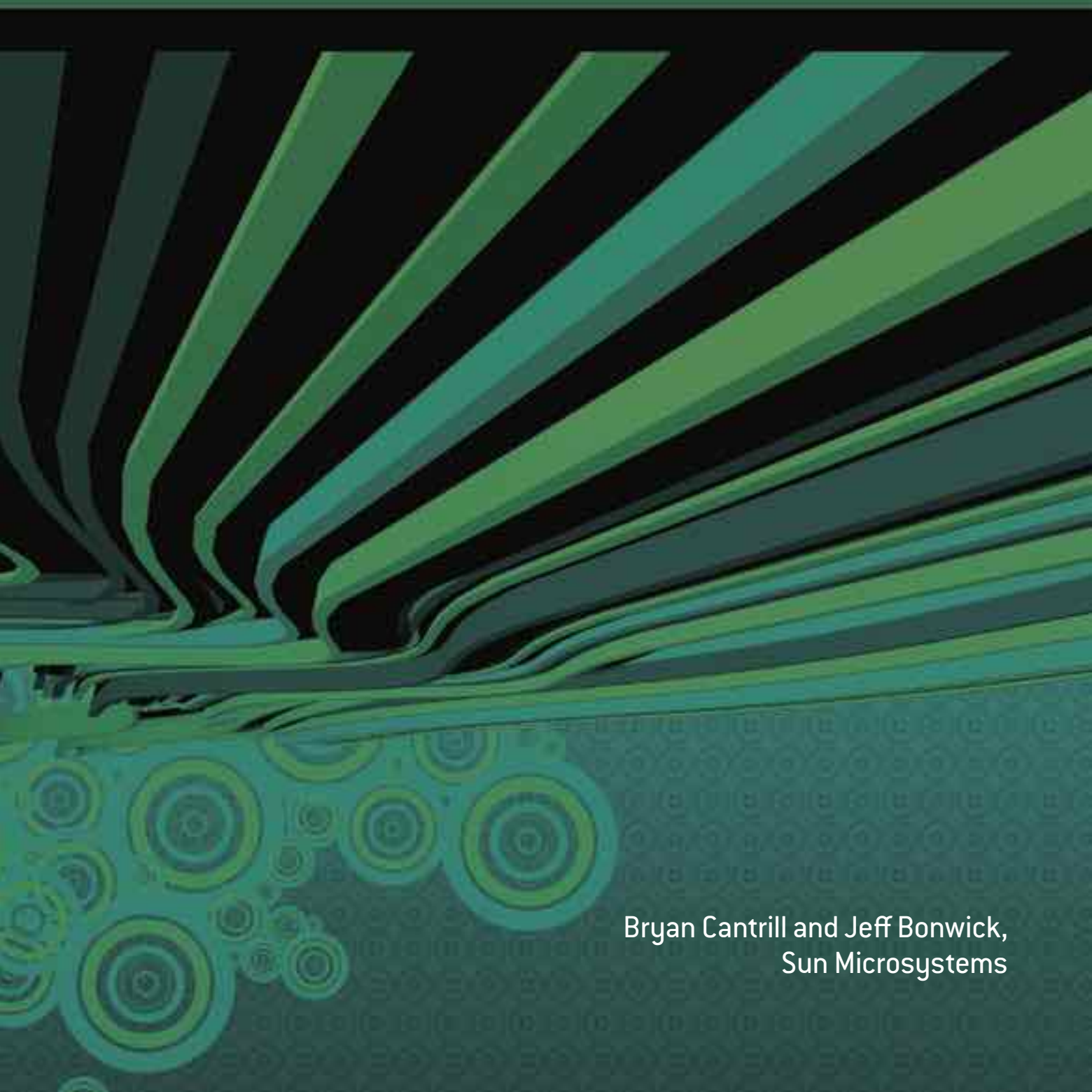


Bryan Cantrill and Jeff Bonwick,
Sun Microsystems

Chances are you won't actually have to write multithreaded code. But if you do, some key principles will help you master this "black art."

Real-world CONCURRENCY

Software practitioners today could be forgiven if recent microprocessor developments have given them some trepidation about the future of software. While Moore's law continues to hold (that is, transistor density continues to double roughly every 18 months), as a result of both intractable physical limitations and practical engineering considerations, that increasing density is no longer being spent on boosting clock rate. Instead, it is being used to put multiple CPU cores on a single CPU die. From the software perspective, this is not a revolutionary shift, but rather an evolutionary one: multicore CPUs are not the birthing of a new paradigm, but rather the progression of an old one (multiprocessing) into more widespread deployment. Judging from many recent articles and papers on the subject, however, one might think that this blossoming of concurrency is the coming of the apocalypse, that "the free lunch is over."[1]

As practitioners who have long been at the coal face of concurrent systems, we hope to inject some calm reality (if not some hard-won wisdom) into a discussion that has too often descended into hysterics. Specifically, we hope to answer the essential question: what does the proliferation of concurrency mean for the software that you develop? Perhaps regrettably, the answer to that question is neither simple nor universal—your software's relationship to concurrency depends on where it physically executes, where it is in the stack of abstraction, and the business model that surrounds it.

Given that many software projects now have components in different layers of the abstraction stack spanning different tiers of the architecture, you may well find that even for the software that you write, you do not have one answer but several: you may be able to leave some of your code forever executing in sequential bliss, and some may need to be highly parallel and explicitly multithreaded. Further complicating the answer, we will argue that much of your code will not fall neatly into either category: it will be essentially sequential in nature but will need to be aware of concurrency at some level.

Real-world CONCURRENCY

Although we assert that less—much less—code needs to be parallel than some might fear, it is nonetheless true that writing parallel code remains something of a black art. We also therefore give specific implementation techniques for developing a highly parallel system. As such, this article is somewhat dichotomous: we try both to argue that most code can (and should) achieve concurrency without explicit parallelism, and at the same time to elucidate techniques for those who must write explicitly parallel code. This article is half stern lecture on the merits of abstinence and half Kama Sutra.

SOME HISTORICAL CONTEXT

Before we discuss concurrency with respect to today's applications, it would be helpful to explore the history of concurrent execution. Even by the 1960s—when the world was still wet with the morning dew of the computer age—it was becoming clear that a single central processing unit executing a single instruction stream would result in unnecessarily limited system performance. While computer designers experimented with different ideas to circumvent this limitation, it was the introduction of the Burroughs B5000 in 1961 that proffered the idea that ultimately proved to be the way forward: disjoint CPUs concurrently executing different instruction streams but sharing a common memory. In this regard (as in many) the B5000 was at least a decade ahead of its time. It was not until the 1980s that the need for multiprocessing became clear to a wider body of researchers, who over the course of the decade explored cache coherence protocols (e.g., the Xerox Dragon and DEC Firefly), prototyped parallel operating systems (e.g., multiprocessor Unix running on the AT&T 3B20A), and developed parallel databases (e.g., Gamma at the University of Wisconsin).

In the 1990s, the seeds planted by researchers in the 1980s bore the fruit of practical systems, with many computer companies (e.g., Sun, SGI, Sequent, Pyramid) placing big bets on symmetric multiprocessing. These bets on concurrent hardware necessitated corresponding bets on concurrent software—if an operating system cannot execute in parallel, neither can much else in the system—and these companies independently came to the realization that their operating systems would need to be rewritten around the notion of concurrent execution. These rewrites took place early in the 1990s, and the resulting systems were polished over the decade; much of the resulting technology can today be seen in open source operating systems such as OpenSolaris, FreeBSD, and Linux.

Just as several computer companies made big bets around multiprocessing, several database vendors made bets around highly parallel relational databases; upstarts including Oracle, Teradata, Tandem, Sybase, and Informix needed to use concurrency to achieve a performance advantage over the mainframes that had dominated transaction processing until that time.[2] As in operating systems, this work was conceived in the late 1980s and early 1990s, and incrementally improved over the course of the decade.

The upshot of these trends was that by the end of the 1990s, concurrent systems had displaced their uniprocessor forebears as high-performance computers: when the TOP500 list of supercomputers was first drawn up in 1993, the highest-performing uniprocessor in the world was just #34, and more than 80 percent of the top 500 were multiprocessors of one flavor or another. By 1997, uniprocessors were off the list entirely. Beyond the supercomputing world, many transaction-oriented applications scaled with CPU, allowing users to realize the dream of expanding a system without revisiting architecture.

The rise of concurrent systems in the 1990s coincided with another trend: while CPU clock rate continued to increase, the speed of main memory was not keeping up. To cope with this relatively slower memory, microprocessor architects incorporated deeper (and more complicated) pipelines, caches, and prediction units. Even then, the clock rates themselves were quickly becoming something of a fib: while the CPU might be able to execute at the advertised rate, only a slim fraction of code could actually achieve (let alone surpass) the rate of one cycle per instruction—most code was mired spending three, four, five (or many more) cycles per instruction.

Many saw these two trends—the rise of concurrency and the futility of increasing clock rate—and came to the logical conclusion: instead of spending transistor budget on "faster" CPUs that weren't actually yielding much in terms of performance gains (and had terrible costs in terms of power, heat, and area), why not take advantage of the rise of concurrent software and use transistors to effect multiple (simpler) cores per die?

That it was the success of concurrent software that contributed to the genesis of chip multiprocessing is an incredibly important historical point and bears reemphasis. There is a perception that microprocessor architects have—out of malice, cowardice, or despair—inflicted concurrency on software.[3] In reality, the opposite is the case: it was the maturity of concurrent software that led architects to consider concurrency on the die. (The reader is referred to one of the earliest chip multiprocessors—DEC's Piranha—for a detailed discussion of this motivation.[4]) Were software not ready, these microprocessors would not be commercially viable today. If anything, the "free lunch" that some decry as being over is in fact, at long last, being served. One need only be hungry and know how to eat!

CONCURRENCY IS FOR PERFORMANCE

The most important conclusion from this foray into history is that concurrency has always been employed for one purpose: to improve the performance of the system. This seems almost too obvious to make explicit—why else would we want concurrency if not to improve performance?—yet for all its obviousness, concurrency's *raison d'être* is increasingly forgotten, as if the proliferation of concurrent hardware has awakened an anxiety that all software must use all available physical resources. Just as no programmer felt a moral obligation to eliminate pipeline stalls on a superscalar microprocessor, no software engineer should feel responsible for using concurrency simply because the hardware supports it. Rather, concurrency should be thought about and used for one reason and one reason only: because it is needs to yield an acceptably performing system.

Concurrent execution can improve performance in three fundamental ways: it can reduce latency (that is, make a unit of work execute faster); it can hide latency (that is, allow the system to continue doing work during a long-latency operation); or it can increase throughput (that is, make the system able to perform more work).

Using concurrency to reduce latency is highly problem-specific in that it requires a parallel algorithm for the task at hand. For some kinds of problems—especially those found in scientific computing—this is straightforward: work can be divided *a priori*, and multiple compute elements set on the task. Many of these problems, however, are often *so* parallelizable that they do not require the tight coupling of a shared memory—and they are often able to execute more economically on grids of small machines instead of a smaller number of highly concurrent ones. Further, using concurrency to reduce latency requires that a unit of work be long enough in its execution to amortize the substantial costs of coordinating multiple compute elements: one can envision using concurrency to parallelize a sort of 40 million elements—but a sort of a mere 40 elements is unlikely to take enough compute time to pay the overhead of parallelism. In short, the degree to which one can use concurrency to reduce latency depends much more on the problem than on those endeavoring to solve it—and many important problems are simply not amenable to it.

For long-running operations that cannot be parallelized, concurrent execution can instead be used to perform useful work while the operation is pending; in this model, the latency of the operation is not reduced, but it is *hidden* by the progression of the system. Using concurrency to hide latency is particularly tempting when the operations themselves are likely to block on entities outside of the program—for example, a disk I/O operation or a DNS lookup. Tempting though it may be, one must be very careful when considering using concurrency merely to hide latency: having a parallel program can become a substantial complexity burden to bear just for improved responsiveness. Further, concurrent execution is *not* the only way to hide system-induced latencies: one can often achieve the same effect by employing nonblocking operations (e.g., asynchronous I/O) and an event loop (e.g., the `poll()`/`select()` calls found in Unix) in an otherwise sequential program. Programmers who wish to hide latency should therefore consider concurrent execution as an option, not as a foregone conclusion.

When problems resist parallelization or have no appreciable latency to hide, the third way that concurrent execution can improve performance is to increase the throughput of the system. Instead of using parallel logic to make a single operation faster, one can employ multiple concurrent executions of sequential logic to accommodate more simultaneous work. Importantly, a system using concurrency to increase throughput need *not* consist exclusively (or even largely) of multithreaded code. Rather, those components of the system that share no state can be left entirely sequential, with the system executing multiple instances of these components concurrently. The sharing in the system can then be offloaded to components explicitly designed around parallel execution on shared state, which can ideally be reduced to those elements already known to operate well in concurrent environments: the database and/or the operating system.

Real-world CONCURRENCY

To make this concrete, in a typical MVC (model-view-controller) application, the view (typically implemented in environments such as JavaScript, PHP, or Flash) and the controller (typically implemented in environments such as J2EE or Ruby on Rails) can consist purely of sequential logic and still achieve high levels of concurrency, provided that the model (typically implemented in terms of a database) allows for parallelism. Given that most don't write their own databases (and virtually no one writes their own operating systems), it is possible to build (and indeed, many have built) highly concurrent, highly scalable MVC systems without explicitly creating a single thread or acquiring a single lock; it is concurrency by architecture instead of by implementation.

ILLUMINATING THE BLACK ART

What if you are the one developing the operating system or database or some other body of code that must be explicitly parallelized? If you count yourself among the relative few who need to write such code, you presumably do not need to be warned that writing multithreaded code is hard. In fact, this domain's reputation for difficulty has led some to conclude (mistakenly) that writing multithreaded code is simply impossible: "No one knows how to organize and maintain large systems that rely on locking," reads one recent (and typical) assertion.[5] Part of the difficulty of writing scalable and correct multithreaded code is the scarcity of written wisdom from experienced practitioners: oral tradition in lieu of formal writing has left the domain shrouded in mystery. So in the spirit of making this domain less mysterious for our fellow practitioners (if not also to demonstrate that some of us actually *do* know how to organize and maintain large lock-based systems), we present our collective bag of tricks for writing multithreaded code.

Know your cold paths from your hot paths. If there is one piece of advice to dispense to those who must develop parallel systems, it is to know which paths through your code you want to be able to execute in parallel (the hot paths) versus which paths can execute sequentially without affecting performance (the cold paths). In our experience, much of the software we write is bone-cold in terms of concurrent execution: it is executed only when initializing, in administrative paths, when unloading, etc. Not only is it a waste of time to make such cold paths execute with a high degree of parallelism, but it is also dangerous: these paths are often among the most difficult and error-prone to parallelize.

In cold paths, keep the locking as coarse-grained as possible. Don't hesitate to have one lock that covers a wide range of rare activity in your subsystem. Conversely, in hot paths—those that must execute concurrently to deliver highest throughput—you must be much more careful: locking strategies must be simple and fine-grained, and you must be careful to avoid activity that can become a bottleneck. And what if you don't know if a given body of code will be the hot path in the system? In the absence of data, err on the side of assuming that your code is in a cold path and adopt a correspondingly coarse-grained locking strategy—but be prepared to be proven wrong by the data.

Intuition is frequently wrong—be data intensive. In our experience, many scalability problems can be attributed to a hot path that the developing engineer originally believed (or hoped) to be a cold path. When cutting new software from whole cloth, you will need some intuition to reason about hot and cold paths—but once your software is functional, even in prototype form, the time for intuition has ended: your gut must defer to the data. Gathering data on a concurrent system is a tough problem in its own right. It requires you first to have a machine that is sufficiently concurrent in its execution to be able to highlight scalability problems. Once you have the physical resources, it requires you to put load on the system that resembles the load you expect to see when your system is deployed into production. Once the machine is loaded, you must have the infrastructure to be able to dynamically instrument the system to get to the root of any scalability problems.

The first of these problems has historically been acute: there was a time when multiprocessors were so rare that many software development shops simply didn't have access to one. Fortunately, with the rise of multicore CPUs, this is no longer a problem: there is no longer any excuse for not being able to find at least a two-processor (dual-core) machine, and with only a little effort, most will be able (as of this writing) to run their code on an eight-processor (two-socket, quad-core) machine.

Even as the physical situation has improved, however, the second of these problems—knowing how to put load on the system—has worsened: production deployments have become increasingly complicated, with loads that

are difficult and expensive to simulate in development. As much as possible, you must treat load generation and simulation as a first-class problem; the earlier you tackle this problem in your development, the earlier you will be able to get critical data that may have tremendous implications for your software. Although a test load should mimic its production equivalent as closely as possible, timeliness is more important than absolute accuracy: the absence of a perfect load simulation should not prevent you from simulating load altogether, as it is much better to put a multithreaded system under the wrong kind of load than under no load whatsoever.

Once a system is loaded—be it in development or in production—it is useless to software development if the impediments to its scalability can't be understood. Understanding scalability inhibitors on a production system requires the ability to safely dynamically instrument its synchronization primitives. In developing Solaris, our need for this was so historically acute that it led one of us (Bonwick) to develop a technology (lockstat) to do this in 1997. This tool became instantly essential—we quickly came to wonder how we ever resolved scalability problems without it—and it led the other of us (Cantrill) to further generalize dynamic instrumentation into DTrace, a system for nearly arbitrary dynamic instrumentation of production systems that first shipped in Solaris in 2004, and has since been ported to many other systems including FreeBSD and Mac OS.[6] (The instrumentation methodology in lockstat has been reimplemented to be a DTrace provider, and the tool itself has been reimplemented to be a DTrace consumer.)

Today, dynamic instrumentation continues to provide us with the data we need not only to find those parts of the system that are inhibiting scalability, but also to gather sufficient data to understand which techniques will be best suited for reducing that contention. Prototyping new locking strategies is expensive, and one's intuition is frequently wrong; before breaking up a lock or rearchitecting a subsystem to make it more parallel, we always strive to have the data in hand indicating that the subsystem's lack of parallelism is a clear inhibitor to system scalability!

Know when—and when not—to break up a lock. Global locks can naturally become scalability inhibitors, and when gathered data indicates a single hot lock, it is reasonable to want to break up the lock into per-CPU locks, a hash table of locks, per-structure locks, etc. This might ultimately be the right course of action, but before blindly proceeding down that (complicated) path, carefully examine the work done under the lock: breaking up a lock is not the only way to reduce contention, and contention can be (and often is) more easily reduced by decreasing the hold time of the lock. This can be done by algorithmic improvements (many scalability improvements have been achieved by reducing execution under the lock from quadratic time to linear time!) or by finding activity that is needlessly protected by the lock. Here's a classic example of this latter case: if data indicates that you are spending time (say) deallocating elements from a shared data structure, you could dequeue and gather the data that needs to be freed with the lock held and defer the actual deallocation of the data until after the lock is dropped. Because the data has been removed from the shared data structure under the lock, there is no data race (other threads see the removal of the data as atomic), and lock hold time has been decreased with only a modest increase in implementation complexity.

Be wary of readers/writer locks. If there is a novice error when trying to break up a lock, it is this: seeing that a data structure is frequently accessed for reads and infrequently accessed for writes, one may be tempted to replace a mutex guarding the structure with a readers/writer lock to allow for concurrent readers. This seems reasonable, but unless the hold time for the lock is long, this solution will scale no better (and indeed, may scale worse) than having a single lock. Why? Because the state associated with the readers/writer lock must itself be updated atomically, and in the absence of a more sophisticated (and less space-efficient) synchronization primitive, a readers/writer lock will use a single word of memory to store the number of readers. Because the number of readers must be updated atomically, acquiring the lock as a reader requires the same bus transaction—a read-to-own—as acquiring a mutex, and contention on that line can hurt every bit as much.

There are still many situations where long hold times (e.g., performing I/O under a lock as reader) more than pay for any memory contention, but one should be sure to gather data to make sure that it is having the desired effect on scalability. Even in those situations where a readers/writer lock is appropriate, an additional note of caution is warranted around blocking semantics. If, for example, the lock implementation blocks new readers when a writer is blocked (a common paradigm to avoid writer starvation), *one cannot recursively acquire a lock as reader*: if a writer blocks between the initial acquisition as reader and the recursive acquisition as reader, deadlock will result when the recursive acquisition is blocked. All of this is not to say that readers/writer locks shouldn't be used—just that they shouldn't be romanticized.

Real-world CONCURRENCY

Consider per-CPU locking. Per-CPU locking (that is, acquiring a lock based on the current CPU identifier) can be a convenient technique for diffracting contention, as a per-CPU lock is not likely to be contended (a CPU can run only one thread at a time). If one has short hold times and operating modes that have different coherence requirements, one can have threads acquire a per-CPU lock in the common (noncoherent) case, and then force the uncommon case to grab all the per-CPU locks to construct coherent state. Consider this concrete (if trivial) example: if one were implementing a global counter that is frequently updated but infrequently read, one could implement a per-CPU counter protected by its own lock. Updates to the counter would update only the per-CPU copy, and in the uncommon case in which one wanted to read the counter, all per-CPU locks could be acquired and their corresponding values summed.

Two notes on this technique: first, it should be employed only when the data indicates that it's necessary, as it clearly introduces substantial complexity into the implementation; second, be sure to have a single order for acquiring all locks in the cold path: if one case acquires the per-CPU locks from lowest to highest and another acquires them from highest to lowest, deadlock will (naturally) result.

Know when to broadcast—and when to signal. Virtually all condition variable implementations allow threads waiting on the variable to be awakened either via a signal (in which case one thread sleeping on the variable is awakened) or via a broadcast (in which case all threads sleeping on the variable are awakened). These constructs have subtly different semantics: because a broadcast will awaken all waiting threads, it should generally be used to indicate *state change* rather than *resource availability.* If a condition broadcast is used when a condition signal would have been more appropriate, the result will be a *thundering herd*: all waiting threads will wake up, fight over the lock protecting the condition variable, and (assuming that the first thread to acquire the lock also consumes the available resource) sleep once again when they discover that the resource has been consumed. This needless scheduling and locking activity can have a serious effect on performance, especially in Java-based systems, where `notifyAll()` (i.e., broadcast) seems to have entrenched itself as a preferred paradigm; changing these calls to `notify()` (i.e., signal) has been known to result in substantial performance gains.[7]

Learn to debug postmortem. Among some Cassandras of concurrency, a deadlock seems to be a particular bogeyman of sorts, having become the embodiment of all that is difficult in lock-based multithreaded programming. This fear is somewhat peculiar, because deadlocks are actually among the simplest pathologies in software: because (by definition) the threads involved in a deadlock cease to make forward progress, they do the implementer the service of effectively freezing the system with all state intact. To debug a deadlock, one need have only a list of threads, their corresponding stack backtraces, and some knowledge of the system. This information is contained in a snapshot of state so essential to software development that its very name reflects its origins at the dawn of computing: it is a *core dump*.

Debugging from a core dump—*postmortem* debugging—is an essential skill for those who implement parallel systems: problems in highly parallel systems are not necessarily reproducible, and a single core dump is often one's only chance to debug them. Most debuggers support postmortem debugging, and many allow user-defined extensions.[8] We encourage practitioners to understand their debugger's support for postmortem debugging (especially of parallel programs) and to develop extensions specific to debugging their systems.

Design your systems to be composable. Among the more galling claims of the detractors of lock-based systems is the notion that they are somehow uncomposable: "Locks and condition variables do not support modular programming," reads one typically brazen claim, "building large programs by gluing together smaller programs[:] locks make this impossible."[9] The claim, of course, is incorrect. For evidence one need only point at the composition of lock-based systems such as databases and operating systems into larger systems that remain entirely unaware of lower-level locking.

There are two ways to make lock-based systems completely composable, and each has its own place. First (and most obviously), one can make locking entirely internal to the subsystem. For example, in concurrent operating systems, control never returns to user level with in-kernel locks held; the locks used to implement the system itself are entirely behind the system call interface that constitutes the interface to the system. More generally, this model can work whenever a crisp interface exists between

software components: as long as control flow is never returned to the caller with locks held, the subsystem will remain composable.

Second (and perhaps counterintuitively), one can achieve concurrency and composability by having no locks whatsoever. In this case, there must be no global subsystem state—subsystem state must be captured in per-instance state, and it must be up to consumers of the subsystem to assure that they do not access their instance in parallel. By leaving locking up to the client of the subsystem, the subsystem itself can be used concurrently by different subsystems and in different contexts. A concrete example of this is the AVL tree implementation used extensively in the Solaris kernel. As with any balanced binary tree, the implementation is sufficiently complex to merit componentization, but by not having any global state, the implementation may be used concurrently by disjoint subsystems—the only constraint is that manipulation of a single AVL tree instance must be serialized.

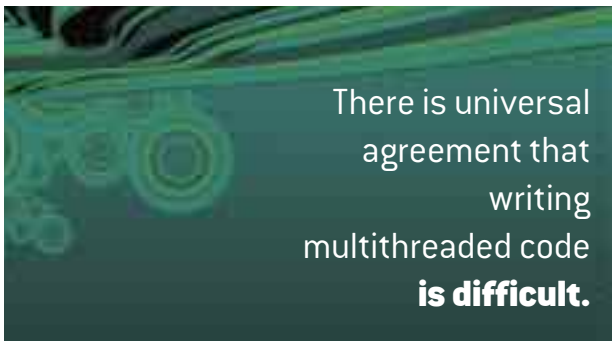
Don't use a semaphore where a mutex would suffice. A semaphore is a generic synchronization primitive originally described by Dijkstra that can be used to effect a wide range of behavior. It may be tempting to use semaphores in lieu of mutexes to protect critical sections, but there is an important difference between the two constructs: unlike a semaphore, a mutex has a notion of *ownership*—the lock is either owned or not, and if it is owned, it has a known owner. By contrast, a semaphore (and its kin, the condition variable) has no notion of ownership: when sleeping on a semaphore, one has no way of knowing which thread one is blocking upon.

The lack of ownership presents several problems when used to protect critical sections. First, there is no way of propagating the blocking thread's scheduling priority to the thread that is in the critical section. This ability to propagate scheduling priority—*priority inheritance*—is critical in a realtime system, and in the absence of other protocols, semaphore-based systems will always be vulnerable to priority inversions. A second problem with the lack of ownership is that it deprives the system of the ability to make assertions about itself. For example, when ownership is tracked, the machinery that implements thread blocking can detect pathologies such as deadlocks and recursive lock acquisitions, inducing fatal failure (and that all-important core dump) upon detection. Finally, the lack of ownership makes debugging much more onerous. A common pathology in a multithreaded system is a lock not being dropped in some errant return path. When ownership is tracked, one at least has the smoking gun of the past (faulty) owner—and, thus, clues as to the code path by which the lock was not correctly dropped. Without ownership, one is left clueless and reduced to debugging by staring at code/the ceiling/into space.

All of this is not to say that semaphores shouldn't be used (indeed, some problems are uniquely suited to a semaphore's semantics), just that they shouldn't be used when mutexes would suffice.

Consider memory retiring to implement per-chain hash-table locks. Hash tables are common data structures in performance-critical systems software, and sometimes they must be accessed in parallel. In this case, adding a lock to each hash chain, with the per-chain lock held while readers or writers iterate over the chain, seems straightforward. The problem, however, is resizing the table: dynamically resizing a hash table is central to its efficient operation, and the resize means changing the memory that contains the table. That is, in a resize the pointer to the hash table must change—but we do not wish to require hash lookups to acquire a global lock to determine the current hash table!

This problem has several solutions, but a (relatively) straightforward one is to *retire* memory associated with



old hash tables instead of freeing it. On a resize, all per-chain locks are acquired (using a well-defined order to prevent deadlock), and a new table is then allocated, with the contents of the old hash table being rehashed into the new table. After this operation, the old table is *not* deallocated but rather placed in a queue of old hash tables. Hash lookups then require a slight modification to operate correctly: after acquiring the per-chain lock, the lookup must check the hash-table pointer and compare it with the hash-table pointer that was used to determine the hash chain. If the hash table has changed (that is, if a hash resize has occurred), it must drop the lock and repeat the lookup (which will acquire the correct chain lock in the new table).

There are some delicate issues in implementing this—the hash-table pointer must be declared volatile,

and the size of the hash table must be contained in the table itself—but the implementation complexity is modest given the alternatives, and (assuming hash tables are doubled when they are resized) the cost in terms of memory is only a factor of two. For an example of this in production code, the reader is directed to the file descriptor locking in Solaris, the source code for which can be found by searching the Internet for "flist_grow."

Be aware of false sharing. There are a variety of different protocols for keeping memory coherent in caching multiprocessor systems. Typically, these protocols dictate that only a single cache may have a given line of memory in a dirty state. If a different cache wishes to write to the dirty line, the new cache must first read-to-own the dirty line from the owning cache. The size of the line used for coherence (the coherence granularity) has an important ramification for parallel software: because only one cache may own a line a given time, one wishes to avoid a situation where two (or more) small, disjoint data structures are both contained within a single line *and* accessed in parallel by disjoint caches. This situation—called *false sharing*—can induce suboptimal scalability in otherwise scalable software. This most frequently arises in practice when one attempts to defract contention with an array of locks: the size of a lock structure is typically no more than the size of a pointer or two and is usually quite a bit less than the coherence granularity (which is typically on the order of 64 bytes). Disjoint CPUs acquiring different locks can therefore potentially contend for the same cache line.

False sharing is excruciating to detect dynamically: it requires not only a bus analyzer, but also a way of translating from the physical addresses of the bus to the virtual addresses that make sense to software, and then from there to the actual structures that are inducing the false sharing. (This process is so arduous and error-prone that we have experimented—with some success—with static mechanisms to detect false sharing.[10]) Fortunately, false sharing is rarely the single greatest scalability inhibitor in a system, and it can be expected to be even less of an issue on a multicore system (where caches are more likely to be shared among CPUs). Nonetheless, this remains an issue that the practitioner should be aware of, especially when creating arrays that are designed to be accessed in parallel. (In this situation, array elements should be padded out to be a multiple of the coherence granularity.)

Consider using nonblocking synchronization routines to monitor contention. Many synchronization primitives have different entry points to specify different behavior if the primitive is unavailable: the default entry point will typically block, whereas an alternative entry point will return an error code instead of blocking. This second variant has a number of uses, but a particularly interesting one is the monitoring of one's own contention: when an attempt to acquire a synchronization primitive fails, the subsystem can know that there is contention. This can be especially useful if a subsystem has a way of dynamically reducing its contention. For example, the Solaris kernel memory allocator has per-CPU caches of memory buffers. When a CPU exhausts its per-CPU caches, it must obtain a new series of buffers from a global pool. Instead of simply acquiring a lock in this case, the code *attempts* to acquire the lock, incrementing a counter when this fails (and then acquiring the lock through the blocking entry point). If the counter reaches a predefined threshold, the size of the per-CPU caches is increased, thereby dynamically reducing contention.

When reacquiring locks, consider using generation counts to detect state change. When lock ordering becomes complicated, at times one will need to drop one lock, acquire another, and then reacquire the first. This can be tricky, as state protected by the first lock may have changed during the time that the lock was dropped—and reverifying this state may be exhausting, inefficient, or even impossible. In these cases, consider associating a generation count with the data structure; when a change is made to the data structure, a generation count is bumped. The logic that drops and reacquires the lock must cache the generation before dropping the lock, and then check the generation upon reacquisition: if the counts are the same, the data structure is as it was when the lock was dropped and the logic may proceed; if the count is different, the state has changed and the logic may react accordingly (for example, by reattempting the larger operation).

Use wait- and lock-free structures only if you absolutely must. Over our careers, we have each implemented wait- and lock-free data structures in production code, but we did this only in contexts in which locks could not be acquired for reasons of correctness. Examples include the implementation of the locking system itself,[11] the subsystems that span interrupt levels, and dynamic instrumentation facilities.[12] These constrained contexts are the

exception, not the rule; in normal contexts, wait- and lock-free data structures are to be avoided as their failure modes are brutal (livelock is much nastier to debug than deadlock), their effect on complexity and the maintenance burden is significant, and their benefit in terms of performance is usually nil.

Prepare for the thrill of victory—and the agony of defeat. Making a system scale can be a frustrating pursuit: the system will not scale until all impediments to scalability have been removed, but it is often impossible to know if the current impediment to scalability is the last one. Removing that last impediment is incredibly gratifying: with that change, throughput finally gushes through the system as if through an open sluice. Conversely, it can be disheartening to work on a complicated lock breakup only to discover that while it *was* the impediment to scalability, it was merely hiding another impediment, and removing it improves performance very little—or perhaps not at all. As discouraging as it may be, you must return to the system to gather data: does the system not scale because the impediment was misunderstood, or does it not scale because a new impediment has been encountered? If the latter is the case, you can take solace in knowing that your work is necessary—though not sufficient—to achieve scalability, and that the glory of one day flooding the system with throughput still awaits you.

THE CONCURRENCY BUFFET

There is universal agreement that writing multithreaded code is difficult: although we have attempted to elucidate some of the lessons learned over the years, it nonetheless remains, in a word, hard. Some have become fixated on this difficulty, viewing the coming of multicore computing as cataclysmic for software. This fear is unfounded, for it ignores the fact that relatively few software engineers actually need to write multithreaded code: for most, concurrency can be achieved by standing on the shoulders of those subsystems that already are highly parallel in implementation. Those practitioners who are implementing a database or an operating system or a virtual machine will continue to need to sweat the details of writing multithreaded code, but for everyone else, the challenge is not how to implement those components but rather how best to use them to deliver a scalable system. While lunch might not be exactly free, it *is* practically all-you-can-eat—and the buffet is open! Q

REFERENCES

1. Sutter, H., Larus, J. 2005. Software and the concurrency revolution. *ACM Queue* 3(7): 54-62.
2. DeWitt, D., Gray, J. 1992. Parallel database systems: the future of high-performance database systems. *Communications of the ACM* 35(6): 85-98.
3. Oskin, M. 2008. The revolution inside the box. *Communications of the ACM* 51(7): 70-78.
4. Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., Verghese, B. 2000. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*: 282-293.
5. Shavit, N. 2008. Transactions are tomorrow's loads and stores. *Communications of the ACM* 51(8): 90.
6. Cantrill, B. 2006. Hidden in plain sight. *ACM Queue* 4(1): 26-36.
7. McKusick, K. A. 2006. A conversation with Jarod Jenson. *ACM Queue* 4(1): 16-24.
8. Cantrill, B. 2003. Postmortem object type identification. In *Proceedings of the Fifth International Workshop on Automated Debugging*.
9. Peyton Jones, S. 2007. Beautiful concurrency. In *Beautiful Code*, ed. A. Oram and G. Wilson. Cambridge, MA: O'Reilly.
10. See reference 8.
11. Cantrill, B. 2007. A spoonful of sewage. In *Beautiful Code*, ed. A. Oram and G. Wilson. Cambridge, MA: O'Reilly.
12. See reference 6.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

BRYAN CANTRILL is a Distinguished Engineer at Sun Microsystems, where he has worked on concurrent systems since coming to Sun to work with Jeff Bonwick on Solaris performance in 1996. Along with colleagues Mike Shapiro and Adam Leventhal, Cantrill developed DTrace, a facility for dynamic instrumentation of production systems that was directly inspired by his frustration in understanding the behavior of concurrent systems.

JEFF BONWICK is a Fellow at Sun Microsystems, where he has worked on concurrent systems since 1990. He is best known for inventing and leading the development of Sun's ZFS (Zettabyte File System), but prior to this he was known for having written (or rather, rewritten) many of the most parallel subsystems in the Solaris kernel, including the synchronization primitives, the kernel memory allocator, and the thread-blocking mechanism.