Selecting Locking Primitives for Parallel Programs

Paul E. McKenney (pmckenne@us.ibm.com)
Sequent Computer Systems, Inc.

Abstract

The only reason to parallelize a program is to gain performance. However, the synchronization primitives used by parallel programs can consume execessive 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 [McK96].

1 Overview

A lock-based parallel program uses synchronization primitives to define critical sections of code in which only one CPU or thread 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.

A poor choice of locking primitive can result in excessive overhead and poor performance under heavy load. The pattern language in this paper will help you determine what kind of locking primitive to use. This paper considers a few straightforward test-and-set, queued, and reader/writer locks, which will handle most situations.

This paper presents the implementation level counterpart to a locking design pattern language [McK96]. Section 2 therefore gives an overview of locking design patterns. 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 almost always deeply intertwined. Therefore, this section presents a brief overview of design-level patterns and the forces that act on them.

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 thread "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 threads' data.[1]

[1] The Active Object pattern [Sch96] describes an object-

```
/* 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

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 requires writers to exclude readers and other writers.

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-thread caches of memory (or of whatever resource is being allocated)[MS93]. Allocating from 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 usually 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.

oriented approach to this sort of communications mechanism. More complex operations that atomically update data owned by many CPUs must use a more complex approach such as two-phase commit [Tay87].

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 rarely updated may often be protected with lower-overhead synchronization primitives than may a data structure with a that is frequently updated.

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 hardware and design forces imposed from above.

3.1 Forces from the Hardware

If your only goal is squeezing the maximum conceivable performance out of the hardware, then you are faced with size, bandwidth, 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 all but the following forces:

Memory Latency: The memory-latency force is analogous to the synchronization-overhead force 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 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 lock data structures are usually quite small, aggressively-parallel programs 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.

Balancing these three forces from the machine will almost always result in a good implementation. In some

[2] One possible to this trend is wide-area-network bandwidth, particularly in areas of the world undergoing telecommunications deregulation. However, the small size of the data required to represent a lock implies that latency, not bandwidth, is the limiting factor for geographically-distributed programs.

Telecommunications lines whose latencies are within a factor of three of speed-of-light limits are already commercially available. So, in absence of some unexpected breakthroughs in the field of physics, CPU speed increase is expected to dominate performance of parallel programs.

cases, other forces involving cache structure must be considered. These forces are sometimes critical to the implementation of large data structures, but seldom affect lock-primitive selection. Cache forces are therefore not discussed further in this paper.

3.2 Forces from the Design

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

Granularity of Parallelism: As noted earlier, granularity of parallelism is best defined as the ratio of the critical-section overhead to that of the locking primitives. The lower the overhead of the locking primitive, the larger the granularity of parallelism. Larger granularities of parallelism result in faster and better-scaling programs.[3]

However, design-level choices affect the critical-section overhead and usually have the larger influence on granularity.

Therefore, granularity of parallelism can be both a force and a context at the implementation level.

Fairness: If your design requires some degree of fair access to critical sections (as would a real-time system), you should use fair primitives. Otherwise, scheduling deadlines may be missed when threads fail to gain timely access to critical sections.

Note that although the design-level 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 repair a poor design.

The design-level contention force becomes an element of the implementation-level context, as does the design-level read-to-write ratio force. Both low contention and high read-to-write ratio must be designed into a parallel program.

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

[3]But if the granularity of parallelism is too coarse, the program will not scale. In the limiting case, the entire program becomes single-threaded so that only one CPU may execute it at a time.

The design-level complexity force is a special case. It is true that locking primitives can be extremely difficult to implement. However, this difficulty is 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 contention, granularity of parallelism, and read-to-write ratio:

Contention: The two most useful measures of a particular critical section's contention are: (1) the fraction wall-clock time during which one CPU is executing in that critical section, and (2) the ratio of time spent waiting to enter the critical section to that spent actually executing in the critical section. Use the first measure at the design level to predict the load level at which contention will limit speedup [McK95]. Use the second measure to help select locking primitives.

Contention is an important part of the context because the locking primitives that are least affected by high memory latencies at low levels of contention impose the heaviest memory-bandwidth load at high levels of contention. You should normally consider a lock to be highly contended if the ratio of wait time to critical-section execution time is more than 10-20%.

Granularity of Parallelism: Granularity of parallelism can be a force due to the different overheads of the locking primitives. However, since the sizes of the critical sections themselves are fixed at design time, granularity of parallelism is also part of the context.

Granularity of parallelism is an important part of the context because fine-grained parallel designs are very sensitive to the overhead of the locking primitives. You should normally consider a critical section to be coarse-grained if the lock overhead is less than 10-20% of the average critical-section overhead.

Read-to-Write Ratio: Read-to-write ratio is defined to be the ratio of the number of read-only critical sections to the number of critical sections that modify data. It is not unusual to encounter extremely high read-to-write ratios, e.g., in the routing tables used in networking protocols.

You can protect an infrequently-changed data structure with lower-overhead locking primitives than can be used for a data structure that is frequently modified.

As a rule of thumb, a low read-to-write ratio is unity or less, a moderate read-to-write ratio ranges from unity to the number of CPUs, and a high read-to-write ratio is greater than the number of CPUs. A moderate read-to-write ratio is required to get any benefit from reader/writer locking primitives, while a high read-to-write ratio is required to get benefit from primitives whose write-side cost that increases linearly with the number of CPUs.

Read-to-write ratio is a force at the design level. However, once the design is chosen, the read-to-write ratio is fixed, and therefore becomes a context at the implementation level.

These three elements of the context are usually the most important. Elements of context that may be important in more specialized situations include preemption [WKS94], thread failure [Her93], and the ratio of memory latency to instruction-execution time.

Although this paper presents locking primitives that operate reasonably well under high contention, you should redesign to avoid high contention if you wish to achieve large speedups. Use the critical-section-partitioning, data-ownership, parallel-fastpath, reader/writer-locking, or allocator-caches design patterns for such redesigns.

Unfortunately, Amdahl's law states that some part of the resulting program will be inherently serial. This portion will limit speedup, and will result in high contention if there are too many CPUs.[4]

Nonetheless, since real systems have a limited number of CPUs, your design might never experience high contention. In this case, the locking primitives are never under much stress, so you have wide latitude when balancing forces.

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)

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

The two types of queued lock tolerate extreme contention without imposing excessive memory-bandwidth loads on the system bus.

The three types of reader/writer lock allow readers to proceed in parallel. The distributed reader/writer lock operates efficiently in face of high read-side contention and fine-grained parallelism.

5.2 Force Resolution

The context controls which forces are most important, as shown in Table 1. The "Innovation" entries denote problematic contexts where good performance requires either ad-hoc locking primitives or a redesign to achieve good performance.

Table 2 shows how each of the patterns resolves each of the forces. Plus signs indicate that a pattern resolves a force well, the more plus signs, the better. For example, test-and-set lock resolves memory size perfectly because test-and-set locks can be as small as one bit in size. This earns test-and-set lock a "+++", the best possible rating.

Minus signs indicate that a pattern resolves a force poorly, the more minus signs, the worse. Again, test-and-set lock provides extreme examples with memory latency because these locks can thrash under high contention, and fairness because these locks provide no fairness mechanism. Test-and-set lock earns a "---" for both of these forces, the worst possible rating.

A zero indicates neither good nor bad resolution.

Slashes indicate that resolution depends strongly on whether the primitive is being used by a reader or a writer. The reader's resolution comes before the slash and the writer's resolution comes after it.

The rating for memory latency assumes low contention, while the rating for memory bandwidth assumes high contention.

[4] "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.

Contention	Granularity	R/W	Forces
Low			Mem. Latency
High	Coarse		Fairness, Mem. B/W
	Fine		Innovation
Low on Write Side, High on Read Side	Coarse on Read Side, Don't Care on Write Side	Moderate to High	Mem. Latency
	Fine on Read Side, Don't Care on Write Side	Moderate	Innovation
		High	Mem. Latency, Mem. B/W

Table 1: Force Prioritization

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

Table 2: Force Resolution

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

5.3 Fault Table

Use Table 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?

Context A parallel program where contention is low, where fairness and performance are not crucial, or where memory size is a limiting factor.

Forces Test-and-set locks resolve Memory Size very well (+ + +), Memory Latency moderately so (+), but they resolve Memory Bandwidth and Fairness very poorly (− − −).

Solution Use a locking primitive based on atomic test-and-set machine instructions as shown in Figure 2. 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.

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

```
#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 2: Test-and-Set Lock Implementation

Old Pattern	Problem	Pattern to use
Test-and-Set Lock (6.1)	Locking primitives consume 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 moderate to 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 3: Lock-Primitive Fault Table

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 under heavy load.

Design Rationale The simple implementation pays off with low memory and CPU overhead when contention is low. If your design enforces low contention (perhaps via the data locking, data ownership, parallel fastpath, hierarchical locking, and allocator caches design patterns), then simple test-and-set locks can be the primitive of choice.

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.

Force Resolution Rating Rationale

- 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 recently modified by some other CPU. Therefore, this single reference is likely to miss the current CPU's cache, and therefore have high overhead on systems with many CPUs.
- 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 spinning on this lock.
- Fairness (− − −): A simple test-and-set lock has no provision for fairness. It is quite possible that an extremely unlucky CPU or thread might perpetually fail to acquire the lock.

6.2 Queued Locks

Problem What locking primitives should you use?

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).[5]

Forces Queued locks resolve Fairness very well (+ + +), Memory Bandwidth and Memory Size reasonably well (+), and Memory Latency rather poorly (−).

Solution Use a queued-lock primitive such as the MCS lock shown in Figure 3 [MCS91a]. The idea behind the queued lock is that each spinning CPU has its own queue element to spin on, so that only the CPU that has just been granted the lock will incur cache misses to access the new lock state. This is in sharp contrast to the test-and-set lock's behavior, where every spinning CPU incurs cache misses in order to examine the new lock state any time any CPU releases the lock.

The `S_LOCK()` primitive enqueues the CPU's element onto the lock queue, and spins waiting for the lock to be granted. The `S_UNLOCK()` primitive grants the lock to the next CPU on the queue. The queue enforces strict FIFO ordering, guaranteeing fair access to the lock.

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

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. Wisniewski et al. [WKS94] proposed a lock that allows limited out-of-order service to prevent active threads from waiting behind blocked threads.

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 when compared to an equivalent program that uses test-and-set locks.

[5] 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.

The `load_linked` primitive fetches the value at the specified address, tracking whether any other CPU modifies this value before the next `store_conditional`. If there has been no such modification, the `store_conditional` simply stores the new value at the specified address. Otherwise, the `store_conditional` fails.

```
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;

        /* Add self to queue. */

        i->next = NULL;
        pred = fetch_and_store(*l, i);
        if (pred != NULL) {

                /*
                 * Someone precedes us, wait
                 * for them to finish.
                 */

                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)) {

                        /* We are last. */

                        return;
                }
                while (i->next == NULL) {
                        continue;
                }
        }

        /*
         * Someone follows us, grant the
         * lock to them.
         */

        i->next->locked = FALSE;
}
```

Figure 3: Queued-Lock Implementation

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.

Force Resolution Rating Rationale

- 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 threads. These additional memory references cause queued locks to be slower than simple test-and-set locks.

6.3 Queued Reader/Writer Locks

Problem What locking primitives should you use?

Context A parallel program where contention is high, read-to-write ratio is moderate or 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 Queued reader/writer locks resolve Fairness very well (+++), Memory Bandwidth reasonably well (+), Memory Size neither well nor poorly (0), and Memory Latency rather poorly (−−).

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.

The basic idea is that when a reader is granted the lock, it first checks to see if the next element on the queue also corresponds to a reader. If so, it immediately grants the lock to this next reader.

When the last reader leaves the critical section, the lock is granted to the writer at the head of the queue.

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.

Force Resolution Rating Rationale

- 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 (0): Queued reader/writer locks carry even more state than simple queued locks and thus require more memory.
- Memory Latency (−−): Queued reader/writer locks must access several disjoint memory locations that are shared by multiple threads. These additional memory references cause queued locks to be even slower than simple queued locks.

6.4 Counter Reader/Writer Lock

Problem What locking primitive should you use?

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

Forces Counter reader/writer locks resolve Fairness very well (+ + +), Memory Size and Read-to-Write Ratio reasonably well (++), Memory Latency neither well nor poorly (0), and Memory Bandwidth very poorly (− − −).

Solution Use a counter-reader/writer-lock primitive such as that shown in Figures 4, 5, and 6, adapted from Algorithm 3 of [MCS91b].

```
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;
```

Figure 4: Counter R/W Lock Data

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 hardware implements either `compare_and_swap()` or both `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. 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.

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

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

        S_LOCK(&(l->srw_lock));
        (l->srw_rdrq)++;
        wrrq = l->srw_wrrq;
        S_UNLOCK(&(l->srw_lock));

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

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

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

        S_LOCK(&(l->srw_lock));
        (l->srw_rdcp)++;
        S_UNLOCK(&(l->srw_lock));
}
```

Figure 5: Counter R/W Lock Read Side

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

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

        S_LOCK(&(l->srw_lock));
        rdrq = l->srw_rdrq;
        wrrq = (l->srw_wrrq)++;
        S_UNLOCK(&(l->srw_lock));

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

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

        S_LOCK(&(l->srw_lock));
        (l->srw_wrcp)++;
        S_UNLOCK(&(l->srw_lock));
}
```

Figure 6: Counter R/W Lock Write Side

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.

Force Resolution Rating Rationale

- 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.
- 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.

6.5 Distributed Reader/Writer Lock

Problem What locking primitive should you use?

Context A parallel program with a high read-to-write ratio and high read-side contention.

Forces
Distributed reader/writer locks resolve Granularity of Parallelism very well on the read side, but very poorly on the write side (+++/−−−), Memory Latency very well on the read side but very poorly on the write side (+ + +/ − −−), Memory Bandwidth very well on the read side but neither well nor poorly on the write side (+ + +/0), Fairness reasonably well (+), and Memory Size very poorly (− − −).

Solution Use a distributed-reader/writer-lock primitive such as that shown in Figure 7 [Tay87, And91].

This implementation uses a per-CPU lock for readers and an additional lock to gate writers. A reader

```
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 7: Distributed Reader/Writer Lock Implementation

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.[6] This results in extremely efficient reader-acquisition at the expense of very slow writer-acquisition, particularly for large numbers of CPUs. Therefore, use distributed reader/writer locks only when the read-to-write ratio is high.

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

Force Resolution Rating Rationale

- Memory Bandwidth (+ + +/0): The read-side primitives do not update shared variables, and thus make no demands on memory bandwidth. Similarly, write-side acquisition can put a very heavy memory-bandwidth load on the system.

- 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 thread 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.

- Granularity of Parallelism (+ + +/ − −−): The low memory latency enjoyed by the read-side locking operations results in very low read-side overhead, which increases the granularity of parallelism.

  Similarly, the high memory latency experienced by the write-side operations reduces the granularity of parallelism.

- 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 threads in the system.

[6] 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.

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.

[Her93] Maurice Herlihy. Implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[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.

[McK95] Paul E. McKenney. Differential profiling. In *MASCOTS'95*, Toronto, Canada, January 1995.

[McK96] Paul E. McKenney. Selecting locking designs for parallel programs. In *Pattern Languages of Program Design*, volume 2, pages 501–531, June 1996.

[MCS91a] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions of Computer Systems*, 9(1):21–65, February 1991.

[MCS91b] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third PPOPP*, pages 106–113, Williamsburg, VA, April 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.

[Sch96] Douglas C. Schmidt. Active object. In *Pattern Languages of Program Design*, volume 2, pages 483–499, June 1996.

[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.