What does the proliferation of concurrency mean for the software you develop?

BY BRYAN CANTRILL AND JEFF BONWICK

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), due to both intractable physical limitations and practical engineering considerations, that increasing density is no longer being spent on boosting clock rate, but rather on putting multiple CPU cores on a single CPU die. From the software perspective, this 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. 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."¹⁰

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. And 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: some of your code may be able to be left forever executing in sequential bliss, and some of your code 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. While we will 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 will also therefore give specific implementation techniques for developing a highly parallel system. As such, this article will be somewhat dichotomous: we will try to both argue that most code can (and should) achieve concurrency without explicit parallelism, and at the same time elucidate techniques for those who must write explicitly parallel code. Indeed, this article is half stern lecture on the merits of abstinence and half Kama Sutra.

Some Historical Context

Before discussing concurrency with respect to today's applications, it is 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 limi-

tation, it was the introduction of the Burroughs B5000 in 1961 that captured 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. But 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 (for example, the Xerox Dragon and DEC Firefly), prototyped parallel operating systems (for example, multiprocessor Unix running on the AT&T 3B20A), and developed par-

allel databases (for example, Gamma at the University of Wisconsin).

In the 1990s, the seeds planted by researchers in the 1980s bore the fruit of practical, shipping systems, with many computer companies (for example, 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, not much else in the system can either. These companies came to the realization (independently) that their operating systems must be rewritten around the notion of concurrent execution. These rewrites took place in

the early 1990s and the resulting systems were polished over the decade. In fact, much of the resulting technology can today be seen in open source operating systems like 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 like 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.⁵ 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, with over 80% of the Top 500 being 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 cost 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.⁷ In reality, the opposite is the case: it was the maturity of concurrent software that led architects to consider concurrency on the die. (Readers are referred to one of the earliest chip multiprocessors—

DEC’s Piranha—for a detailed discussion of this motivation.¹) Were software not ready, these microprocessors would not be commercially viable today. So 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 our foray into the history of concurrency 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? And yet for all its obviousness, concurrency’s *raison d’être* is seemingly 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 superscaler microprocessor, no software engineer should feel responsible for using concurrency simply because the hardware supports it. Rather, concurrency should be considered and used for one reason only: because it is needed to yield an acceptably performing system.

There are three fundamental ways in which concurrent execution can improve performance: to reduce latency (that is, to make a unit of work execute faster); to hide latency (that is, to allow the system to continue doing work during a long latency operation); or to increase throughput (that is, to 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. But many of these problems are often so parallelizable they do not require the tight coupling of a shared memory—and they are often able to more economically execute 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 one can use concurrency to reduce latency depends much more on the problem than 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 to merely hide latency: having a parallel program can become a substantial complexity burden to bear for just improved responsiveness. Further, concurrent execution is *not* the only way to hide system-induced latencies: one can often achieve the same effect by employing non-blocking operations (for example, asynchronous I/O) and an event loop (for example, the `poll()/select()` calls found in Unix) in an otherwise sequential program. Programs that 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, there is a third way to use concurrent execution to improve performance: concurrency may also be used to increase the throughput of the system. That is, instead of using parallel logic to make a single operation faster, one can employ multiple concurrent executions of sequential logic to be able 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 be ideally reduced to those elements already known to operate well in concurrent environments: the database and/or the operating system. To make this concrete, in a typical Model/View/Controller application, the View (typically implemented in environments like JavaScript, PHP, or Flash) and the Controller (typically implemented in environments like 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 database (and virtually no one writes their own operating system), 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

But 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 difficult. In fact, this domain's reputation for difficulty has led some to (mistakenly) conclude 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.⁹ 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 to also demonstrate that some of us actually *do* know how to organize and maintain large lock-based systems), we present some of our collective tricks for writing multithreaded code.

Know your cold paths from your hot paths. If there is one piece of advice to

Tempting though it may be, one must be very careful when considering using concurrency to merely hide latency: having a parallel program can become a substantial complexity burden to bear for just improved responsiveness.

dispense to those that 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 only executed when initializing, in administrative paths, when unloading, and so on. It is not only a waste of time to make such cold paths execute with a high degree of parallelism, it is 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—in those paths 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, some intuition will be required to reason about hot and cold paths. However, once your software is functional in even prototype form, the time for intuition is over: your gut must defer to the data. Gathering data on a concurrent system is a tough problem in its own right. It requires you 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 put load on the system that resembles the load you expect to see when your system is deployed into production. And 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 his-

torically 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.

But if the physical situation has improved, the second of these problems—knowing how to put load on the system—has worsened: production deployments have become increas-

ingly complicated, with loads that are difficult and expensive to simulate in development. It is essential that load generation and simulation be treated as a first-class problem; the earlier this development problem is tackled, the earlier you will be able to get critical data that may have tremendous implications for the software. And while 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 de-

velopment 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, and in developing Solaris, our need for this was so historically acute that it led one of us (Bonwick) to develop a technology to do this (lockstat) 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

rearchitecting a subsystem to make it more parallel, we always strive to have the data in hand indicating 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, and so on. 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 reducing the hold time of the lock. This can be done by algorithmic improvements (many scalability improvements have been had by reducing execution under the lock from quadratic time to linear time!) or by finding activity that is needlessly protected by the lock. As 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 reduced 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, it can be tempting 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 well 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 num-

instrumentation of production systems that first shipped in Solaris in 2004, and has since been ported to many other systems including FreeBSD and MacOS.³ (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 to not only to find those parts of the system that are inhibiting scalability, but to gather sufficient data to understand which techniques will be best suited to reduce that contention. Prototyping new locking strategies is expensive, and one's intuition is frequently wrong; before breaking up a lock or

ber 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 (for example, 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. And 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 blocker (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. None of this is to say that readers/writer locks shouldn't be used, just that they shouldn't be romanticized.

Know when to broadcast—and when to signal. Virtually all condition variable implementations allow threads waiting on the variable to be awoken either via a signal (in which case one thread sleeping on the variable is awoken) or via a broadcast (in which case all threads sleeping on the variable are awoken). 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()` (that is, broadcast) seems to have entrenched itself as a preferred paradigm; changing these calls to `notify()` (or `signal`) has been known to result in substantial performance gains.⁶

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.”⁸ The claim, of course, is incorrect; for evidence one need only point at the composition of lock-based systems like 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 there is a crisp interface between software components: as long as control flow is never returned to the caller with locks held, the subsystem will remain composable.

Secondly, (and perhaps counterintuitively), one can achieve concurrency and composability by having no locks whatsoever. In this case, there must be no global subsystem state; all 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 that is 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 example AVL tree instance must be serialized.

The Concurrency Buffet

It's difficult to communicate over a decade of accumulated wisdom in a single article, and space does not permit us ex-

ploration of the more arcane (albeit important) techniques we have used to deliver concurrent software that are both high-performing and reliable. Despite our attempt to elucidate some of the important lessons that we have learned over the years, concurrent software remains, in a word, difficult. 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 need to actually write multithreaded code: for most, concurrency can be achieved by standing on the shoulders of those subsystems that are highly parallel in implementation. Those practitioners implementing a database or an operating system or a virtual machine will continue to sweat the details of writing multithreaded code, but for everyone else, the challenge is not how to implement those components but rather how to best use them to deliver a scalable system. And while lunch might not be exactly free, it is practically all-you-can-eat. The buffet is open. Enjoy!

References

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

Bryan Cantrill is a Distinguished Engineer at Sun Microsystems, where he works on concurrent systems. Along with colleagues Mike Shapiro and Adam Leventhal, Bryan developed DTrace, a facility for dynamic instrumentation of production systems that was directly inspired by Bryan's frustration in understanding the behavior of concurrent systems.

Jeff Bonwick is a Fellow at Sun Microsystems, where he works on concurrent systems. Best known for inventing and leading the development of Sun's Zettabyte Filesystem (ZFS), Bonwick has also written (or rather, rewritten) many of the most parallel subsystems in the Solaris kernel, including the synchronization primitives, kernel memory allocator, and thread-blocking mechanism.