acmqueue

Nonblocking Algorithms and Scalable Multicore Programming

Exploring some alternatives to lock-based synchronization

Samy Al Bahra, AppNexus

Real-world systems with complicated quality-of-service guarantees may require a delicate balance between throughput and latency to meet operating requirements in a cost-efficient manner. The increasing availability and decreasing cost of commodity multicore and many-core systems make concurrency and parallelism increasingly necessary for meeting demanding performance requirements. Unfortunately, the design and implementation of correct, efficient, and scalable concurrent software is often a daunting task.

Concurrent systems on commodity multicore processors typically rely on lock-based synchronization to guarantee the coherency of shared mutable state. Lock-based synchronization has several limitations, however, including sensitivity to preemption and the possibility of deadlock, making it inappropriate or impractical for certain environments. This does not imply that lock-based synchronization is a barrier to the design and implementation of efficient and scalable software.

Nonblocking synchronization may be used in building predictable and resilient systems while avoiding the problems of lock-based synchronization. This class of synchronization is not a silver bullet, especially outside of the abstract models in which its performance properties were originally defined. Several of the performance bottlenecks and error conditions associated with lock-based synchronization remain (albeit in more obfuscated forms); therefore, ensuring correctness requires more complex verification methods, and in some cases nonblocking algorithms require the adoption of complex support systems. These complexities frequently make nonblocking synchronization a victim of hype, fear, uncertainty, and doubt. This article aims to equip the reader with the knowledge needed to identify situations that benefit from nonblocking synchronization.

COMMON PRINCIPLES

Before elaborating on typical motivations for adopting nonblocking data structures, this section highlights some important principles for understanding scalability on multiprocessor systems in the traditional threading model. Both lock-based and nonblocking synchronization have performance characteristics that are a function of these principles. Practitioners who are already intimately familiar with cache-coherent multiprocessors, out-of-order execution, and the design and implementation of lock-based synchronization objects may wish to skip this section. While this section is by no means exhaustive, additional references have been provided. Paul E. McKenney outlines a methodology for choosing appropriate lock-based synchronization mechanisms,[19] and other common principles have been described in previous articles.[6,15,21]

CONTENTION INHIBITS SCALABILITY

Contention on shared objects in parallel applications is a primary impediment to scalability and predictability. Regardless of the higher-level synchronization facilities being used, contention over a shared object involves some form of serialization by the underlying runtime environment, whether it is a language runtime or a shared-memory multiprocessor.

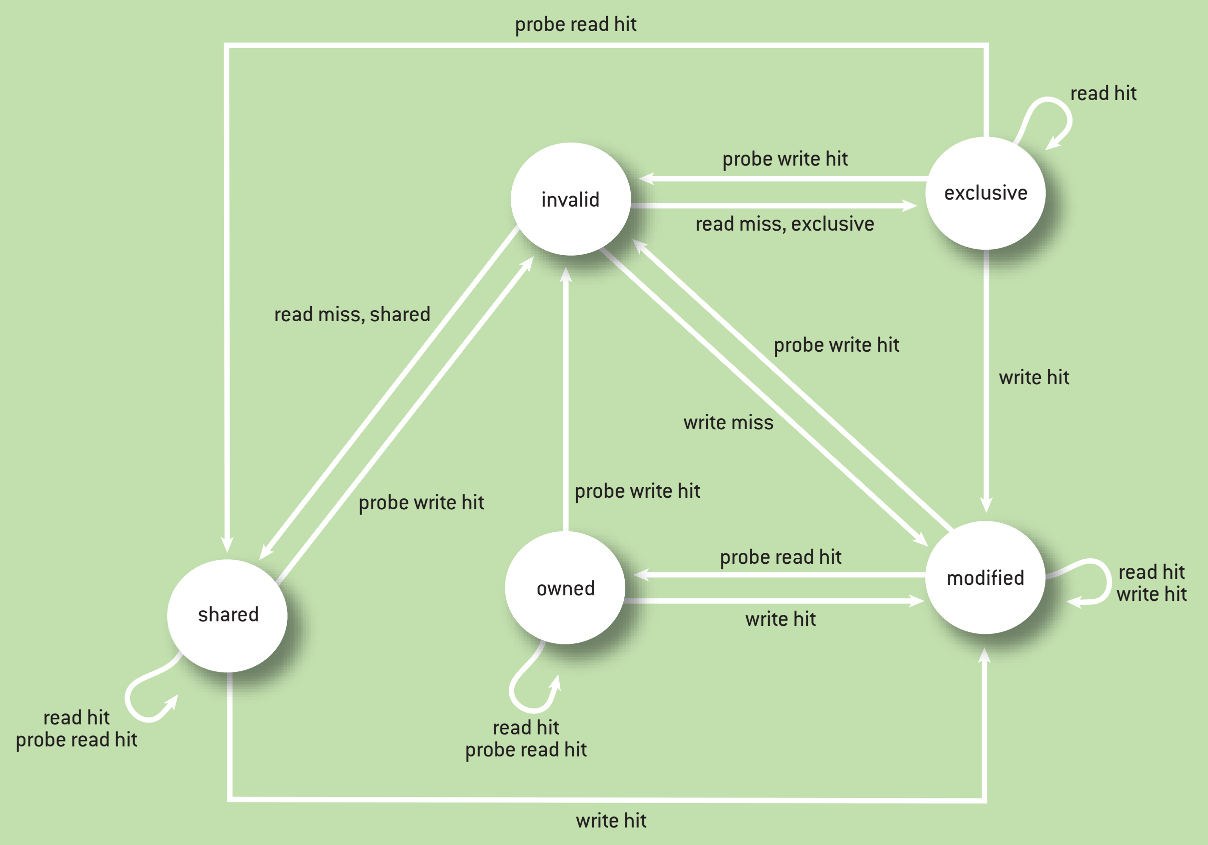
Little’s law (L = λW) is a fundamental result of queuing theory and it can be used to understand better the effects of contention on serialized resources. The law states that that average number of outstanding requests (L) in a system is a product of the effective arrival rate of requests (λ) and average request completion time (W). Contention, a product of the effective arrival rate of requests to a shared resource, leads to a queuing effect that directly impacts the responsiveness of a system.

Regardless of which synchronization scheme is used, contention over shared data inhibits the scalability and predictability (in terms of latency and sometimes throughput) of a parallel application. The program designer is responsible for minimizing contention at a systemic level in order to achieve higher performance. The chosen synchronization paradigm, whether it is lock-based or nonblocking, exhibits reduced performance in the presence of contention.

SHARED MUTABLE STATE AND CONTENTION

Understanding the mechanisms that provide coherency guarantees on multiprocessors is a

FIGURE 1 **The MOESI State Machine**



prerequisite to understanding contention on such systems. Cache-coherency protocols are the prominent mechanisms for guaranteeing the eventual consistency of shared state across multiple cache-coherent processors. These protocols implement coherency guarantees at the cache-line level, the unit that caches write to and read from main memory (at least from the point of view of the coherency mechanism).

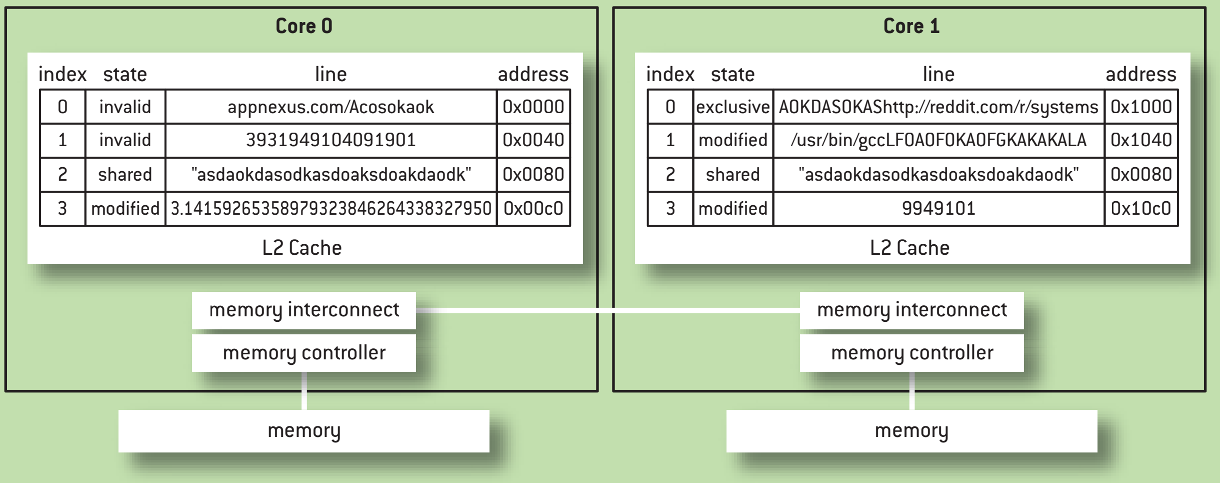
The three prominent cache-coherency protocols on commodity processors—MOESI, MESI, and MESIF—derive their acronyms from the states they define for cache lines: Modified, Owned, Exclusive, Shared, Invalid, and Forward. The cache-coherency protocol manages these states with the help of a memory-coherence mechanism that guarantees the eventual consistency of shared memory. Figure 1 illustrates the state machine associated with the MOESI protocol.[1] You may interpret the probe transitions as those triggered by external memory accesses originating from other cores.

The following example illustrates the life cycle of shared reads and writes in a cache-coherent multiprocessor. The state machine associated with this life cycle has been simplified for brevity. This program spawns a thread that reads a modified variable:

```
volatile int x = 443;

static void *
thread(void *unused)
{

	fprintf(stderr, “Read: %d\n”, x);
	return NULL;
}

int
main(void)
{
	pthread_t a;

	x = x + 10010;
	if (pthread_create(&a, NULL, thread, NULL) != 0) {
		exit(EXIT_FAILURE);
	}

	pthread_join(a, NULL);
	return 0;
}
```

The example assumes that the coherency mechanism being used is bus snooping, which allows any processor to monitor any memory accesses to shared locations. The initial state of the system is presented below. There are two sockets, each with one core and a 256-byte L2 direct-mapped write-back cache with 64-byte cache lines. (See figure 2).

FIGURE 2 **Initial State of System**



Core 0

index	state	line	address
0	invalid	appnexus.com/Acosokaok	0x0000
1	invalid	3931949104091901	0x0040
2	shared	"asdaokdasodkasdoaksdoakdaodk"	0x0080
3	modified	3.14159265358979323846264338327950	0x00c0

Core 1

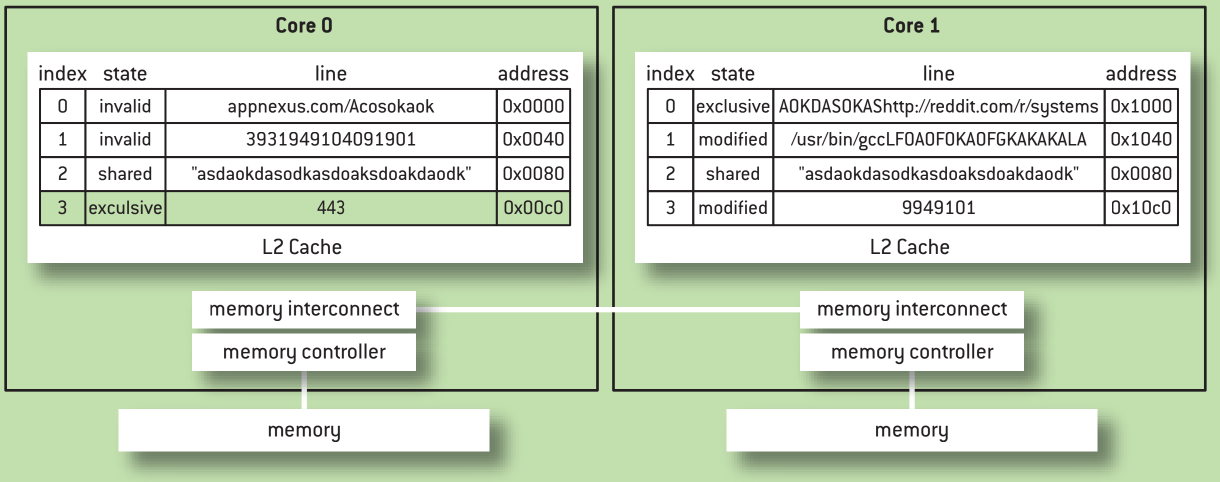
index	state	line	address
0	exclusive	AOKDASOKAShttp://reddit.com/r/systems	0x1000
1	modified	/usr/bin/gccLFOAOFOKAOFGKAKAKALA	0x1040
2	shared	"asdaokdasodkasdoaksdoakdaodk"	0x0080
3	modified	9949101	0x10c0

The process initially executes on core 0. Assume that the address of variable `x` is `0x20c4`. The increment of the value 10010 to `x` (`x = x + 10010;`) may decompose into three operations:

- Load `x` from memory into a CPU register.
- Increment the register by 10010.
- Store the value of the register into the memory location of `x`.

The address of `x` is `0x20c4` and is hashed to cache line 3. Since this cache line is in a modified state and contains data from a different address (`0x00c0`), it must be written back to main memory. No other socket contains the cache line for `0x20c4`, so a 64-byte block (starting from `0x20c0`) is read into cache line 3 from main memory and set to the exclusive state (see figure 3). The store into `x`

FIGURE 3 **After Initial Load of Variable x**



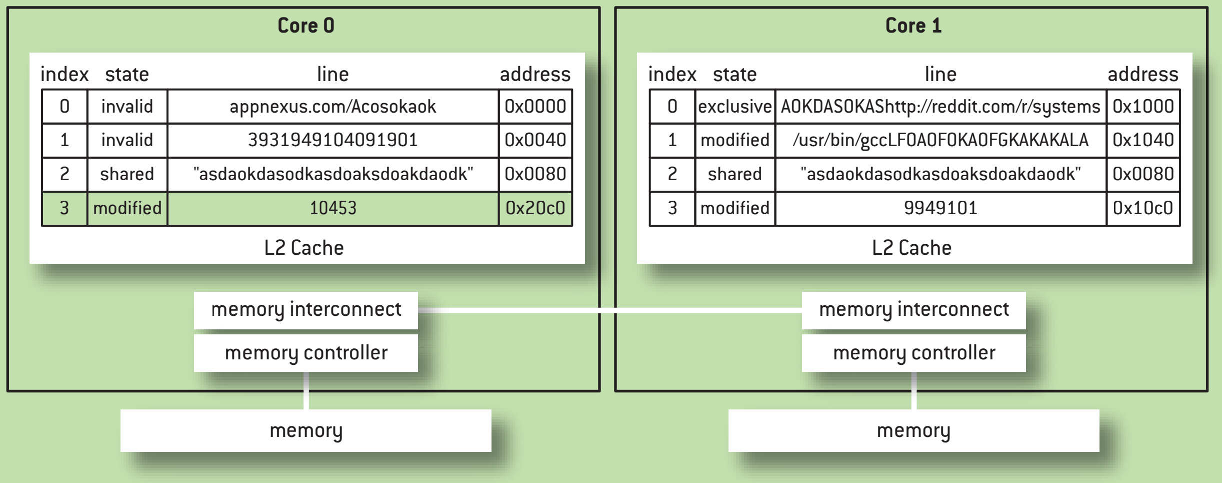
Core 0

index	state	line	address
0	invalid	appnexus.com/Acosokaok	0x0000
1	invalid	3931949104091901	0x0040
2	shared	"asdaokdasodkasdoaksdoakdaodk"	0x0080
3	exculsive	443	0x00c0

Core 1

index	state	line	address
0	exclusive	AOKDASOKAShttp://reddit.com/r/systems	0x1000
1	modified	/usr/bin/gccLFOAOFOKAOFGKAKAKALA	0x1040
2	shared	"asdaokdasodkasdoaksdoakdaodk"	0x0080
3	modified	9949101	0x10c0

FIGURE 4 **After Store to x**



Core 0

index	state	line	address
0	invalid	appnexus.com/Acosokaok	0x0000
1	invalid	3931949104091901	0x0040
2	shared	"asdaokdasodkasdoaksdoakdaodk"	0x0080
3	modified	10453	0x20c0

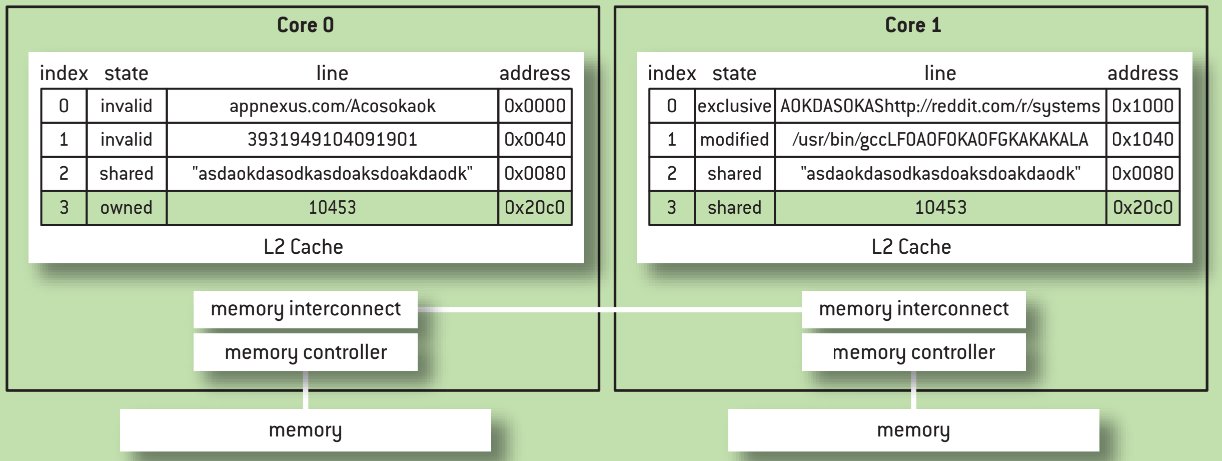
Core 1

index	state	line	address
0	exclusive	AOKDASOKAShttp://reddit.com/r/systems	0x1000
1	modified	/usr/bin/gccLFOAOFOKAOFGKAKAKALA	0x1040
2	shared	"asdaokdasodkasdoaksdoakdaodk"	0x0080
3	modified	9949101	0x10c0

transitions the cache line from exclusive to modified state as shown in figure 4.

The new thread spawns and eventually begins executing the `thread` function. Assume this execution occurs on core 1. The `thread` function executes a load from the address of `x` as an input to the `fprintf` function call. This load issues a read probe, requiring a transition from the modified state to the owned state (see figure 5). MOESI allows for cache-to-cache transfer of data, an advantage if probe latency is lower than latency to main memory. In this specific configuration, this operation involves a probe on what is typically a higher-latency memory interconnect (higher

FIGURE 5 **After Remote Read of x**



Core 0

index	state	line	address
0	invalid	appnexus.com/Acosokaok	0x0000
1	invalid	3931949104091901	0x0040
2	shared	"asdaokdasodkasdoaksdoakdaodk"	0x0080
3	owned	10453	0x20c0

Core 1

index	state	line	address
0	exclusive	AOKDASOKAShttp://reddit.com/r/systems	0x1000
1	modified	/usr/bin/gccLFOAOFOKAOFGKAKAKALA	0x1040
2	shared	"asdaokdasodkasdoaksdoakdaodk"	0x0080
3	shared	10453	0x20c0

latency to intrasocket cache access). The latency associated with the probe is also a function of the topology. On larger-scale machines with asymmetric interconnect topologies, substantial performance mismatch may exist on memory accesses; some sockets may require one hop for an invalidation cycle, while others may require two or more.

A cache-friendly program (one able to retain shared state in cache with minimal coherence traffic) scales well on modern cache-coherent multiprocessors. Mutations to actively shared state lead directly to contention as cache controllers mediate ownership of cache lines.

COHERENCY GRANULARITY

As mentioned earlier, the cache line is the granularity at which coherency is maintained on a cache-coherent multiprocessor system. This is also the unit of contention. If logically disparate mutable objects share the same cache line, then operations on these objects exhibit contention and may become a scalability bottleneck. This is called *false sharing*.

Consider the following C snippet. Each thread is provided a unique index (`UNIQUE_THREAD_ID`) into an array of counters that each thread reads from and writes to. Since these counters are laid out sequentially in memory and cache, cache-line ownership ping-pongs between all processors incrementing the counter value. This vicious cycle of invalid/shared/modified/owned cache-line transitions is an example of the kind of cache-line ping-ponging that can occur as a result of false sharing.

```
#define N_THR 8

struct counter {
        unsigned long long value;
};

static volatile struct counter counters[N_THR];

void *
thread(void *unused)
{

        while (leave == false) {
                counters[UNIQUE_THREAD_ID].value++;
        }

        return NULL;
}
```

One way of avoiding this problem is to guarantee that at most one counter is in a given cache line by padding to a cache-line size. Assuming the host processor of the application has 64-byte cache lines, the following modification avoids false sharing:

```
struct counter {
       unsigned long long value;
       char pad[64 - sizeof(unsigned long long)];
};
```

In the same vein, padding should be done only when necessary. It is still important to keep access patterns in mind. For example, if two mutexes are acquired in succession on the hot path (i.e., the most frequently executed segment of code), then cache-line padding does little but drive up lock-operation latency. Excessive padding may impact the overall memory footprint of the application, causing increased pressure on memory resources. Additional approaches exist to detect and minimize false sharing.[16,17,19] On modern architectures, atomic operations applied to cached memory locations rely on the cache-coherency protocol to ensure atomicity. Older systems relied solely on bus locking to provide such atomicity guarantees. After a cache-line invalidation step, a bus-lock signal assertion prevents any further accesses to and from main memory from all processors. This can lead to significant system-wide performance degradation. Avoid atomic operations that cross a cache-line boundary. Even if it is supported by the target architecture, it is likely implemented through some form of bus or directory locking. On the x86 architecture, for example, such atomic *split transactions* are defined as asserting a bus-lock signal.

UNAVOIDABLE COSTS OF ORDERING AND VISIBILITY GUARANTEES

Certain correctness requirements call for the use of synchronization instructions that are heavier-weight than atomic loads and stores. To improve performance, many modern processors batch memory operations and/or provide some form of instruction-level parallelism. The presence of these optimizations may require the use of memory-barrier (or serializing) instructions to enforce ordering of memory operations and their intended side effects. (The term *memory barrier* can be used interchangeably with *memory fence*.) The need for barrier instructions depends on the processor memory model, which also defines the reordering possibilities of memory operations.

Consider the following source-code snippet:

```
volatile int w, x, y, z;

w = 0;
x = 1;
z = y;
```

Under a sequential consistency model, instructions (including loads and stores) are executed in order. The store to `x` occurs after the store to `w` and before the load from `y` to `z`. Modern processors that are able to handle more than one load or store at a time generally have more relaxed constraints on the ordering of these operations in order to improve performance. Several modern architectures even implement lockup-free caches,[2] which can handle multiple cache misses in parallel while still allowing for cache accesses. For example, the load from `y` may be serviced before the store to `x`, or the store to `x` may be executed before the store to `w`. Even in the absence of instruction reordering, it may still be possible that a remote processor sees the store to `z` before the store to w on some architectures.

Here's a less innocuous example:

```
volatile int ready = 0;

void
produce(void)
{
        message = message_new();
        message->value = 5;
        message_send(message);
        ready = 1;
}

void
consume(void)
{

        while (ready == 0) {
                ; /* Wait for ready to be non-zero. */
        }

        message = message_receive();
        result = operation(&message->value);
}
```

In this pseudocode, a producer thread executing on a dedicated processor creates a message and then signals a dedicated consumer thread on another processor by updating `ready`. With no memory barriers, it is possible for the processor executing `consume` to receive or process the remote store to `ready` before the remote store to `message->value` completes and/or is made visible to other processors. To guarantee that `produce` commits the store to `message->value` before the `ready = 1` operation, a memory barrier may be necessary. Then, to guarantee that the reception of the message in `consume`, after the consumer has observed `ready` in a non-zero state, contains any memory updates before the accompanying `ready = 1` operation (in `produce`), another memory barrier may be necessary:

```
volatile int ready = 0;

void
produce(void)
{
        message = message_new();
        message->value = 5;
        message_send(message);

        /*
         * Make sure the above memory operations complete before
         * any following memory operations.
         */
        MEMORY_BARRIER();
        ready = 1;
}

void
consume(void)
{

        while (ready == 0) {
                ; /* Wait for ready to be non-zero */
        }

        /*
         * Make sure we have an up-to-date view of memory relative
         * to the update of the ready variable.
         */
        MEMORY_BARRIER();
        message = message_receive();
        result = operation(&message->value);
}
```

Some processors have specialized barrier instructions that guarantee the partial ordering of only some memory operations (examples include serializing only stores with respect to each other or only loads with respect to each other). The requirement of memory barriers depends on the underlying memory model. The most common memory-ordering guarantees on modern general-purpose processors are:

TSO (Total Store Ordering)
• Loads are not reordered with respect to each other.
• Stores are not reordered with respect to each other.
• Stores are not reordered with respect to prior loads.
• Loads can be reordered with respect to prior stores.
• Stores to the same location have a global ordering.
• Atomics are serializing.
• Examples include x86 TSO[26] and SPARC TSO.

PSO (Partial Store Ordering)

• Loads are not reordered with respect to each other.
• Stores can be reordered with respect to each other.
• Loads can be reordered with respect to stores.
• Stores to the same location have a global ordering.
• Atomics can be reordered with respect to stores.
• Examples include SPARC PSO.

RMO (Relaxed Memory Ordering)

• Loads are reordered with respect to each other.
• Loads can be reordered with respect to stores.
• Stores can be reordered with respect to each other.
• Stores to the same location have a global ordering.
• Atomics can be reordered with respect to stores and loads.
• Examples include Power[27] and ARM.[7]

Again, Paul McKenney offers additional details on these memory models.[19]
To make matters worse (in terms of complexity), some languages allow their compilers and runtime environments to reorder operations. Until recently, even C and C++ lacked a concurrent memory model (so the semantics of concurrent accesses were very much compiler- and environment-specific). Many programming languages may exclusively emit memory barriers through acquire/release semantics. Memory models and their implications will be explored in greater detail in an upcoming *ACM Queue* article.

Popular lock-based synchronization mechanisms hide the details of memory barriers from program designers by having implicit heavyweight memory barriers. This simplifies concurrent programs as developers are left to reason with serializability,[3] but it sometimes comes at the cost of reduced performance.

Memory barriers are also necessary for many other concurrent algorithms, especially classes of algorithms that do not rely on locks and their ordering guarantees. Ultimately, the ability to avoid heavyweight synchronization mechanisms depends on the correctness and visibility requirements of the data structure. Recently, the requirement of expensive synchronization instructions for certain correctness constraints in the presence of common access patterns has been formalized.[4]

ATOMIC READ-MODIFY-WRITE OPERATIONS ARE EXPENSIVE

The cost of atomic read-modify-write operations depends on the underlying architecture and actual programming language implementation. TSO is becoming an increasingly common memory model on commodity processors. Among other things, this memory model defines atomic operations as having a total ordering. This guarantee comes at a significant cost, even in the absence of contention. Table 1 illustrates noncontending throughput of atomic CAS (compare-and-swap) operations (`lock cmpxchg`) versus regular CAS operations (`cmpxchg`). The atomic CAS can provide total ordering and atomicity guarantees across multiple processors, while the regular CAS cannot. These measurements were made on an Intel Core i7-3615QM at 2.30 GHz and include the overhead of register spillage.

TABLE 1. The cost of atomic read-modify-write operations

Operation	Throughput (operations/second)
Atomic CAS	147,304,564
CAS	458,940,006

On TSO architectures, these atomic operations imply heavyweight memory-barrier semantics. Even on architectures with weaker memory models, atomic operations may be implemented in terms of complex instructions that incur the cost of long dependency chains in the pipeline. In other cases, programming languages may themselves define a total ordering on these instructions, which may lead to unnecessarily heavyweight memory fences being emitted. Atomic operations should be used only if necessary. On most architectures, atomic read-modify-write instructions are expensive (relative to other instructions)—even in the absence of contention.

Before deciding on the use of atomic operations, take into account the progress guarantees of the actual atomic implementation. For example, some architectures relying on LL/SC (load-linked/store-conditional primitives) such as ARM or Power architectures may still leave an application sensitive to external disturbances (jitter) such as preemption, even in the absence of contention.

Take advantage of fetch semantics provided by the atomic operation arsenal. If a platform provides an atomic fetch-and-increment operation or a CAS operation that returns the previous value, use it.

```
int shared_flag = 0;

void
worse(void)
{
	int old_value = 0;

	/*
	 * Under contention, this may involve a write invalidation
	 * followed by another transition to shared state (at least
	 * two probes).
	 */
	while (atomic_cas(&shared_flag, old_value, 1) == false) {
		old_value = atomic_load(&shared_flag);
	}
}

void
better(void)
{
	int old_value = 0;
```

```
        int snapshot;

        while (true) {
                /*
                 * We generate a single write cycle to retrieve the
                 * value we are comparing against. This can reduce
                 * cache coherency traffic by at least 50% compared
                 * to the previous worse() usage pattern.
                 */
                snapshot = atomic_cas_value(&shared_flag, old_value, 1);

                /* Operation has completed, exit loop. */
                if (old_value == snapshot)
                        break;

                old_value = snapshot;
        }
}
```

BE WARY OF TOPOLOGY

The latency and throughput properties of memory accesses vary according to the type of memory access. For example, accessing memory in a remote cache may be cheaper than accessing local uncached memory, and accessing remote uncached memory is more expensive than accessing local uncached memory. Table 2 displays the latency of a point-to-point wake-up on a two-socket 12-core Intel Xeon L5640 machine at 2.27 GHz. The latency is measured in CPU TSC (timestamp counter) ticks. In one scenario (local), a thread on one core signals a thread on another core on the same socket. In the second scenario (remote), a thread on one socket signals a thread on another socket. The remote scenario includes the overhead of going over the memory interconnect. This performance asymmetry is an example of a non-negligible NUMA (non-uniform memory access) factor. The higher the NUMA factor, the higher this asymmetry. Optimizing the placement of objects that are shared across cores minimizes NUMA effects. For example, if shared state is accessed by a cluster of threads on a single socket, then there is no reason for that object to be contained in remote memory.

TABLE 2. **Latency of a point-to-point wake-up**

Scenario	Latency (TSC)
Local	319
Remote	657

If an object is shared across many sockets on a system, make sure the object is in a location that minimizes hop distances to as many cores as possible according to the workload. NUMA-aware kernels such as Linux or Solaris by default have a first-touch policy when determining memory page placement. The placement of a page is determined by the position in the NUMA hierarchy of the first thread to access it. This behavior is configurable.[24,25]

Synchronization objects that are not NUMA-aware may also be susceptible to NUMA effects. These effects manifest not only as a fast-path performance mismatch between cores but also as starvation or even livelock under load.

Table 3 displays the throughput of lock-unlock operations on a single spinlock by two threads executing on the same socket (local scenario), and the same behavior on different sockets (remote scenario). The unit of measurement is a/s (acquisitions per second). The remote scenario shows a dramatic increase in starvation.

Fair locks guarantee starvation freedom at the cost of increased preemption sensitivity. Variants of these locks also allow for scalable point-to-point wake-up mechanisms. Recently, lock cohorting was developed as a generalized methodology for allowing NUMA awareness in locks at the cost of a higher fast-path latency.[10] Other NUMA-aware variants exist for popular primitives such as read-write locks. [5,18]

PREDICTABILITY MATTERS

Predictability is a desirable property in high-performance concurrent applications that require stringent latency or throughput bounds. The progress guarantees of synchronization mechanisms define behavior in the presence of contention and unforeseen execution delays (which can include jitter). If a program is designed solely for the fast path, then minor disturbances in execution and/or workload may compound to result in unpredictably low performance.

Many synchronization primitives rely on an adaptive scheme for waiting on resource availability. For example, many mutex implementations use busy-waiting (polling) for a bounded period of time and then fall back on heavier-weight polling or asynchronous signaling facilities (examples include `futex` on Linux or bounded yield and sleep on FreeBSD). These semantics allow for better system-wide response under co-programming (multiple processes on the same core), fairness under load, and an improved energy profile, among other things. That said, they might exhibit high operation-latency spikes under specific execution histories. They may also have increased constant latency overhead compared to simpler busy-wait mechanisms—even on uncontended execution. While a system's latency profile may be acceptable with low contention, the situation may quickly become volatile under even moderate levels of contention or external delays.

Systems relying on blocking synchronization can be especially sensitive to execution delays. A significant delay in one thread holding a synchronization object leads to significant delays for all other threads waiting on the same synchronization object. Examples of such execution delays include process preemption and timer interrupts. The significance of these delays depends on the latency constraints of the application. If the application is required to be in the microsecond latency range, then even a network interrupt may be significant. On general-purpose processors

TABLE 3. **Throughput of lock-unlock operations on a single spinlock**

Scenario	Thread 0	Thread 1	% Difference
Local	64,782,426 a/s	62,271,265 a/s	~4%
Remote	69,495,790 a/s	16,009,527 a/s	~334%

and operating systems, isolating all external sources of delays is difficult. If there are strict latency requirements for a program relying on blocking synchronization in the fast path, consider minimizing external disturbances (realtime scheduling policies and interrupt rerouting are common). For the time scales where every disturbance counts, specialized realtime operating systems and hardware exist to address those challenges.

NONBLOCKING SYNCHRONIZATION

Before looking at reasons for adopting nonblocking data structures, this section introduces some terminology and briefly examines the complexities associated with the practical design, implementation, and application of nonblocking data structures.

In the literature, nonblocking synchronization and nonblocking operations fall into three primary classes of algorithms, each with a unique set of progress guarantees:

- **OF (Obstruction Freedom).** The algorithm provides single-thread progress guarantees in the absence of conflicting operations.
- **LF (Lock Freedom).** The algorithm provides system-wide progress guarantees. At least one active invocation of an operation is guaranteed to complete in a finite number of steps. There is no guarantee of starvation freedom.
- **WF (Wait Freedom).** The algorithm provides per-operation progress guarantees. Every active invocation of an operation completes in a finite number of steps. In the absence of overload, there is a guarantee of starvation freedom.

There is a total ordering to these classes of algorithms such that any wait-free algorithm is also lock-free and obstruction-free; any lock-free algorithm is also obstruction-free. A nonblocking data structure implements operations that satisfy the progress guarantees in the nonblocking hierarchy (summarized earlier) for some (or an infinite) level of concurrency.[13] It is possible to implement a data structure that provides a nonblocking progress guarantee for a limited number of readers or writers. A traditional example of this is the Michael Scott's two-lock queue, which provides wait-free progress guarantees in the presence of up to one concurrent enqueue and one concurrent dequeue operation.[23]

It is also possible for a data structure to implement different progress guarantees for different sets of operations. Examples include a wait-free enqueue operation with a multiconsumer-blocking dequeue operation, as seen in the URCU (userspace read-copy-update) library (http://lttng.org/urcu), and a bounded-FIFO (first in first out) with a single-producer wait-free enqueue and multi-consumer lock-free dequeue, as seen in the Concurrency Kit library (http://concurrencykit.org/).

STATE SPACE EXPLOSION

Nonblocking data structures rely on atomic loads and stores or more complex atomic read-modify-write operations. The mechanism used depends on the level of concurrency the data structures intend to support and their correctness requirements.[4,13] In lock-based synchronization, it is usually sufficient to reason in terms of locking dependencies and critical sections (and read-side or write-side critical sections for asymmetric lock-based synchronization such as read-write locks).[3] When reasoning about the correctness of concurrent algorithms in the absence of critical sections, such as in nonblocking algorithms, the number of execution histories involving the interaction of shared variables can be enormous. This state space can quickly become infeasible for a mere human being to exhaust strictly through state-based reasoning. Consider the following program:

```
int x = 1;
int y = 2;
int z = 3;

void
function(void)
{
	/*
	 * This returns a unique integer value
	 * for every thread.
	 */
	int r = get_thread_id();

	atomic_store(&x, r);
	atomic_store(&y, r);
	atomic_store(&z, r);
}
```

Assuming that this program is executed by two threads under a memory model that implements TSO, there are approximately 20 possible execution histories if you take only the three store operations into consideration—and consider them completely uniform and deterministic. With four threads, 369,600 distinct execution histories exist. A derivation yields that for N deterministic processes with M distinct actions, there are $(NM)! / (M!)^N$ execution histories.[28] Coupled with programming language reordering possibilities, processor memory reordering possibilities, and out-of-order execution, the state space will grow even more quickly in complexity.

Verifying the correctness of nonblocking algorithms, especially those of the wait-free and lock-free class, requires a good understanding of a program's underlying memory model. The importance of understanding the memory model cannot be overstated, especially if the model itself is not precisely defined or allows for machine- or compiler-dependent behavior (as is the case for languages such as C or C++). What may seem like memory-model minutiae may actually be a violation in the correctness of your program. Let's take a close look at the following C snippet:

```
void
producer_thread(void)
{
	message_t *message;

	message = message_create();
	memset(message, 0, sizeof *message);
	send_to_consumer(message);
}
```

```
void
consumer_thread(void)
{
	message_t *message;

	message = message_receive();

	/*
	 * The consumer thread may act in an erroneous
	 * manner if any of the contents of the message
	 * object are non-zero.
	 */
	assert(message->value == 0);
}
```

Assume that send_to_consumer consists solely of loads and stores and no serializing instructions. On x86 processors that implement TSO, stores to memory locations are committed in order with a few exceptions. Assuming that the memset function consists solely of regular store instructions, then the assertion in consumer_thread should not fail. The reality is that certain instructions, operations, and memory types are not guaranteed to comply with the x86 TSO model.[26] In the example, the memset function can be implemented with high-performance non-temporal and/or vectorized store instructions that violate the usual memory-ordering semantics. It would be necessary to determine whether the compiler or standard library implementation and target platform guarantees memory-store serialization with respect to memset. Fortunately, most (but not all) popular implementations of memset that use these instructions emit a memory barrier of some kind (by silent contract), but this is not a requirement. If you are designing lockless concurrent programs in a low-level language, be prepared to explore similar dark recesses of implementation-defined behavior.

CORRECTNESS

The most common correctness guarantee for nonblocking data structures and operations is linearizability. This requires that operations appear to complete atomically at some point between their invocation and completion. The *linearization point* of an operation is the instant in which the operation appears to have been completed atomically. It helps to reason in terms of linearization points with respect to the sequential specification of the data structure. For reasons described in the previous section, proving the linearizability of an algorithm is often nontrivial.

Specializing a nonblocking data structure for a fixed level of concurrency can greatly reduce the complexity of the state space. Considering the potential size of the state space of nonblocking concurrent algorithms, however, more advanced methods of testing and validation are necessary for verification. Mathieu Desnoyers introduces some of these techniques in greater depth in “Proving the Correctness of Nonblocking Data Structures” elsewhere in this issue of *ACM Queue*.

THE WOES OF UNMANAGED LANGUAGES

Languages that lack built-in support for automatic memory management such as C and C++ require specialized techniques for managing dynamically allocated lockless objects. (This section may not be relevant to those who plan on working exclusively with languages that have automatic memory management such as Java and C#.)

A popular scheme for reducing contention in lock-based synchronization involves decoupling the liveness of an object from its reachability. For example, a concurrent cache implementation using lock-based synchronization may have a mutex protecting a cache but a separate mutex protecting concurrent accesses for objects contained within the cache. This scheme is commonly implemented using in-band reference counters:

```
cache_t cache;
object_t *object;

object_t *
cache_lookup(int key)
{
	object_t *object;

	lock(cache);

	/* Returns object associated with that key. */
	object = cache_find(key);

	/* No object found associated with that key. */
	if (object == NULL) {
		unlock(cache);
		return NULL;
	}

	/*
	 * Increment reference counter which was initialized to
	 * 1 when inserted into cache.
	 *
	 * The actual reference counter is contained or is in-band
	 * the actual object it is reference counting.
	 */
	atomic_increment(&object->ref);
	unlock(cache);
	return object;
}

/* Remove an object from cache. */
void
object_delete(object_t *object)
```

```
{
        lock(cache);
        cache_remove(object);
        unlock(cache);

        /* Remove the cache reference. */
        object_delref(object);
}

void
object_delref(object_t *object)
{
        int new_ref;

        /*
         * Atomically decrements the integer pointed to
         * by the argument and returns the newly decremented
         * value.
         */
        new_ref = atomic_decrement(&object->ref);
        if (new_ref == 0) {
                /*
                 * If the reference count is 0 then the object
                 * is no longer reachable and is not currently
                 * being used by other threads, so it is safe to
                 * destroy it.
                 */
                free(object);
        }
}
```

This scheme allows for concurrent accesses to objects fetched from the cache while still allowing for concurrent fetches to proceed (though note that it may not scale for high-frequency, short-lived transactions because of contention over reference counters). The scheme works in the current example because the reachability of the object is managed atomically with respect to the reference counter. In other words, there is never a state in which the reference counter is 0 and the object is still in the cache. The object is never destroyed if there are still references to it. The reference counter in this case provides a mechanism for allowing the program safely to reclaim memory associated with concurrently accessed objects whose reachability is determined by cache state.

On modern processors, nonblocking data structures are implemented in terms of atomic operations that can modify only one target memory location at a time. Assume that the previous cache example was actually transformed to become lock-free. The reachability of an object in the cache is likely determined by a linearization point consisting of an atomic operation to a single memory location. For the reference counting scheme to work, the reachability of the object must be managed atomically with respect to incoming references to it. In the absence of atomic operations that operate on disparate memory locations, this common in-band reference-counting scheme is unable to provide sufficient safety guarantees.[9]

For this reason, dynamic nonblocking and lockless data structures (structures that access dynamically allocated memory that may also be freed at runtime) must typically rely on alternative safe memory-reclamation mechanisms. Besides full-fledged garbage collection, safe memory-reclamation techniques include:

- **EBR (Epoch-Based Reclamation).**[11] This is a passive scheme that allows for safe memory reclamation by carefully monitoring observers of a global epoch counter. It is not suitable for protecting objects that are created and destroyed at a high frequency. Threads attempting synchronous (immediate) object destruction using EBR may block until the mechanism detects a safe destruction point. It is possible to implement variants of this scheme with minimal cost on the fast path.
- **HP (Hazard Pointers).**[22] This scheme works through the live detection of references to objects that require safe memory reclamation. To take advantage of it, nonblocking algorithms require modification that is specific to this scheme. It is more suitable, however, for protecting objects that are created and destroyed at a high frequency, and threads wanting to destroy protected objects are not required to block for an unbounded amount of time. This scheme can come at a high cost (full memory barrier) on the fast path and may not be suitable for traversal-heavy workloads.
- **QSBR (Quiescent-State-Based Reclamation).**[8,20] This is a passive scheme that allows for safe memory reclamation through the detection of quiescent states in a program, in which no references to objects with live references could possibly exist. It is not suitable for protecting objects that are created and destroyed at a high frequency. Threads attempting synchronous (immediate) object destruction using QSBR may block until the mechanism detects a safe destruction point. Variants of this scheme can be implemented with zero cost on the fast path.
- **PC (Proxy Collection).** This is an out-of-band amortized reference-counting scheme.

Additional mechanisms exist to achieve safe memory reclamation.[14] Thomas Hart et al. compared the performance properties of the first three schemes.[12] Note that these mechanisms may have attractive performance properties compared with reference counting for many workloads and are not required to be used exclusively with nonblocking data structures. Paul McKenney explores safe memory reclamation in greater detail in "Structured Deferral: Synchronization via Procrastination" elsewhere in this issue of *ACM Queue*.

RESILIENCE

Systems that require any of the following properties may benefit from the use of lock-free and wait-free algorithms:

- **Deadlock Freedom**. The absence of locking means that wait-free and lock-free data structures are immune to deadlock conditions, which can be difficult to avoid in large and complex locking hierarchies.
- **Async-Signal Safety**. Providing deadlock freedom and coherency in the context of asynchronous interruptions of a critical section is a nontrivial problem. Wait-free and lock-free synchronization is immune to these problems.

• **Termination Safety.** Wait-free and lock-free operations that satisfy linearizability may be aborted at any time. This means that processors or threads that are in the middle of wait-free and lock-free operations may terminate without sacrificing the overall availability of a system.
• **Preemption Tolerance.** With wait freedom, any operation is guaranteed to complete in a finite number of steps in the absence of resource overload, even in the presence of preemption and other external delays. Lock freedom guarantees the overall progress of a system, as a delay in one thread can only incur a bounded delay in another thread (linearizability may require other threads to assist in the completion of the delayed operation, which may require additional instructions on the fast path).
• **Priority Inversion Avoidance.** In the absence of overload to resources such as memory, wait freedom can provide tight-bound guarantees for priority inversion, but this may at times come at a significant cost to the fast path. This additional overhead can be avoided with algorithms specialized for a fixed level of concurrency. Lock freedom can avoid priority inversion on uniprocessor systems at a lower overhead than lock-based synchronization with the right scheduling policies.[2] On multiprocessor systems with high levels of contention, bounded priority inversion is difficult to avoid (the extent to which it can be avoided is a function of the fairness and prioritization guarantees provided by the underlying coherence mechanism, which usually provides little to no guarantee on either). Lock freedom remains vulnerable to unbounded priority inversion even in the absence of overload to memory resources, as a lower-priority thread may still cause any number of delays of a higher-priority thread with interfering operations. Contention avoidance may be used to prevent this situation.

BRIDGING THE GAP FROM ABSTRACT MODELS

It is important to frame the progress guarantees of nonblocking synchronization in the context of real hardware. The wait-freedom progress guarantee can break down in real-world systems under severely high levels of contention over memory and coherency resources. At these levels of contention, it is possible to starve processors from acceptable rates of operation completion. The progress guarantees of your program can be only as good as those of the concurrency primitives it is built on and of the underlying coherency mechanism. Fortunately, as interconnect bandwidth continues to increase and interconnect latency continues to decrease, the severity of this problem dwindles as well.

Lock-based synchronization is, of course, also susceptible to this problem. In the absence of overloading of memory resources and in the presence of jitter, wait-freedom can provide stronger latency-bound guarantees than lock-based synchronization (because of the increased tolerance to external delays). This latency bound is a function of the underlying microarchitecture, memory resources, and contention.

The fast-path latency of write operations on a nonblocking data structure is usually higher than the fast-path latency of a lock-based variant. This trade-off is especially common for nonblocking objects designed to work for any level of concurrency, because they usually require one or more heavier-weight atomic read-modify-write operations on the fast path. This trade-off in fast-path latency is a function of the cost of an underlying platform's lock implementations and the complexity of the nonblocking algorithm.

This concept can be illustrated by comparing the performance of a spinlock-protected stack with

TABLE 4. **Uncontested latency of various operations**

Operation	Intel Core i7-3615QM	IBM Power 730 Express
spinlock_push	17	29
lockfree_push	25	12
spinlock_pop	18	29
lockfree_pop	27	12

a lock-free stack (the implementation used was ck_stack.h from concurrencykit.org). The lock-free stack contains a single `compare_and_swap` operation for both the push and pop operations. On the x86 architecture, the fast-path latency of the spinlock-backed stack implementation is significantly lower than a lock-free stack implementation. On the Power 7 architecture, this is not the case. Table 4 displays uncontested latency (in ticks) of these various operations.

On the x86 it is possible to implement a spinlock using significantly cheaper atomic operations (in this case, the `xchg` instruction) than the ones used in the lock-free stack. The TSO of the architecture does not require the use of explicit memory barriers for this algorithm. On the other hand, the lock-free stack implementation requires the more complex `cmpxchg` (and `cmpxchg16b`) instruction, which exhibits higher baseline latency.

On the Power architecture, both the spinlock and lock-free implementations rely on the same underlying LL/SC primitives. The primary difference is that the spinlock implementation requires a heavyweight memory barrier on invocation of every stack operation, while the nonblocking stack requires only lighter-weight load-and-store memory barriers.

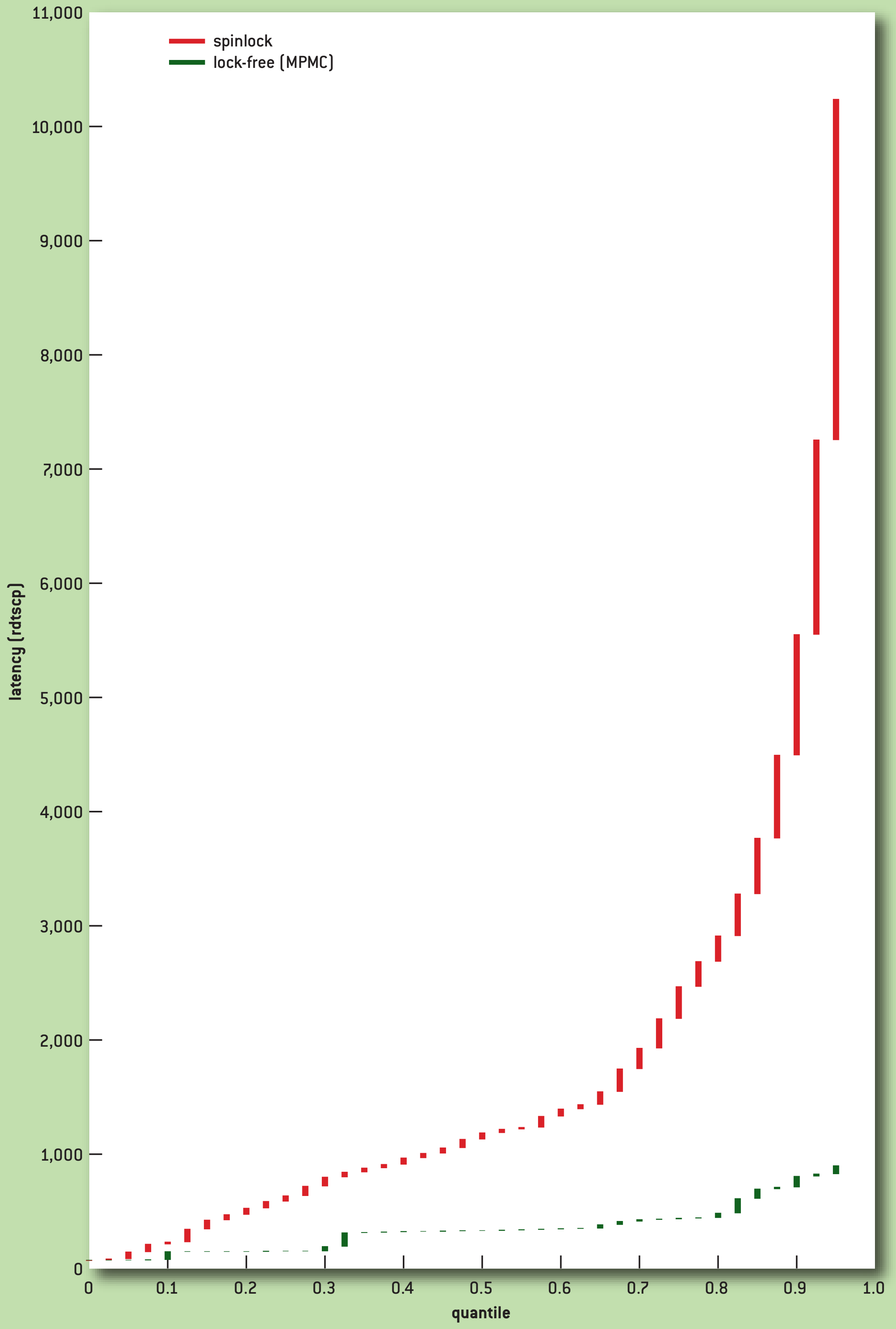
Regardless of the architecture-imposed tradeoff in latency, the desirable properties of nonblocking data structures are revealed under contention. Figure 6 depicts the latency distribution of a push-only workload on a single stack across four threads on an x86 server (Intel Xeon L5640). Active measures were taken to avoid jitter, including use of the `SCHED_FIFO` scheduling class, core affinity, and IRQ (interrupt request) affinity (Linux 2.6.32-100.0.19.el5).

The latency distribution illustrated in figure 6 is *not* a result of a stronger latency bound on individual stack operations, but a side effect of stronger *system-wide* progress guarantees provided by lock-free algorithms. Specifically, lock-free algorithms are able to guarantee progress in the presence of preemption. In the blocking data structure, a preempted or blocked thread inside the stack-operation-critical section prevents system-wide progress. On the other hand, the lock-free stack forces a stack operation retry only if another thread has made progress by updating the stack. If all sources of jitter were removed from the test system, then the latency profile of the spinlock-based stack would prevail.

Asymmetric workloads involving a combination of write- and read-only operations can also benefit from nonblocking synchronization. Often, in combination with the safe memory reclamation techniques described here, it is possible to provide strong guarantees of progress for readers with minimal to no heavyweight synchronization instructions on the fast path. This is a desirable property: it avoids many of the write-side/read-side fairness problems exhibited in popular single-preference read-write locks, while also avoiding fast-path degradation associated with heavier-weight fair read-write locks.

FIGURE 6

Latency Distribution of a Push-Only Workload



CONCLUSION

Having examined some of the common principles of parallel programming on multiprocessor systems and the real-world implications of nonblocking synchronization, this article has aimed to equip readers with the background needed to explore alternatives to lock-based synchronization. Even though the design, implementation, and verification of nonblocking algorithms are all difficult, these algorithms are becoming more prevalent in standard libraries and open source software. The information presented here helps identify situations that call for the resiliency and performance characteristics of nonblocking synchronization.

For those of you who begin the perilous undertaking of designing your own nonblocking algorithms, the other articles in this issue provide valuable information (and please remember the opportunities for simplification through workload specialization).

ACKNOWLEDGMENTS

Thanks to Mathieu Desnoyers and Paul McKenney for their contributions in lockless synchronization to the open source community, as well as their insightful feedback on this article. Thanks to Devon H. O'Dell and Gabriel Parmer for feedback on this article and throughout this whole process. Thanks to Theo Schlossnagle for his support in the early stages of this process. Thanks to Silvio Cesare, Wez Furlong, Maxime Henrion, David Joseph, Paul Khuong, Abel Mathew, Shreyas Prasad, Brendon Scheinman, Andrew Sweeney, and John Wittrock for their helpful feedback. Thanks to Jim Maurer, the editorial board, and the rest of the *ACM Queue* team for their support and feedback.

REFERENCES

1. AMD. 2007. *AMD64 Architecture Programmer's Manual*, Volume 2: System Programming. Publication 24593.
2. Anderson, J. H., Ramamurthy, S., Jeffay, K. 1997. Real-time computing with lock-free shared objects. *ACM Transactions On Computer Systems* 15(2): 134-165; http://doi.acm.org/10.1145/253145.253159.
3. Attiya, H., Ramalingam, G., Rinetzky, N. 2010. Sequential verification of serializability. *SIGPLAN Notices* 45(1): 31-42; http://doi.acm.org/10.1145/1707801.1706305.
4. Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., Vechev, M. 2011. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*: 487-498; http://doi.acm.org/10.1145/1926385.1926442.
5. Calciu, I., Dice, D., Lev, Y., Luchangco, V., Marathe, V. J., Shavit, N. 2013. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*: 157-166; http://doi.acm.org/10.1145/2442516.2442532.
6. Cantrill, B., Bonwick, J. 2008. Real-world concurrency. *Queue* 6(5): 16-25; http://doi.acm.org/10.1145/1454456.1454462.
7. Chong, N., Ishtiaq, S. 2008. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*: 16-19; http://doi.acm.org/10.1145/1353522.1353528.

8. Desnoyers, M., McKenney, P. E., Stern, A. S., Dagenais, M. R., Walpole, J. 2012. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems* 23(2): 375-382.
9. Detlefs, D. L., Martin, P. A., Moir, M., Steele Jr., G. L. 2001. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*: 190-199; http://doi.acm.org/10.1145/383962.384016.
10. Dice, D., Marathe, V. J., Shavit, N. 2012. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*: 247-256; http://doi.acm.org/10.1145/2145816.2145848.
11. Fraser, K. 2003. Practical lock freedom. Ph.D. thesis. Computer Laboratory, University of Cambridge. Also available as Technical Report UCAM-CL-TR-579, Cambridge University.
12. Hart, T. E., McKenney, P. E., Demke Brown, A., Walpole, J. 2007. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing* 67(12): 1270-1285; http://dx.doi.org/10.1016/j.jpdc.2007.04.010.
13. Herlihy, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13(1): 124-149; http://doi.acm.org/10.1145/114005.102808.
14. Herlihy, M., Luchangco, V., Moir, M. 2002. The repeat offender problem: a mechanism for supporting dynamic-sized lock-free data structures. Technical Report. Sun Microsystems Inc.
15. Herlihy, M., Shavit, N. 2008. *The Art of Multiprocessor Programming*. San Francisco: Morgan Kaufmann Publishers Inc.
16. Kandemir, M., Choudhary, A., Banerjee, P., Ramanujam, J. 1999. On reducing false sharing while improving locality on shared memory multiprocessors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*: 203-211. IEEE Computer Society; http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=807529&contentType=Conference+Publications.
17. Liu, T., Berger, E. D. 2011. SHERIFF: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM International Conference on Object-oriented Programming Systems Languages and Applications*: 3-18; http://doi.acm.org/10.1145/2048066.2048070.
18. Luchangco, V., Nussbaum, D., Shavit, N. 2006. A hierarchical CLH queue lock. In Proceedings of the 12th International Conference on Parallel Processing, ed. W. E. Nagel, W. V. Walter, W. Lehner, 801-810. Springer-Verlag Berlin Heidelberg; http://dx.doi.org/10.1007/11823285_84.
19. McKenney, P. E. 1996. Selecting locking primitives for parallel programming. *Communications of the ACM* 39(10): 75-82; http://doi.acm.org/10.1145/236156.236174.
20. McKenney, P. E. 2004. Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels. Ph.D. dissertation. Oregon Health and Science University. AAI3139819.
21. McKenney, P. E. 2011. Is parallel programming hard, and, if so, what can you do about it? kernel.org; https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.
22. Michael, M. M. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15(6): 491-504; http://dx.doi.org/10.1109/TPDS.2004.8.
23. Michael, M. M., Scott, M. L. 1995. Simple, fast, and practical nonblocking and blocking concurrent queue algorithms. Technical Report. University of Rochester, Rochester, NY.

24. Novell. A NUMA API for Linux; http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf.
25. Oracle. 2010. Memory and Thread Placement Optimization Developer's Guide; http://docs.oracle.com/cd/E19963-01/html/820-1691/.
26. Owens, S., Sarkar, S., Sewell, P. 2009. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics*, ed. S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, 391-407. Springer Berlin Heidelberg; http://dx.doi.org/10.1007/978-3-642-03359-9_27.
27. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D. 2011. Understanding POWER multiprocessors. *SIGPLAN Notices* 46(6): 175-186; http://doi.acm.org/10.1145/1993316.1993520.
28. Schneider, F. B. 1997. *On Concurrent Programming*. New York: Springer-Verlag New York Inc.

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

SAMY AL BAHRA is an engineering team lead at AppNexus, playing a key role in the development of a leading realtime online advertising platform. Prior to his work at AppNexus, he was involved with Message Systems in the development of a high-performance messaging server and was the lead developer of the George Washington University High Performance Computing Laboratory UPC I/O library reference implementation. He is the maintainer of Concurrency Kit (http://concurrencykit.org), a library that provides a plethora of specialized concurrency primitives, lockless data structures, and other technologies to aid the research, design, and implementation of high-performance concurrent systems.