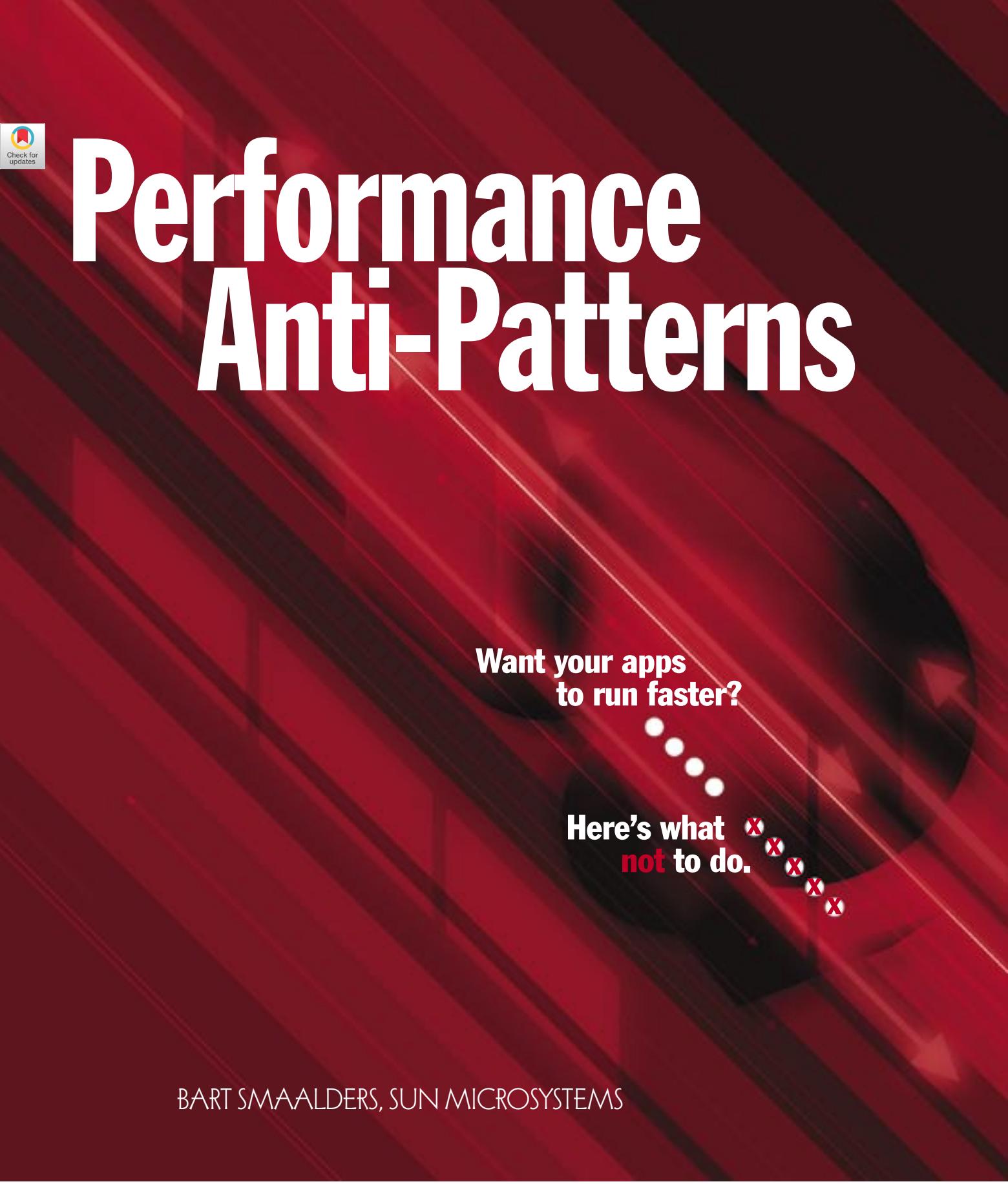
Performance Anti-Patterns



BART SMAALDERS, SUN MICROSYSTEMS

Performance pathologies can be found in almost any software, from user to kernel, applications, drivers, etc. At Sun we've spent the last several years applying state-of-the-art tools to a Unix kernel, system libraries, and user applications, and have found that many apparently disparate performance problems in fact have the same underlying causes. Since software patterns are considered *abstractions of positive experience*, we can talk about the various approaches that led to these performance problems as *anti-patterns*—something to be avoided rather than emulated.

Some of these anti-patterns have their roots in hardware issues, some are the result of poor development or management practices, and some are just common mistakes—but we've seen them all repeatedly. In this article we discuss these mistakes: what causes them, how to find them, and how best to avoid them.

FIXING PERFORMANCE AT THE END OF THE PROJECT

Software development is often a resource-constrained problem. Rarely do project or product teams have all the work years they might want. Unfortunately, an area that often receives short shrift is measuring and evaluating performance. Far too often, project teams race to the end of the schedule developing new features and fixing bugs, and performance work is left as an afterthought. They often fail to formulate performance goals or benchmarks, and the first time the developers even consider performance is after reports of performance problems are received from their beta test sites. At this point in a project, we have often joked about getting out the "perf spray," hoping we could quickly spray on some performance as if it were some sort of flashy paint.

Performance Anti-Patterns

This anti-pattern seems obvious, but many projects have rediscovered this the hard way. If your team doesn't bother to model or measure software performance or waits until near the end of the project to begin, it's unlikely to get good results except by happy accident.

MEASURING AND COMPARING THE WRONG THINGS

Selecting a benchmark and comparing the results seems, initially at least, to be a simple problem, but a host of mistakes are made during this process. For some releases of Solaris, for example, the performance metric was to be no more than 2 percent slower than the previous release on a set of large system benchmarks. This was a mistake, akin to exercising just hard enough to gain only two pounds per year. It practically ensured that Solaris performance would decrease over time.

This goal also ignored competitive realities, as no attempt was made to compare the performance with other operating systems. In addition, the use of large benchmarks meant that when a regression in performance occurred, the dearth of test rigs ensured that analyzing the performance regressions would hamper additional testing.

What and how should we be measuring? A good benchmark is:

- **Repeatable,** so experiments of comparison can be conducted relatively easily and with a reasonable degree of precision.
- **Observable,** so if poor performance is seen, the developer has a place to start looking. Nothing is more frustrating than a complex benchmark that delivers a single number, leaving the developer with no additional information as to where the problem might lie.
- **Portable,** so that comparisons are possible with your main competitors (even if they are your own previous releases). Maintaining a history of the performance of previous releases is a valuable aid to understanding your own development process.
- **Easily presented,** so that everyone can understand the comparisons in a brief presentation.
- **Realistic,** so that measurements reflect customer-experienced realities.
- **Runnable,** so that all developers can quickly ascertain the effects of their changes. If it takes days to get performance results, it won't happen very often.

Not all benchmarks selected will meet all of these criteria, but it's important that some of them do. Make sure to select enough benchmarks so that important parts of your product's performance envelope aren't a surprise when it ships—and avoid selecting benchmarks that don't really represent your customer, because your team will end up optimizing for the wrong behavior. Resist the temptation to optimize for the benchmark; the recent discovery that some operating systems have "improved system call performance" by moving the `getpid(2)` system call into user-land is a perfect example; no real application calls `getpid(2)` enough to matter.

Selecting a benchmark is asking for that aspect of performance to be optimized, probably at the expense of other aspects that are not being measured. If as an operating system developer you want faster system calls, design a benchmark that is a weighted average of the calls your customers' applications make most frequently. Be careful what you ask for, because you're likely to get it.

ALGORITHMIC ANTIPATHY

For many software developers, algorithms are something they studied back during their college days, and thankfully not something with a lot of relevance to their day jobs. During Solaris 10 development, Solaris engineers fixed a long list of performance problems across the kernel and user libraries. Toward the end of the release, we spent some time reviewing just what had been improved and by how much—and what was the underlying cause of the performance problem. Interestingly enough, all the really big improvements (above, say, 200 percent) resulted from changes in algorithms. Over and over again, all the other performance fixes—using specialized SIMD processor instructions such as SSE2 or VIS, inserting memory prefetch instructions, cycle shaving—paled in significance compared with simply going back and rethinking the locking algorithms and/or data structures.

A key part of algorithm selection is having a realistic benchmark or workload in hand to support making decisions based on actual results rather than intuition or folklore. This means the most effective time to do performance and scalability work is in the earlier phases of the project, perhaps the exact opposite of what usually happens. All the clever compilation options are pretty useless when dealing with O(n2) algorithms for large values of *n*. Poor algorithms are the number 1 (and probably numbers 2 and 3 as well) cause of poor software system performance.

REUSING SOFTWARE

Who would be so incautious as to code an O(n2) algorithm? Well, for small values of *n*, the difference may not matter, and the simplistic algorithm might indeed be faster. But a perfectly reasonable value of *n* in 1992 might look rather absurd in 2005, and for long-lived products or source bases such as operating systems, this is a real concern. For example, the basic design of the Solaris VM system comes from SunOS 4.0, which was in development in 1985. Back then, a well-configured machine had perhaps 1,000 pages of real memory. Twenty years later, a well-configured desktop machine may have several million pages, and large servers, far more.

Yes, we've made extensive changes to compensate for this, but were we to rewrite the VM system from scratch, we would make significantly different choices in light of today's conditions. Other examples of the same problem include databases where the indices are no longer aligned with the prevailing queries, open hash tables undersized for current problems, hash functions that don't work well anymore with changes in input, etc.

One of the most valuable methods of catching these sorts of problems is for the original developer both to document the assumptions about the externalities affecting the code and to provide some sort of programmatic means of validating those assumptions later on. One useful technique, for example, is to keep track of maximum hash chain lengths on an open hash, as well as the total hash table population; this allows for the easy identification of both undersize (or the need for self-resizing) hash tables and poor hash functions. Another technique is to force an error when assumptions are violated; this causes a hard failure with, hopefully, a clear error rather than a mysterious slowdown of unknown cause; this of course may not be appropriate for some applications. Software reuse is a fine goal, but beware of violating the assumptions made during its development.

ITERATING BECAUSE THAT'S WHAT COMPUTERS DO WELL

During our efforts to tune OpenWindows performance in the early 1990s, we found that engineering users frequently complained about scrolling performance on their X desktops. After examining the shell, terminal emulator, and X server performance, we found that an amazing amount of work was involved just to scroll a single line of text to the screen. This work was nicely distributed in a variety of different places, none of them apparently subject to the quick fix we performance engineers were, of course, looking for at the tail end of the project. Our discussions led us to the idea of avoiding scrolling the screen as much as possible, so we started searching for ways to avoid doing this while still preserving the existing semantics and user experience. After some thought, we introduced a buffering streams module that would be plumbed into the pseudo-terminal. This would coalesce multiple small writes from the user's application into a single large block readable at once by the terminal emulator.

The terminal emulator was recoded to take the entire block of text from the application and place it in the window scroll buffer and then jump to the end of the buffer. This led to a more than tenfold improvement in scrolling performance—and made the system feel much faster even under load, since we were doing so much less work. The point of this anecdote is obvious: If your application is doing unneeded or unappreciated work—repainting the screen multiple times, computing statistics too frequently, etc.—then eliminating such waste is a lucrative area for performance work. What often matters for applications is the end state of the program, not the exact series of steps used to get there. Often a shortcut is available that will allow us to reach the goal more quickly. It's like shortening the race course rather than speeding up the car: With the exception of correctly used memory prefetch instructions, the only way to go faster in software is to do less.

PREMATURE OPTIMIZATION

Frequently during performance audits we find software that appears to be carefully tuned and optimized, only to discover that the hand-unrolled loops, register declarations, inline functions (sometimes written in assembly language), and other apparent artifacts of a protracted tuning effort are instead *performance decorations*. They make the software look fast but have no positive impact on actual delivered performance, not unlike a pair of chromed valve covers on a car. These optimizations are often done during initial development by engineers just coming off the crash tuning phase of the previous project.

Premature optimization indeed often adversely affects performance on real benchmarks, either by increasing the instruction cache footprint enough to cause misses and pipeline stalls, confusing the register allocator in the compiler, or sometimes by discouraging other engineers from carefully looking for the actual source of the performance problems. Low-level cycle shaving has its place—but only at the end of the performance effort, not during initial code development. Even then, the conditions under which the tuning is done should be carefully documented to help others later evaluate whether those conditions are still valid.

Performance Anti-Patterns

As noted earlier, the most effective software performance work focuses on algorithms, not low-level details; there is nothing sillier than a hand-coded assembler linked list instead of a simple hash table coded in C. Another reason such low-level optimization isn't a good idea is that it tends to be platform-centric: For software that needs to run well on a diverse set of systems or processors, these techniques often require separate versions of these routines for each platform—a painful and expensive approach from a development, portability, and test perspective. Wait on such efforts until you're sure (based on the result of actual experiments, not intuition) that this will be a cost-effective way of increasing performance. Donald Knuth said it: "Premature optimization is the root of all evil."

FOCUSING ON WHAT YOU CAN SEE RATHER THAN ON THE PROBLEM

Those of us working on Solaris performance are frequently asked to determine the causes of an application's poor performance, often with the thought that there must be some underlying operating system defect causing the problem. Although this sometimes turns out to be an issue, in the vast majority of cases the problem actually turns out to be in the application itself—and then, often in the way the customer is using the application.

Thinking about the nature of applications programming makes the reason clear. Each line of code at the top level of the application causes, in general, large amounts of work elsewhere farther down in the software stack. As a result, inefficiencies at the top layer have a large multiplier magnifying their impact, making the top of the stack a good place to look for possible speed-ups. Traditionally, however, the dearth of suitable tools for observing the behavior of applications has led performance engineers to attempt to use low-level tools such as truss, strace, or the various performance monitoring commands such as iostat to diagnose performance problems; this has often led to attempts to speed up common operating system calls or I/O operations rather than modifying applications to reduce the number of calls being made. For example, if you're faced with a database application performance problem, look at SQL first; once that has been tuned, the database properly indexed, etc., then perhaps it's time to look at disk utilization.

Fortunately, the advent of high-level dynamic instrumentation tools such as DTrace in Solaris 10 has made observation of high-level application behavior and attribution of the resultant effects on the system much easier; this should help system performance engineers focus on the real causes of poor performance.

SOFTWARE LAYERING

Many software developers become fond of using layering to provide various levels of abstraction in their software. While layering is useful to some extent, its incautious use significantly increases the stack data cache footprint, TLB (translation look-aside buffer) misses, and function call overhead. Furthermore, the data hiding often forces either the addition of too many arguments to function calls or the creation of new structures to hold sets of arguments. Once there are multiple users of a particular layer, modifications become more difficult and the performance trade-offs accumulate over time. A classic example of this problem is a portable application such as Mozilla using various window system toolkits; the various abstraction layers in both the application and the toolkits lead to rather spectacularly deep call stacks with even minor exercising of functionality. While this does produce a portable application, the performance implications are significant; this tension between abstraction and implementation efficiencies forces us to reevaluate our implementations periodically. In general, layers are for cakes, not for software.

EXCESSIVE NUMBERS OF THREADS

Once programmers become familiar with threads (or worse, multiple cooperating processes), one of the more common mistakes is deciding to use a thread (or process) per connection or other unit of pending work. This is admittedly a simple programming model with the per-connection/task state being conveniently kept on the thread stacks. This works well in small or LAN test environments, when connections drain quickly and a small number of threads is sufficient to completely load the machine. Once this application is deployed against thousands of slow connections, the programmers discover that their pride and joy requires thousands of threads to use the hardware effectively, and these thousands of threads don't really perform all that well since the machine now has a lot of TLB and cache pressure from all those stacks.

The right answer is to limit the number of threads to a more reasonable number (near the number of CPUs)

and to use a work-pile model and asynchronous I/O to multiplex the worker threads against the tasks to be done. These sorts of application architectures scale much more easily and behave much more gracefully under heavy loads. After all, if the application's net throughput begins to drop over a certain load level, the situation is inherently unstable—not a good place to be.

ASYMMETRIC HARDWARE UTILIZATION

Modern CPUs are much, much faster than the memory systems connected to them. Some processor designs use three levels of caches to hide the latency of memory accesses, and multilevel TLBs are now becoming common as well. These caches and TLBs use varying degrees of associativity, or ways, to spread the application load across the caches, but this technique is often accidentally thwarted by other performance optimizations.

One simple example is a performance tool that reorders functions in a shared library to place the most commonly used functions at the beginning of the library, an apparently reasonable strategy to reduce ITLB (instruction translation look-aside buffer) miss rates and paging. When applied to many shared libraries, however, this results in the beginning of each shared library's text segment being much more frequently accessed than other portions; since the SPARC 32-bit ABI (application binary interface) requires 64KB alignment of executable text sections, this means that those TLB entries falling on 64KB boundaries are much more commonly used, effectively reducing the size of the ITLB by a factor of 8. The same kind of effect has been seen when large numbers of identical processes use large pages for their heap or stack to avoid excessive TLB pressure: With newer CPUs having page sizes larger than their L3 caches, the page coloring algorithms that prevented hot-spotting from occurring with small pages are no longer effective, and some lines in the caches become very overused.

Another example of this occurred with a database vendor whose code allocated large chunks of shared memory (allocated with large pages) in a multiple of the L2 cache size; by having similar access patterns in each of the large chunks, the vendor reduced the hit rate of the L2 cache significantly. Hot-spotting can also occur with physical memory; allocation behavior on NUMA (nonuniform memory access) machines that favor one area of memory (and thus a subset of the available memory controllers) over another can cause noticeable increases in average memory access times.

Detecting and avoiding this hot-spotting problem can be difficult, since hardware counters and tools are often lacking in the ability to allow direct observation of these effects; once detected, avoiding such hot-spotting can be difficult without application-visible effects such as skewing heap or stack addresses. We've had some success in the past modeling various cache and TLB configurations with simple programs and playing the program's access patterns against the models; more work is definitely needed in this space, given that this is such a common performance inhibitor in otherwise well-tuned applications. Where possible, constructing simple graphs (either directly measured or modeled) of relative cache-line and TLB entry miss frequencies will almost instantly point out any fruitful areas for further examination. Often the deliberate injection of randomness into allocation patterns, memory layout, etc. is needed to prevent hot-spotting; for example, we've randomized the allocation patterns of physical memory across memory controllers for large shared memory segments in the Solaris kernel to avoid the static patterns that always seem to interfere with at least one important application's performance.

NOT OPTIMIZING FOR THE COMMON CASE

When designing locking algorithms, one can take into account significant asymmetries in the usage patterns of the access routines. The frequent operations can occur several orders of magnitude more often than the infrequent ones; designing locking algorithms to take advantage of this asymmetry can yield significant benefits. One simple example is the use of per-bucket hash table locks; this improves scalability for simple searches, inserts, or deletes into the table, while penalizing operations that require access to the entire table, such as resizes. A key part of algorithm selection is having a realistic benchmark in hand to support decisions based on actual results rather than intuition or folklore.

A more complex example is the method used to lock the CPU list in the Solaris kernel. This list of CPU structures is frequently traversed by different CPUs attempting to make scheduling decisions. Since Solaris supports both taking CPUs on- and offline and (on capable hardware) adding new CPUs or removing others, the CPU list must be modifiable when (rarely) needed, yet be efficiently traversable by many CPUs at the same time. The current locking implementation for the CPU list leverages the vast preponderance of read accesses versus write by forcing any CPU that wishes to modify the list to cause all other CPUs to run a special pause thread; thus, a thread wishing to traverse the list safely need only prevent its own preemption. This requires only a local (nonatomic) memory reference.

Performance Anti-Patterns

This ensures fast, scalable locking of the CPU list for reading, but makes modifications many orders of magnitude more expensive and completely unscalable, a reasonable trade-off given the relative frequencies of the two operations.

NEEDLESS SWAPPING OF CACHE LINES BETWEEN CPUS

As noted earlier, system designers use caches to hide memory latencies from the CPUs. On multiprocessors, carefully designed hardware protocols ensure that only one cache in a system contains a modified version of memory; multiple caches may contain unmodified copies of memory. When one CPU attempts to write memory currently in another CPU's cache, a cache-to-cache transfer takes place to move ownership of that cache line (typically 64 bytes) between CPUs. On large multiprocessors, this can take a significant amount of time and available bandwidth; minimizing the amount of these transfers measurably improves scalability. The cause of these transfers is often simple to understand: a single counter in an infrequently accessed code path that is incremented by each thread (and CPU) that traverses that routine. Should that code path become more frequently used, exchanging the cache line containing that counter may become the limiting factor in application scalability.

Some examples are a little more difficult to spot: An array of integer counters indexed by CPU ID is a classic example of *false sharing*, so named because the CPUs don't appear to be sharing the counters; the problem is that the granularity of memory ownership is 64 bytes, forcing 16 CPUs to collide on the same cache line. In software they don't share the data—but they do in hardware. A subtler version of the same problem arises from using a general-purpose allocator such as `malloc(3C)` to allocate eight-byte chunks of memory for different threads; multiple calls to `malloc` could easily return different blocks residing on the same cache line.

Another cause of needless cache-line swapping is the placebo lock; this makes the programmer feel better but doesn't really do anything. A frequently seen example is a simple counter that is protected by a lock for reading. The lock is acquired, the counter read, and the lock dropped. On all modern hardware running Solaris, 32-bit reads are atomic: The lock does nothing except reduce scalability needlessly.

A subtler locking problem that often impacts scalability is the misuse of reader-writer locks. At first blush, reader-writer locks appear to improve scalability significantly in the case of frequent readers and infrequent writers—but looking closely at how these locking primitives are implemented shows us that we grab and release a simple lock both upon getting the reader-writer lock and upon releasing it. Thus, we do twice as many atomic operations. If the hold time of the reader lock is short, as is typically the case, we would do better just using a simple mutex instead of a reader-writer lock. Reader-writer locks make sense when the read lock is held for a long time; the cost of the added atomic operations is overcome by the scalability improvements afforded by having multiple readers in the critical section at the same time.

Detection of excessive cache-to-cache transfers or false sharing can be difficult. One useful, reasonably portable technique is to look at statements that both reference memory and rank prominently in their execution time profile; this works well if most (unshared) loads don't miss the caches. This area remains a largely unexplored opportunity for developer tools; careful integration with both the compiler and runtime data collection will be required to solve this properly.

AVOIDING ANTI-PATTERNS

This list of performance anti-patterns is by no means complete; however, being familiar with those issues that have made our work more challenging should help others avoid them—or at least recognize them more quickly. Although not all of our projects may have the performance resources we might like, avoiding these anti-patterns will make even those limited resources that much more effective. Remember, the performance work done at the beginning of the project in terms of benchmark, algorithm, and data-structure selection will pay tremendous dividends later on—enough, perhaps, to allow you to avoid that traditional performance fire drill at the end. Q

LOVE IT, HATE IT? LET US KNOW

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

BART SMAALDERS (http://blogs.sun.com/barts.) works on improving Solaris performance as part of the kernel group at Sun Microsystems. He has worked on OpenWindows, CDE, and SunCluster products. He is co-author of *Programming with Threads* with Steve Kleiman and Devang Shah.