/Writer Lock (6.4)
Distributed Reader/Writer Lock (6.5)	Program's write-side locking primitives are too slow even under light load, and the program has a low read-to-write ratio.	Test-and-Set Lock (6.1) Queued Lock (6.2)

Table 1: Lock-Primitive Fault Table

6.2 Queued Locks

Problem What locking primitives should you use if your design can encounter high contention and if fairness is also important? "High" contention in this context means that there is often more than one CPU or process waiting to acquire the lock.

Context A parallel program where contention is high and where fair access to critical sections is important. This program must run on a machine that has atomic instructions capable of implementing a queued lock (either `compare_and_swap` or `load_linked` and `store_conditional` will suffice).

Forces

- Fairness (+ + +): The queue guarantees that CPUs will be admitted to the critical section in strict FIFO order.
- Memory Bandwidth (+): Use of a queue allows each CPU waiting on the lock to spin on a separate location of memory. This separate spinning avoids the bad behavior that test-and-set primitives are prone to at high contention.
- Memory Size (+): Queued locks carry more state and thus require more memory than do simple test-and-set locks. Test-and-set locks require O(L) memory, where L is the number of locks in the system, while queued locks require O(L+N) memory, where N is the number of CPUs. The memory required is still modest, with only a few bytes required for each lock and for each queue element.
- Memory Latency (−): Queued locks must access several disjoint memory locations that are shared by multiple processes. These additional memory references cause queued locks to be slower than simple test-and-set locks.
- Read-to-Write Ratio (− − −): A simple queued lock will never allow readers to proceed in parallel.

Solution Use a queued-lock primitive such as the MCS lock shown in Figure 4 [MCS91a]. The idea behind the queued lock is to give each spinning CPU its own queue element to spin on. These elements are linked together in FIFO order. The lock is granted in turn to each element of the list, thereby guaranteeing fair access.

```
typedef struct qnode *lock_t;
typedef struct qnode_s {
        lock_t *next;
        bool_t locked;
} qnode_t;

void
S_LOCK(lock_t *l, qnode_t *i) {
        qnode_t *pred;

        i->next = NULL;
        pred = fetch_and_store(l, i);
        if (pred != NULL) {
                i->locked = TRUE;
                pred->next = i;
                while (i->locked) {
                        continue;
                }
        }
}

void
S_UNLOCK(lock_t *l, qnode_t *i) {
        if (i->next == NULL) {
                if (compare_and_swap(l,
                                     i,
                                     NULL)) {
                        return;
                }
                while (i->next == NULL) {
                        continue;
                }
        }
        i->next->locked = FALSE;
}
```

Figure 4: Queued-Lock Implementation

The `fetch_and_store()` atomic primitive stores its second argument into the location pointed to by its first, returning the overwritten value.

The `compare_and_swap()` atomic primitive tests that the location pointed to by the first argument is equal to the value passed in the second argument. If it is, it stores the third argument into the location pointed to by the first. It returns `TRUE` if it in fact does the swap.

Researchers have proposed a number of variants to the queued lock. Lim and Agarwal [LA93] proposed a reactive lock that switches between a test-and-set and a queued mode depending on the level of contention it experiences. Wisniewski et al. [WKS94] proposed a lock that allows limited out-of-order service to prevent active processes from waiting behind blocked processes.

Resulting Context A program with locking primitives that enforce fairness and limit their memory-bandwidth load. However, this program will suffer reduced performance at low loads compared to an equivalent program that uses test-and-set locks.

Design Rationale The key to perfect fairness is to use a queue. This queue allows each CPU waiting on the lock to spin on a separate memory location, eliminating useless consumption of memory bandwidth at high levels of contention.

6.3 Queued Reader/Writer Locks

Problem What locking primitives should you use if your design can encounter high contention when fairness is an issue and if there is a high read-to-write ratio? In this context, "high" contention means that there is often more than one CPU or process waiting to acquire the lock. "High" read-to-write ratio in this context means "much greater than one".

Context A parallel program where both contention and read-to-write ratio are high and where fair access to critical sections is important. This program must run on a machine that has atomic instructions capable of implementing a queued lock (either `compare_and_swap` or `load_linked` and `store_conditional` will suffice).

Forces

- Fairness (+ + +): The queue guarantees that CPUs will be admitted to the critical section in strict FIFO order.
- Read-to-Write Ratio (++): Queued reader/writer locks allow readers to proceed in parallel, but force some updating of shared variables even when all processes or CPUs are reading. Updating these shared variables results in overhead, which limits the queued reader/writer lock's ability to allow readers to proceed in parallel.
- Memory Bandwidth (+): Use of a queue allows each CPU waiting on the lock to spin on a separate location of memory. This separate spinning avoids the bad behavior that test-and-set primitives are prone to at high contention.
- Memory Size (0): Queued reader/writer locks carry even more state than simple queued locks and thus require more memory. The memory required is still modest, with only a few bytes required for each lock and for each queue element.
- Memory Latency (−−): Queued reader/writer locks must access several disjoint memory locations that are shared by multiple processes. These additional memory references cause queued locks to be even slower than simple queued locks.

Solution Use a queued-reader/writer-lock primitive such Algorithm 4 presented by Mellor-Crummey and Scott [MCS91b]. This algorithm is rather long, so it is not shown here.

Resulting Context A program with locking primitives that enforce fairness and limit their memory-bandwidth load, while allowing reads to proceed in parallel. However, this program will suffer reduced performance at low loads compared to an equivalent program that uses test-and-set locks.

Design Rationale The key to perfect fairness is to use a queue. This queue allows each CPU waiting on the lock to spin on a separate memory location, greatly reducing useless consumption of memory bandwidth at high levels of contention.

6.4 Counter Reader/Writer Lock

Problem What locking primitive do you use in a coarse-grained parallel program with a high read-to-write ratio? In this context, a "high" read-to-write ratio means "much greater than one".

Context A parallel program with a high read-to-write ratio, high contention, and coarse-grained parallelism.

Forces

- Fairness $(+++)$: The counters guarantee that CPUs will be admitted to the critical section in strict FIFO order.
- Memory Size $(++)$: Counter reader/writer locks carry more state than do simple test-and-set locks, and thus require more memory. However, the memory usage for each counter reader/writer lock is typically much less than twenty bytes.
- Read-to-Write Ratio $(++)$: Counter reader/writer locks allow readers to proceed in parallel, but force some updating of shared variables even when all processes or CPUs are reading. Updating these shared variables results in overhead, which limits the counter reader/writer lock's ability to allow readers to proceed in parallel.
- Memory Latency (0): Counter reader/writer locks must update several counters, which results in greater sensitivity to memory latency than that of the simpler test-and-set lock.
- Memory Bandwidth $(---)$: If a large number of CPUs try to acquire the lock at the same time, each of the losing CPUs will have issued a memory transaction whose only effect is to inform the CPU that it lost. This useless memory traffic can cause severe performance problems if there are a large number of CPUs.

Solution Use a counter-reader/writer-lock primitive such as that shown in Figure 5.

This implementation is based on a "ticket lock" layered on a simple test-and-set lock. The idea is that the lock maintains the cumulative number of requests and completions for readers and writers. Each requester takes a snapshot of the number of requests so far, increments the appropriate request counter, then waits for all prior conflicting requests to complete.

Note that if the machine has `compare_and_swap()` or `load_linked()` and `store_conditional()`, the underlying test-and-set lock can be eliminated and all of the primitives can be based on atomic instructions.

This implementation is also batched fair, in other words, readers that request access after a given writer will not be allowed to proceed in parallel with readers that requested access before that reader.

```
typedef struct {
        lock_t srw_lock;
        int srw_rdrq, srw_wrrq;
        int srw_rdcp, srw_wrcp;
} srwlock_t;

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

        S_LOCK(&(l->srw_lock));
        (l->srw_rdrq)++;
        wrrq = l->srw_wrrq;
        S_UNLOCK(&(l->srw_lock));
        while (l->srw_wrcp != wrrq) {
                continue;
        }
}

void
S_RDUNLOCK(srwlock_t *l)
{
        S_LOCK(&(l->srw_lock));
        (l->srw_rdcp)++;
        S_UNLOCK(&(l->srw_lock));
}

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

        S_LOCK(&(l->srw_lock));
        rdrq = l->srw_rdrq;
        wrrq = (l->srw_wrrq)++;
        S_UNLOCK(&(l->srw_lock));
        while ((l->srw_rdcp != rdrq) ||
               (l->srw_wrcp != wrrq)) {
                continue;
        }
}

void
S_WRUNLOCK(srwlock_t *l)
{
        S_LOCK(&(l->srw_lock));
        (l->srw_wrcp)++;
        S_UNLOCK(&(l->srw_lock));
}
```

Figure 5: Counter R/W Lock Implementation

Batched-fair implementations avoid starvation of writers by readers and vice versa. Reader-preference and writer-preference implementations are also available. Reader-preference implementations are sometimes useful because they allow more readers to proceed in parallel.

Mellor-Crummey and Scott [MCS91b] present a good overview of alternative implementations of counter-reader/writer locks.

Resulting Context A program with locking primitives that allow readers to proceed in parallel. This program will work well if it has coarse-grained parallelism and a high read-to-write ratio.

Design Rationale Explicitly tracking the number of readers and writers provides a simple reader-writer locking mechanism. The heavy use of shared variables results in memory-latency overhead. This overhead sets a lower bound on the granularity of parallelism for which the counter-reader/writer lock is effective.

6.5 Distributed Reader/Writer Lock

Problem What locking primitive do you use in a fine-grained parallel program with a high read-to-write ratio and high contention? In this context, a "high" read-to-write ratio means "much greater than the number of CPUs".

Context A parallel program with a very high read-to-write ratio, high contention, and fine-grained parallelism.

Forces

- Read-to-Write Ratio (+ + +): Distributed reader/writer locks allow readers to proceed in parallel without updating shared variables. There is therefore almost no limit to this lock's ability to allow processes to proceed in parallel.
- Memory Latency (++?): Distributed reader/writer locks do not update shared variables for read-side locking. If almost all locks are read-acquired, each CPU or process will retain its state in its own cache, and there will be no memory latency overhead.

  On the other hand, write-side acquisition imposes a severe memory latency overhead.
- Memory Bandwidth (++?): Again, the read-side primitives do not update shared variables, and thus make no demands on memory bandwidth. And, again, write-side acquisition can put a very heavy memory-bandwidth load on the system.
- Fairness (+): Typical implementations approximate batch fairness (where a batch of readers is allowed to proceed in parallel between each pair of writers). However, the distributed nature of the lock makes it impossible to strictly enforce fairness.
- Memory Size (− − −): Distributed reader/writer locks require O(N) memory for each and every lock, where N is the number of CPUs or processes in the system.

Solution Use a distributed-reader/writer-lock primitive such as that shown in Figure 6 [Tay87, And91].

This implementation uses a per-CPU lock for readers and an additional lock to gate writers. A reader acquires only its CPU's lock, while a writer must acquire the writer-gate lock as well as each of the reader-side per-CPU locks. This results in extremely efficient reader-acquisition at the expense of very slow writer-acquisition, particularly for large numbers of CPUs.[4] Therefore, use distributed reader/writer locks only when the read-to-write ratio greatly exceeds the number of CPUs.

Since the distributed reader/writer lock is itself based on locking primitives, it is reasonable to ask how these underlying primitives should be implemented. There is no reason to use anything other than test-and-set locks for the per-CPU `srw_rlocks`, since at most two CPUs will contend any one of these locks at a given time. However, high write-side contention might force use of a queued lock for `srw_wlock`.

Resulting Context A program with locking primitives that allow readers to proceed in parallel. This program will work well even in face of fine-grained reader-side parallelism, but requires a very high read-to-write ratio.

Design Rationale The key to high speedups is to avoid unnecessary interactions between processes or CPUs. The distributed reader/writer locking primitives avoid all interactions between readers, thereby allowing very high speedups. However, there is a price to be paid on the writer side. Distributed reader/writer

[4] A implementation designed to run on a particular machine would pad the locks out to the size of that machine's cachelines in order to avoid false sharing.

```
typedef struct {
        lock_t srw_wlock;
        lock_t srw_rlock[NCPUS];
} srwlock_t;

void
S_RDLOCK(srwlock_t *l)
{
        S_LOCK(&(l->srw_rlock[ME]));
}

void
S_RDUNLOCK(srwlock_t *l)
{
        S_UNLOCK(&(l->srw_rlock[ME]));
}

void
S_WRLOCK(srwlock_t *l)
{
        int i;

        S_LOCK(&(l->srw_wlock));
        for (i = 0; i < NCPUS; i++) {
                S_LOCK(&(l->srw_rlock[i]));
        }
}

void
S_WRUNLOCK(srwlock_t *l)
{
        int i;

        for (i = 0; i < NCPUS; i++) {
                S_UNLOCK(&(l->srw_rlock[i]));
        }
        S_UNLOCK(&(l->srw_wlock));
}
```

Figure 6: Distributed Reader/Writer Lock Implementation

locks can be very efficient, but only if the read-to-write ratio is very high.

7 Acknowledgments

I owe thanks to Ward Cunningham, Steve Peterson, and Douglas Schmidt for encouraging me to set these ideas down and for many valuable conversations, and to Dale Goebel for his consistent support.

References

[And90] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[And91] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys:*, 1991.

[HJ91] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, pages 18–28, September 1991.

[LA93] Beng-Hong Lim and Anant Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *Transactions on Computer Systems*, 11(3):253–294, August 1993.

[McK95a] Paul E. McKenney. Differential profiling. In *MASCOTS'95*, Toronto, Canada, January 1995.

[McK95b] Paul E. McKenney. Selecting locking designs for parallel programs. In *Pattern Languages of Program Design*, September 1995.

[MCS91a] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ASPLOS*, January 1991.

[MCS91b] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MS93] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings*, Berkeley CA, February 1993.

[SC91] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *IEEE Computer*, pages 30–38, September 1991.

[Sch95] Douglas C. Schmidt. Active object. In *Pattern Languages of Program Design*, September 1995.

[SR84] Z. Segall and L. Rudolf. Dynamic decentralized cache schemes for MIMD parallel processors. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[Tay87] Y. C. Tay. *Locking Performance in Centralized Databases.* Academic Press, 1987.

[WKS94] Robert W. Wisniewski, Leonidas Kontothanassis, and Michael L. Scott. Scalable spin locks for multiprogrammed systems. In *8th IEEE Int'l. Parallel Processing Symposium*, Cancun, Mexico, April 1994.