Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Sharing Is the Root of All Contention

Sharing requires waiting and overhead, and is a natural enemy of scalability

February 13, 2009
URL:http://www.drdobbs.com/architecture-and-design/sharing-is-the-root-of-all-contention/214100002

Quick: What fundamental principle do all of the following have in common?

- Two children drawing with crayons.
- Three customers at Starbucks adding cream to their coffee.
- Four processes running on a CPU core.
- Five threads updating an object protected by a mutex.
- Six threads running on different processors updating the same lock-free queue.
- Seven modules using the same reference-counted shared object.

Answer: In each case, multiple users share a resource that requires coordination because not everyone can use it simultaneously — and *such sharing is the root cause of all resulting contention* that will degrade everyone's performance. (Note: The only kind of shared resource that doesn't create contention is one that inherently requires no synchronization because all users can use it simultaneously without restriction, which means it doesn't fall into the description above. For example, an immutable object can have its unchanging bits be read by any number of threads or processors at any time without synchronization.)

In this article, I'll deliberately focus most of the examples on one important case, namely mutable (writable) shared objects in memory, which are an inherent bottleneck to scalability on multicore systems. But please don't lose sight of the key point, namely that "sharing causes contention" is a general principle that is not limited to shared memory or even to computing.

The Inherent Costs of Sharing

Here's the issue in one sentence: Sharing fundamentally requires waiting and demands answers to expensive questions.

As soon as something is shared in a way that doesn't allow unlimited simultaneous use, we are forced to partially or fully serialize access to it. The word "serialize" should leap out as a bright red flag -- it is the inverse and antithesis of "parallelize," and a fundamental enemy of scalability. That the very act of sharing a thing demands overhead is inherent in the notion of sharing itself, not only in software; it applies equally to "one child at a time can use the blue crayon" as to "one process at a time can write to the file" and "one writer at a time can use the shared object."

That we're forced to serialize access, in turn, means two things: we are forced to (potentially) wait, and we are also forced to start answering expensive questions like: "*Can I use it now?*" Computing the answer to any

question in software costs CPU cycles. But these are more expensive questions, because answering them requires agreement at many levels of the system — across software threads and processes, and across hardware processors and memory subsystems — and getting that agreement can cost substantial communication and synchronization overhead that's not even visible in the source code.

A Roadmap

Table 1 helps to map out these costs by breaking down their sources (rows) and how they manifest (columns), and giving examples of each case. The rows categorize the sources into three kinds of sharing overhead, where the first two are fundamental and the third arises when attempts to optimize these costs fail:

- `Blocking progress (fundamental): Sharing requires waiting.` Clearly, when one client has exclusive access to the resource, some or all other clients must wait idly and are unable to make progress. (The worst case is when a client could starve, or be forced to wait indefinitely.)
- `Slowing progress (fundamental): Sharing demands answers to expensive questions.` Beyond actual waiting, there is usually a cost for just asking for the coordination — whether or not it is actually needed.
- `Wasting progress (secondary): Resolving contention can mean throwing away work.` Additionally, some protocols perform optimistic execution to mitigate the first two costs and perform faster in the case of no contention. But when contention actually happens, the protocol may have to undo and redo otherwise — useful work. The more contention we encounter, the more effort we waste.

	A. Software, visible in your source code	B. Software, invisible in your source code	C. Hardware, invisible in your source code
1. Blocking progress: The basic waiting requirement. When it's one's turn the others inherently must wait	Blocking calls like mutex.lock(), semaphore.wait() Opening a file for write Writer locking a RW mutex "CAS loop" in lock-free & obstruction-free algorithms	Threads sharing a CPU core: context switches Synchronized I/O (e.g., console)	Write to shared memory: exclusive ownership False sharing: two *possibly-unshared* objects in a cache line Tasks using the same spindle (e.g., HDD, DVD) Tasks accessing the same server
2. Slowing progress: Coordination / synchronization overhead. Extra cost of synchronization that we wouldn't otherwise need	mutex.unlock() Atomic variable read or write (esp. write), even in wait-free algorithms Compare-and-swap (CAS)	Threads sharing a CPU core: context switches Readers locking a RW mutex: CAS Sharing a reference counted object: CAS Inhibits optimizations	Write to shared memory: sync after write, propagate values to other caches/memory CAS: full memory sync before & after to guarantee atomicity Inhibits optimizations
3. Wasting progress: Throwing away work. Having to undo/redo	Lock-free algorithms involving a retrying "CAS loop"	Backoff/retry protocols	CAS implemented with loop Tasks sharing a cache: eviction

`Table 1`: Examples of contention penalties

These penalties also manifest in various ways, shown in the columns. The simplest case is when the potential cost is visible in our source code (column A). For example, just by looking at the code we know that the expression `mutex.lock()` might cause our thread to wait idly. But each penalty also manifests in invisible ways, both in software and in hardware (columns B and C), particularly on multicore hardware.

Visible Costs (A1, A2, A3)

Consider the following code which takes locks for synchronization to control access to a shared object `x`:

```
// Example 1: Good, if a little boring;
// plain-jane synchronized code using locks

// Thread 1
mutX.lock();           // A1, A2: potentially blocking call
Modify( x, thisWay );
mutX.unlock();         // A2: more expense for synchronization

// Thread 2
mutX.lock();           // A1, A2: potentially blocking call
Modify( x, thatWay );  // ok, no race
mutX.unlock();         // A2: more expense for synchronization

// Thread 3
mutX.lock();           // A1, A2: potentially blocking call
Modify( x, anotherWay ); // ok, no race
mutX.unlock();         // A2: more expense for synchronization
```

The obvious waiting costs (type A1) are clear: You can read them right in the source code where it says `mutX.lock()`. If these threads try to run concurrently, the locks force them to wait for each other; only one thread may execute in the critical section at a time, and the use of `x` is properly serialized.

The A1 waiting characteristics may vary. For example, if `mutX` is a reader/writer mutex, that will tend to improve concurrency among readers, and readers' lock acquisitions will succeed more often and so be cheaper on average; but acquiring a write lock will tend to block more often as writers are more likely to have to wait for readers to clear. We could also rewrite the synchronization in terms of atomic variables and compare-and-swap (CAS) operations, perhaps with a CAS loop that optimistically assumes no contention and goes back and tries again if contention happens, which is common in all but the best lock-free algorithms (lock-free and obstruction-free; only wait-free algorithms avoid A1 entirely). [2]

We also incur the performance tax of executing the synchronization operations themselves (A2). Some of these are probably obvious; for example, taking a lock on a contended mutex clearly requires spinning or performing a context switch which can be expensive. But even taking an uncontended lock on an available mutex isn't free, because it requires a synchronized memory operation such as a compare-and-swap (CAS). And, perhaps surprisingly, so does releasing a lock with `mutX.unlock()`, because *releasing* a lock requires writing a value that signifies the mutex is now free, and that write has to be synchronized and propagated across the system to other caches and cores. (We'll have more to say about memory synchronization when we get to cases C1-C3.)

Even the best wait-free algorithms can't avoid paying this toll, because wait-free algorithms are all about avoiding the waiting costs (A1); they still rely on synchronized atomic memory operations and so must pay the overhead taxes (A2). On many mainstream systems, the write to unlock a mutex, or to write or CAS an atomic variable in lock-free code, is significantly more expensive than a normal memory write for reasons we'll consider in a moment, even when there are no other contending threads on other processors that will ever care to look at the new value.

Finally, some algorithms may actually have to redo their work (A3), particularly if they perform optimistic execution. Lock-free code to change a data structure will frequently do its work assuming that no other writers are present and that its final atomic "commit" will succeed, and perform the commit using a CAS operation that lets it confirm whether the key variable still has its original expected value; if not, then the commit attempt failed and the code will need to loop to try again and attempt a CAS commit again (hence, the term "CAS loop") until it succeeds. Each time the code has to retry such a loop, it may need to partially or entirely throw away the work it did the previous time through. The more contention there is, the more often it will have to re-execute. On top of losing the work, there is often an extra overhead if abandoning the work requires that we do some cleanup (e.g., deallocation or destruction/disposal of scratch resources) or perform

compensating actions to "undo" or "roll back" the work.

Besides shared memory operations, other kinds of shared resources cause similar problems. For example, if two processes try to open a file for exclusive access with a blocking call, one must wait (A1), for the same reason one child must wait for another to finish using a shared green crayon. Opening a file for exclusive use is also usually a more expensive operation than opening it for read-only use because it incurs more coordination overhead (A2); it's the same kind of overhead as when two coffee customers are waiting for the cream, and after waiting for the previous customer to finish they have to make eye contact and maybe even talk to decide which of the waiters gets to go next.

In all these cases, the key point is that the potential blocking or overhead or retrying is clearly visible in the source code. You can point to the lines that manipulate a mutex or an atomic variable or perform CAS loops and see the costs. Unfortunately, life isn't always that sweet.

Invisible Costs in Software (B1, B2, B3)

Column B lists examples of some costs of sharing that still occur in software, but typically aren't visible by reading your own source code.

Two pieces of code may have to wait for each other because of synchronization being done under the covers by low-level libraries or by the operating system (B1). This can arise because they share a resource (not just memory): For example, two threads may try to write to the console, or append to the same file, using a synchronized I/O mode; if they try to execute their conflicting actions at the same time, the I/O library will serialize them so that one will block and wait until the other is done before proceeding. As another example, if two threads are running on the same CPU core, the operating system will let them share the core and have the illusion of executing simultaneously by making them take turns, interleaving their execution so that each periodically is preempted and waits for the other to get and use its next quantum.

These kinds of invisible sharing can also impose invisible synchronization overhead costs (B2). In the case of two threads sharing a CPU core, in addition to waiting they incur the overhead of context switches as the operating system has to save and restore the registers and other essential state of each thread; thanks to that tax, the total CPU time the threads receive adds up to less than 100 percent of the processor's capacity.

While we're speaking of B2, I have some mixed news for fans of reader/writer mutexes and reference counting. A reader/writer mutex is a common tool to achieve better concurrency for a "read-mostly" shared object. The idea is that, since readers dominate and won't interfere with each other, we should let them run concurrently with each other. This can work pretty well, but taking a read lock requires at least an expensive CAS on the shared reference count, and the CAS expense grows with the number of participating threads and processors. (Note that many reader/writer mutexes require much more overhead than just that because they maintain other lists, counters, and flags under the covers, so that they can block readers when a writer is waiting or active, allow priority entry, and maintain fairness and other properties.) [12]. The shorter the reader critical section and the more frequently that it's called, the more the synchronization overhead will begin to dominate. A similar problem arises with plain old reference counting, where the CAS operations to maintain the shared reference count itself can limit scalability. We'll consider the reasons why shared writes in general, and CAS in particular, are so expensive in more detail when talking about column C (see below).

Sharing also inhibits optimizations (still B2). When we share a resource, and notably when we share an object in memory, directly or indirectly we have to identify critical sections in the code to acquire and release the resource. As I described in detail in [2] and [3], compilers (and processors and caches, in C2) have to restrict the optimizations they can perform across those barriers, and that can result in a measurable penalty especially in inner loops.

Finally, for similar reasons as those we saw with A3, backoff algorithms and optimistic algorithms that are implemented under the covers inside libraries or by the operating system can result in invisible wasted work (B3).

That concludes our tour of software-related costs...but the story would be grossly incomplete without looking in detail at costs in hardware. Let's focus first once again on the important case of shared memory objects, then broaden the net again to remember that the principle is fully general.

Shared Objects Cause Overhead — Even In a Race (C1, C2)

Here's an interesting fact, and perhaps surprising: Writing to a shared memory object from multiple processors makes those processors wait for each other (C1) and incurs synchronization overhead (C2), even if you fail to do any locking or other synchronization — that is, *even in a race*. To see why, let's dig a little.

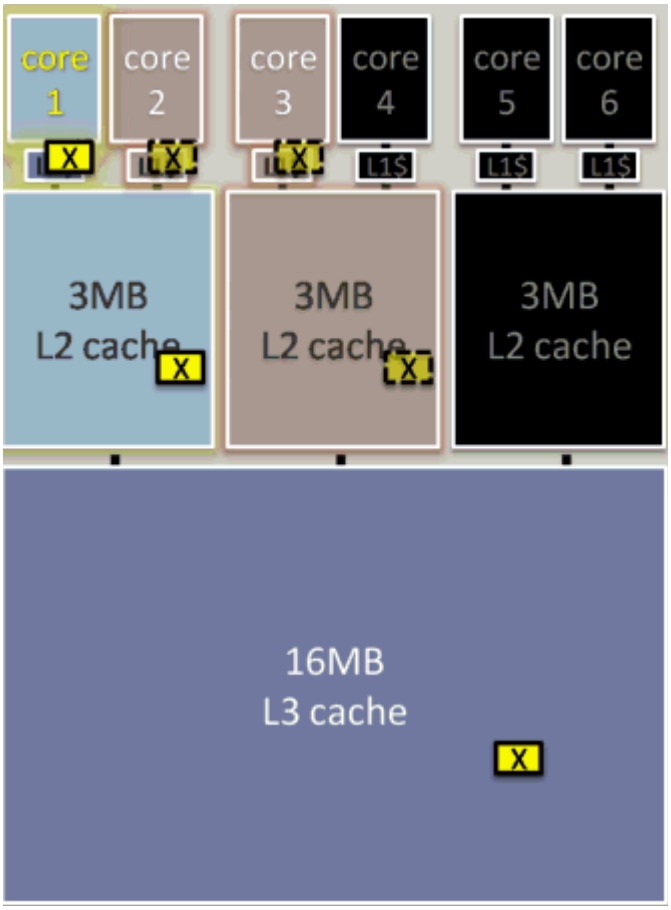
If we change the Example 1 code to remove the locking, then we have a classic race condition:

```
// Example 2: Evil code for demonstration only
// (closed course, professional driver)

// Thread 1
Modify( x, thisWay );     // race condition

// Thread 2
Modify( x, thatWay );     // race condition

// Thread 3
Modify( x, anotherWay );  // race condition
```

Take a moment to consider this question: Have we really removed all waiting (row 1) and overhead (row 2)? Is the performance of this code now going to be the same as if the threads were using only unshared local objects?

Perhaps surprisingly, on mainstream multicore systems the answer is No. We've definitely removed the obvious and visible synchronization (column A in Table 1). But on mainstream multicore systems, we are still incurring invisible costs (column C) just because we're writing to a shared object through potentially complex levels of cache. Achieving cache coherence across the system inherently requires coordination and overhead, to hide the complexity of the memory hierarchy and maintain the illusion that the data only exists in one place (its expected memory location) when it actually exists in multiple places (the actual memory location plus all the copies in various caches). [4] For more about the memory hierarchy and its costs, see also [5].

Consider how threads 1-3 might execute on the first three cores of a typical modern mainstream multicore processor: an Intel Dunnington, whose simplified block diagram is shown in Figure 1. There are six CPU cores and three levels of cache on the die; each core has 96KB of private level 1 (L1) cache; each pair of processors shares 3MB of L2 cache; and all six processors share up to 16MB of L3 cache in common. (If you already have experience with NUMA architectures, bladed servers, and such, this may look very familiar. All mainstream commodity hardware will be basically NUMA from now on.)



`Figure 1`: Using `x` on three cores of an Intel Dunnington processor; core 1 has exclusive access

When a core uses object `x`, the memory containing the accessed parts of `x` must be loaded from RAM up through the pyramid of cache leading to that core. In particular, caches manipulate memory in small contiguous chunks called cache lines, so asking for one byte requires loading the entire cache line containing that byte. In Figure 1, if cores 1, 2, and 3 request access to `x`, the hardware has to load the cache line(s) containing `x` into the processor-wide L3 cache, then into the two L2 caches for cores 1-2 and 3-4, then into the three L1 caches for cores 1, 2, and 3.

For an unshared or read-only shared memory location, it's enough to have each core load the cache line(s) containing that location all the way up into its local cache. So if `x` were immutable and all the accesses were reads, we'd be done and there would be no contention; each core could simply load a copy of the data into its cache and read it independently.

But, in our case, cores 1-3 are modifying `x`, and that's where it gets interesting because to *write* to a memory location a core must additionally have exclusive ownership of the cache line containing that location. While one core has exclusive use, all other cores trying to write the same memory location must wait and take turns — that is, they must run serially. Conceptually, it's as if each cache line were protected by a hardware mutex, where only one core can hold the hardware lock on that cache line at a time. Here, let's say core 1 gets to go first: It acquires exclusive use of the cache line, which typically includes following some procedure for invalidating that cache line in at least core 3's L2 cache and possibly also in core 2's and core 3's L1 caches; it performs its update to the appropriate parts of the cache line, then flushes the result out to L2, L3, and RAM for other cores to see; and then, when memory has been sufficiently synchronized, it releases ownership of the cache line. In the meantime, cores 2 and 3 must wait idly (C1), waiting for their turn to reload the cache line with the new value and then perform the same exclusion dance in turn. The deeper the memory hierarchy, the "farther apart" two cores can be and the greater the cost of synchronizing them (C2). [6]

It can be discouraging to see how the overheads of writing to a shared memory location add up. To avoid race conditions, our software should be using mutexes or other synchronization, which means impeding threads' progress as they wait for each other (A1 in Table 1) and incurring the overhead of software synchronization operations (A2 in Table 1). But even in a race, the mere act of sharing a writable memory location causes contention in hardware: We block other cores' and threads' progress by incurring waiting on the memory location (C1), and we incur the cost of synchronization to flush and propagate each write to cores near and far across the rest of the cache hierarchy (C2).

All Sharing Is Bad -- Even of "Unshared" Objects (C1, C2)

It's bad enough that writing to the same shared object inherently throttles scalability, but what's worse is that writing to nearby objects can have the same effect thanks to false sharing.

False sharing (aka "ping-ponging") occurs when two different objects — shared or not! — happen to live in the same cache line, and the cache system treats them as a single lump for caching purposes. I've discussed some of the perils of false sharing before, in [5] and Example 4 of [7], but it arises again here: Sadly, the C1 and C2 costs of writing to shared memory can cost you even when you're not actually writing to the same shared object.

`Figure 2`: Sample memory layout for four objects across 32-byte cache lines

Figure 2 shows a sample memory layout where each row represents a 32-byte cache line; note that objects `obj2` and `obj3`, and part of `obj4`, all reside on the same line. This can cause a real performance problem if `obj2`, `obj3`, and `obj4` are not intended to be shared together, because it is typically impossible for most mainstream multicore hardware to let two different cores simultaneously write to different parts of the same cache line without serializing those writes. For example, a core that wants to update (or in some cases even read from) `obj2` must wait for another core that happens to be concurrently writing to the first part of `obj4`. So these two threads end up competing at least enough to get these writes serialized, even if they're supposed to be able to run fully concurrently -- if, say, `obj2` and `obj4` are protected by different mutexes or, worse still, because `obj2` and `obj4` are actually supposed to be unshared private objects used only by its respective thread!

Will shuffling objects around or adding padding significantly affect performance? It sure can. Here are two examples: First, in Example 4 of [7], I demonstrated code and performance measurements where merely adding padding increased scalability of a lock-free queue on a 24-core benchmark machine from peaking at 15 cores to peaking at over 20 cores. Second, a few days before I wrote this article, the PLINQ [8] team discovered that moving a single field of a data structure to get it off a popular cache line improved the scalability of one benchmark from 5.2x to 14.8x on a 16-core machine. Incidentally, you usually won't notice such scalability differences on machines with too few cores, because first you have to get enough hardware concurrency for the scalability to matter; on a 4-core machine, the improvement from that same change was only from 3.8x to 3.9x.

The moral(s) of the story? Joe Duffy put it well:

1. Test on bigger machines wherever possible. [*After all, a "big" machine today is probably similar to a "typical" machine your customer will be running tomorrow. -hs*]
2. Validate any performance fixes or experiments on the larger machines, else you will miss

cache-related regressions and you might write off potential performance improvements that would have showed a lot of gain had you only run it on a big machine.
3. False sharing is deadly. And even more so on big machines. We knew that, but you'd be surprised how tiny a change led to the cache problems: the movement of a single field [moving a single writable field out of a cache line that otherwise contained only read-only immutable data, so that the cache line went from "read-mostly" to "read-only" and the writer thread stopped interfering with otherwise-independent reader threads]. [9]

Today's performance profiling tools are poor at helping us find these penalties. For example, in all common profiling tools and simple utilities like Windows' Task Manager, your CPU appears busy even while idly waiting for memory due to miss latency and cache line exclusion, so that even when all cores appear to be pegged at 100% the cores may easily be waiting for memory 90% of the time or more. The industry is still working on providing decent tools to let programmers detect these and similar sharing-related performance issues, particularly those due to hardware effects.

Be aware of false sharing, and "if variables A and B are…liable to be used by two different threads, keep them on separate cache lines" to avoid penalizing scalability. [5]

Other Shared Memory Costs in Hardware (C1, C2, C3)

Is a shared read less expensive than a shared write? We saw that a shared write is expensive because it needs exclusion (C1), and must be propagated across the memory subsystem and conceptually requires a synchronization after the write (C2). We might expect that reads are cheaper, because they conceptually don't require exclusivity or propagation, and in general writes are more expensive than reads. But the answer depends on the processor: On many processors, shared reads are indeed cheaper, close or equal in cost to an unshared read; on other processors, in some circumstances even a shared read must incur full hardware synchronization overhead, such as that a sequentially consistent read requires a `hwsync` instruction on current PowerPC processors. [10]

What about the cost CAS compared to a shared write? A CAS also requires exclusion (C1) and conceptually requires synchronization not only after, but also before, the operation (C2), so we would expect a CAS to be at least as expensive, and probably more expensive, than a shared write. That's usually the case, and on some processors the most efficient implementation of CAS is to generate a loop that can actually perform, not just one, but repeated heavyweight synchronizations (C3). [10] (Note that this isn't the same as the "CAS loop" lock-free coding technique I mentioned earlier, which meant writing an explicit loop in your code that performs repeated CAS operations; here I mean that the implementation of the CAS operation itself can have a loop buried inside by which it could be forced to retry and multiple heavy sync instructions each time the software tries to perform a single CAS.)

Other Invisible Costs in Hardware (C1, C2, C3)

We've focused on synchronizing memory operations, but there are other examples of costs in column C.

Consider two threads or processes sharing a spindle. If two threads or processes try to read or write files on the same hard disk drive, or read files on the same DVD-ROM, their operations will have to wait for each other (C1), much as processes do when scheduled on the same core. Additionally, every time the spindle device switches to serve a different user request, it will experience overhead for moving the (usually singular) head across the media and probably some rotational delay for the right part of the media to reach the head (C2), much as processes sharing a core experience will context switch overhead; the amount of time the clients each get access to the hardware will add up to less than 100% due to this overhead.

As mentioned under B2, sharing also inhibits optimizations at the hardware level (still C2). Having critical sections in code prevents processors from reordering instructions, and caches from performing optimizations, that would use memory and other resources more efficiently.

Finally, consider a very different kind of memory sharing: cache residency and eviction under contention (C3). Two threads or processes may share a cache in common at any level of the hierarchy; for example, in Figure 1, they may share only L3 in common (e.g., if running on cores 1 and 6), share both L3 and L2 in common (e.g., if running on cores 5 and 6), or share L3, L2, and even L1 in common (e.g. if running on the same core). Each of the two threads or processes has a given hot working set of data it needs to perform its current work. When the shared cache is big enough for one or both of the contending threads' hot data taken by itself, but not big enough for the total, then running the threads simultaneously means that memory will appear to be slower as they more often evict both their own and each other's data from the shared cache. For example, in Figure 1 and given two threads running memory-intensive inner loops each using 2MB of hot data, the threads can run on cores 1 and 6 without experiencing cache eviction interference, because 4MB fits into L3 cache (assume that the rest of the system's work fits in the remainder). If they run on cores 5 and 6, however, their demand for 4MB will contend for 3MB of L2 cache, which doesn't fit without regular eviction and reloading from L3. If they run on the same core, interleaved with context switches, then on each context switch from one thread to the other we can expect L1 cache to be completely evicted and reloaded, which is a kind of "undo/redo" C3 cost over and above the B2 cost of context switching.

Stepping Back: A Perspective on Symptoms vs. Causes

I frequently see assertions like these, which are basically correct but focus attention on symptoms rather than on the root cause:

- "Taking an exclusive lock kills scalability."
- "Compare-and-swap kills scalability."

Yes, using locks and CAS incurs waiting and overhead expenses, but those all arise from the root cause of having a mutable shared object at all. Once we have a mutable shared object, we have to have some sort of concurrency control to prevent races where two writers corrupt the shared object's bits; instead of trying to get exclusive use of each individual byte of an object, we typically employ a mutex that conveniently stands for the object(s) it protects and synchronize on that instead; and mutexes in turn are ultimately built on CAS operations. Alternatively, we might bypass mutexes and directly use CAS operations in lock-free code. Whichever road we take, we end up in the same place and ultimately have to pay the CAS tax and all the other costs of synchronizing memory. And, as we saw in column C, even in a race we'd still have real contention in hardware and degraded performance — for no other reason than that we're writing to a shared object.

I wouldn't say "locks kill scalability" or "CAS kills scalability" for the same reason I wouldn't say something like "Stop signs and traffic lights kill throughput." The underlying problem that throttles throughput is having a shared patch of pavement that requires different users to acquire exclusive access; once you have that, you need some way to control the sharing, especially in the presence of attempted concurrent access. [11] So, once again, it's *sharing* that kills scalability, the ability to increase throughput using more concurrent workers -- whether we're sharing "the file," "the object," or "the patch of pavement." Even having to ask for "the crayon" kills scalability for a family's artistic output as we get more and more little artists.

Summary

Sharing is the root cause of all contention, and it is an enemy of scalability. In all of the cases listed at the outset, the act of sharing a resource forces its users to wait idly and to incur overhead costs -- and, often

enough, somebody ends up crying. Much the same drama plays out whether it's because kids both want the one blue crayon at the same time, processes steal a shared CPU's cycles or evict each other's data from a shared cache, or threads possibly retry work and are forced to ping-pong the queue's memory as only one processor can have exclusive access at a time. Some of the costs are visible in our high-level code; others are invisible and hidden in lower layers of our software or hardware, though equally significant.

Prefer isolation: Make resources private and unshared, where possible; sometimes duplicating resources is the answer, like providing an extra crayon or a copy of an object or an additional core. Otherwise, prefer immutability: Make shared resources immutable, where possible, so that no concurrency control is required and no contention arises. Finally, use mutable shared state when you can't avoid it, but understand that it's fundamentally an enemy of scalability and minimize touching it in performance-sensitive code including inner loops. Avoid false sharing by making sure that objects that should be usable concurrently by different threads stay on different cache lines.

On Deck

The fact that sharing is the root cause of contention and an enemy of scalability adds impetus to pursuing isolation. Next month, we'll look at the question of isolation in more detail as we turn to the topic of the correct use of threads (hint: threads should try hard to communicate only through messages). Stay tuned.

Acknowledgments

Thanks to Joe Duffy for his comments and data on this topic.

Notes

[1] For details on wait-free vs. lock-free vs. obstruction-free algorithms, see: M. Herlihy. "Obstruction-Free Synchronization: Double-Ended Queues as an Example" (Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003). Available at http://www.cs.brown.edu/~mph/HerlihyLM03/main.pdf.

[2] H. Sutter. "Use Critical Sections (Preferably Locks) to Eliminate Races" (Dr. Dobb's Journal, 32(10), October 2007). Available online at http://www.ddj.com/cpp/201804238.

[3] H. Sutter. "Apply Critical Sections Consistently" (Dr. Dobb's Journal, 32(11), November 2007). Available online at http://www.ddj.com/hpc-high-performance-computing/202401098.

[4] For simplicity I'm going to talk about the way caches tend to operate on mainstream desktop and server hardware, but there's a lot of variety out there.

[5] H. Sutter. "Maximize Locality, Minimize Contention." (Dr. Dobb's Journal, 33(6), June 2008.) Available online at http://www.ddj.com/architect/208200273.

[6] For added fun, imagine the synchronization involved in adding two more levels to the memory hierarchy: having a server containing some system-wide RAM and several "blades," where each blade contains two or four Dunnington chips and additional cache or RAM that is local to the processors on that blade.

[7] H. Sutter "Measuring Parallel Performance: Optimizing a Concurrent Queue" (Dr. Dobb's Journal, 34(1), January 2009). Available online at http://www.ddj.com/cpp/211800538.

[8] J. Duffy and E. Essey. "Parallel LINQ: Running Queries On Multi-Core Processors" (MSDN Magazine, October 2007).

[9] Joe Duffy, personal communication.

[10] P. McKenney and R. Silvera. "Example POWER Implementation for C/C++ Memory Model" (ISO/IEC C++ committee paper JTC1/SC22/WG21/N2745, August 2008). Available online at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2745.html.

[11] I find it useful to compare a mutable shared object to the patch of pavement in an intersection. You don't want timing "races" because they result in program crashes. Sharing requires control: Locks are like stop signs or traffic lights. Wait-free code is often like building a cloverleaf -- it can eliminate some contention by removing direct sharing (sometimes by duplicating resources), at the cost of more elaborate architecture and more space and pavement (that you won't always have room for), and paying very careful attention to getting the timing right as you automatically merge the cars on and off the freeway (which can be brittle and have occasional disastrous consequences if you don't get it right).

[12] As this article was going to press, Joe Duffy posted a nice overview of some of the issues of reader/writer mutexes: J. Duffy. "Reader/writer locks and their (lack of) applicability to fine-grained synchronization" (Blog post, February 2009). Available online at http://www.bluebytesoftware.com/blog/default,date,2009-02-11.aspx.

Herb is a bestselling author and consultant on software development topics, and a software architect at Microsoft. He can be contacted at www.gotw.ca.