DOI:10.1145/1831407.1831422

Article development led by acmqueue
queue.acm.org

More important principles to keep in mind when designing high-performance software.

BY CARY MILLSAP

Thinking Clearly About Performance, Part 2

IN PART 1 of this article (*Communications*, Sept. 2010, p. 55), I covered some of the fundamentals of performance. *Performance* is a relation between a *task* and the *time* it consumes. That relation is measurable either as *throughput* or *response time*. Because users feel variance in performance more than they feel the mean, it's good to express performance goals in a *percentile* format, such as "Task *T* must have response time of *R* seconds or less in *P* proportion or more of executions." To diagnose a performance problem, you need to state your goals objectively, in terms of either throughput or response time, or both.

A *sequence diagram* is a helpful graphical tool for understanding how a task's execution consumes your time. A *profile* is a table that shows details about response time for a single task execution. With a profile, you can learn exactly how much improvement to expect for a proposed investment, but only if you understand the pitfalls of making incorrect assumptions about *skew*.

Minimizing Risk. As mentioned in Part 1, the risk that repairing the performance of one task can damage the performance of another reminds me of something that happened to me once in Denmark. It's a quick story:

Scene: The kitchen table in Måløv, Denmark; *the* oak table, in fact, of Oak Table Network fame, a network of Oracle practitioners who believe in using scientific methods to improve the development and administration of Oracle-based systems.[8] Roughly 10 people sit around the table, working on their laptops and conducting various conversations.

Cary: Guys, I'm burning up. Would you mind if I opened the window for a little bit to let some cold air in?

Carel-Jan: Why don't you just take off your heavy sweater?

The End.

There is a general principle at work here that humans who optimize know: when everyone is happy except for you, make sure your local stuff is in order before you go messing around with the global stuff that affects everyone else, too.

This principle is why I flinch whenever someone proposes to change a system's Oracle SQL*Net packet size when the problem is really a couple of poorly written Java programs that make unnecessarily many database calls (and, hence, unnecessarily many network I/O calls as well). If everybody is getting along fine except for the user of one or two programs, then the safest solution to the problem is a change whose scope is localized to just those one or two programs.

Efficiency. Even if everyone on the entire system is suffering, you should still focus first on the program that the business needs fixed. The way to begin is to ensure the program is working as efficiently as it can. *Efficiency* is the inverse of how much of a task execution's total service time can be eliminated without adding capacity and without sacrificing required business function.

In other words, efficiency is an inverse measure of waste. Here are some examples of waste that commonly occur in the database application world:

- A middle-tier program creates a distinct SQL statement for every row it inserts into the database. It executes 10,000 database `prepare` calls (and thus 10,000 network I/O calls) when it could have accomplished the job with one `prepare` call (and thus 9,999 fewer network I/O calls).
- A middle-tier program makes 100 database `fetch` calls (and thus 100 network I/O calls) to fetch 994 rows. It could have fetched 994 rows in 10 `fetch` calls (and thus 90 fewer network I/O calls).
- A SQL statement (my choice of article adjective here is a dead giveaway that I was introduced to SQL within the Oracle community) touches the database buffer cache 7,428,322 times to return a 698-row result set. An extra filter predicate could have returned the seven rows that the end user really wanted to see, with only 52 touches upon the database buffer cache.

Certainly, if a system has some global problem that creates inefficiency for broad groups of tasks across the system (for example, ill-conceived index, badly set parameter, poorly configured hardware), then you should fix it. Do not tune a system to accommodate programs that are inefficient, however. (Admittedly, sometimes you need a tourniquet to keep from bleeding to death, but do not use a stopgap measure as a permanent solution. Address the inefficiency.) There is a great deal more leverage in curing the program inefficiencies themselves. Even if the programs are commercial off-the-shelf applications, it will benefit you more in the long run to work with your software vendor to make your programs efficient than it will to try to optimize your system to run with an inherently inefficient workload.

Improvements that make your program more efficient can produce tremendous benefits for everyone on the system. It is easy to see how top-line reduction of waste helps the response time of the task being repaired. What many people do not understand as well is that making one program more efficient creates a side effect of performance improvement for other programs on the system that have no apparent relation to the program being repaired. It happens because of the influence of *load* upon the system.

Load is competition for a resource induced by concurrent task executions. It is the reason the performance testing done by software developers does not catch all the performance problems that show up later in production.

One measure of load is *utilization*, which is resource usage divided by resource capacity for a specified time interval. As utilization for a resource goes up, so does the response time a user will experience when requesting service from that resource. Anyone who has ridden in an automobile in a big city during rush hour has experienced this phenomenon. When the traffic is heavily congested, you have to wait longer at the tollbooth.

The software you use does not actually "go slower" as your car does when you are going 30mph in heavy traffic instead of 60mph on the open road. Computer software always goes the same speed no matter what (a constant number of instructions per clock cycle), but certainly response time degrades as resources on your system get busier.

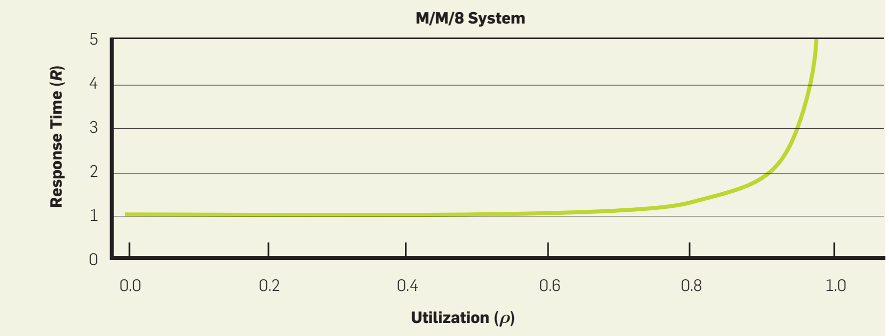
There are two reasons that systems get slower as load increases: *queuing delay* and *coherency delay*.

Queuing delay. The mathematical relationship between load and response time is well known. One mathematical model, called M/M/*m*, relates response time to load in systems that meet one particularly useful set of specific requirements.[7] One of the assumptions of M/M/*m* is the system you are modeling has "theoretically perfect scalability." This is akin to having a physical system with "no friction," an assumption that so many problems in introductory physics courses invoke.

Regardless of some overreaching assumptions such as the one about perfect scalability, M/M/*m* has a lot to teach us about performance. Figure 1 shows the relationship between response time and load using M/M/*m*.

In the figure, you can see mathematically what you feel when you use a system under different load conditions. At low load, your response time is essentially the same as your response time at

Figure 1. This curve relates response time as a function of utilization for an M/M/*m* system with *m* = 8 service channels.



no load. As load ramps up, you sense a slight, gradual degradation in response time. That gradual degradation does not really do much harm, but as load continues to ramp up, response time begins to degrade in a manner that's neither slight nor gradual. Rather, the degradation becomes quite unpleasant and, in fact, hyperbolic.

Response time (R), in the perfect scalability M/M/*m* model, consists of two components: *service time (S)* and *queuing delay (Q)*, or $R = S + Q$. Service time is the duration that a task spends consuming a given resource, measured in time per task execution, as in *seconds per click*. Queuing delay is the time that a task spends enqueued at a given resource, awaiting its opportunity to consume that resource. Queuing delay is also measured in time per task execution (for example, seconds per click).

In other words, when you order lunch at Taco Tico, your response time *(R)* for getting your order is the queuing delay time *(Q)* that you spend in front of the counter waiting for someone to take your order, plus the service time *(S)* you spend waiting for your order to hit your hands once you begin talking to the order clerk. Queuing delay is the difference between your response time for a given task and the response time for that same task on an otherwise unloaded system (don't forget our *perfect scalability* assumption).

The Knee

When it comes to performance, you want two things from a system:

▸ The best response time you can get: you do not want to have to wait too long for tasks to get done.

▸ The best throughput you can get: you want to be able to cram as much load as you possibly can onto the system so that as many people as possible can run their tasks at the same time.

Unfortunately, these two goals are contradictory. Optimizing to the first goal requires you to minimize the load on your system; optimizing to the second goal requires you to maximize it. You can not do both simultaneously. Somewhere in between—at some load level (that is, at some utilization value)—is the optimal load for the system.

The utilization value at which this optimal balance occurs is called the *knee*. This is the point at which throughput is maximized with minimal negative impact to response times. (I am engaged in an ongoing debate about whether it is appropriate to use the term *knee* in this context. For the time being, I shall continue to use it; see the sidebar for details.) Mathematically, the knee is the utilization value at which response time divided by utilization is at its minimum. One nice property of the knee is it occurs at the utilization value where a line through the origin is tangent to the response-time curve. On a carefully produced M/M/*m* graph, you can locate the knee quite nicely with just a straight-edge, as shown in Figure 2.

Table 1. M/M/*m* knee values for common values of *m*.

Service channel count	Knee utilization
1	50%
2	57%
4	66%
8	74%
16	81%
32	86%
64	89%
128	92%

Another nice property of the M/M/*m* knee is that you need to know the value of only one parameter to compute it. That parameter is the number of parallel, homogeneous, independent *service channels*. A service channel is a resource that shares a single queue with other identical resources, such as a booth in a toll plaza or a CPU in an SMP (symmetric multiprocessing) computer.

The italicized lowercase *m* in the term M/M/*m* is the number of service channels in the system being modeled. The M/M/*m* knee value for an arbitrary system is difficult to calculate, but I have done it in Table 1, which shows the knee values for some common service channel counts. (By this point, you may be wondering what the other two Ms stand for in the M/M/*m* queuing model name. They relate to assumptions about the randomness of the timing of your incoming requests and the randomness of your service times. See http://en.wikipedia.org/wiki/Kendall%27s_notation for more information, or *Optimizing Oracle Performance*[7] for even more.)

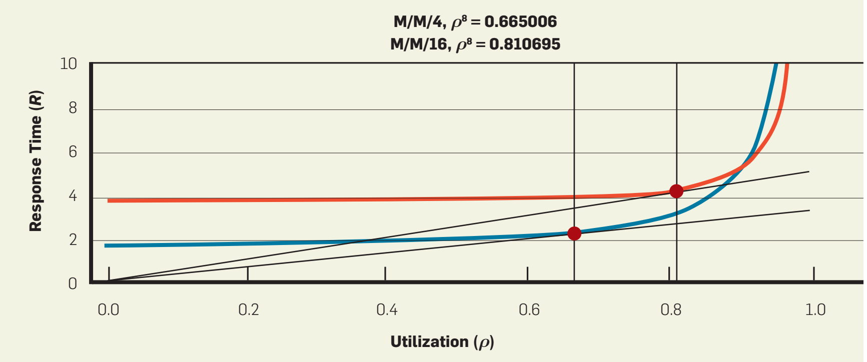
Why is the knee value so important? For systems with randomly timed service requests, allowing sustained resource loads in excess of the knee value results in response times and throughputs that will fluctuate severely with microscopic changes in load. Hence, on systems with random request arrivals, it is vital to manage load so that it will not exceed the knee value.

Relevance of the Knee

How important can this knee concept be, really? After all, as I've told you, the M/M/*m* model assumes this ridiculously utopian idea that the system you are thinking about scales perfectly. I know what you are thinking: it doesn't.

What M/M/*m* does give us is the knowledge that even if your system did scale perfectly, you would still be stricken with massive performance problems once your average load exceeded the knee values in Table 1. Your system

Figure 2. The knee occurs at the utilization at which a line through the origin is tangent to the response time curve.



Open Debate About Knees

In this article, I write about knees in performance curves, their relevance, and their application. Whether it is even worthwhile to try to define the concept of *knee*, however, has been the subject of debate going back at least 20 years.

There is significant historical basis to the idea that the thing I have described as a knee in fact is not really meaningful. In 1988, Stephen Samson argued that, at least for M/M/1 queuing systems, no "knee" appears in the performance curve. "The choice of a guideline number is not easy, but the rule-of-thumb makers go right on. In most cases there is not a knee, no matter how much we wish to find one," wrote Samson.[3]

The whole problem reminds me, as I wrote in 1999,[2] of the parable of the frog and the boiling water. The story says that if you drop a frog into a pan of boiling water, he will escape. But if you put a frog into a pan of cool water and slowly heat it, then the frog will sit patiently in place until he is boiled to death.

With utilization, just as with boiling water, there is clearly a "death zone," a range of values in which you can't afford to run a system with random arrivals. But where is the border of the death zone? If you are trying to implement a procedural approach to managing utilization, you need to know.

My friend Neil Gunther (see http://en.wikipedia.org/wiki/Neil_J._Gunther for more information about Neil) has debated with me privately that, first, the term *knee* is completely the wrong word to use here, in the absence of a functional discontinuity. Second, he asserts that the boundary value of .5 for an M/M/1 system is wastefully low, that you ought to be able to run such a system successfully at a much higher utilization value than that. Finally, he argues that any such special utilization value should be defined expressly as the utilization value beyond which your average response time exceeds your tolerance for average response time (Figure A). Thus, Gunther argues that any useful not-to-exceed utilization value is derivable only from inquiries about human preferences, not from mathematics. (See http://www.cmg.org/measureit/issues/mit62/m_62_15.html for more information about his argument.)

The problem I see with this argument is illustrated in Figure B. Imagine that your tolerance for average response time is *T*, which creates a maximum tolerated utilization value of ρT. Notice that even a tiny fluctuation in average utilization near ρT will result in a huge fluctuation in average response time. I believe that your customers feel the variance, not the mean. Perhaps they *say* they will accept average response times up to *T*, but humans will not be tolerant of performance on a system when a 1% change in average utilization over a one-minute period results in, say, a tenfold increase in average response time over that period.

I do understand the perspective that the knee values I've listed in this article are below the utilization values that many people feel safe in exceeding, especially for lower-order systems such as M/M/1. It is important, however, to avoid running resources at average utilization values where small fluctuations in utilization yield too-large fluctuations in response time.

Having said that, I do not yet have a good definition for a *too-large fluctuation*. Perhaps, like response-time tolerances, different people have different tolerances for fluctuation. But perhaps there is a fluctuation tolerance factor that applies with reasonable universality across all users. The Apdex Application Performance Index standard, for example, assumes the response time *F* at which users become "frustrated" is universally four times the response time *T* at which their attitude shifts from being "satisfied" to merely "tolerating."[1]

The knee, regardless of how you define it or what we end up calling it, is an important parameter to the capacity-planning procedure that I described earlier in the main text of this article, and I believe it is an important parameter to the daily process of computer system workload management.

I will keep studying.

References

1. Apdex; http://www.apdex.org.
2. Millsap, C. Performance management: myths and facts (1999); http://method-r.com.
3. Samson, S. MVS performance legends. In *Proceedings of Computer Measurement Group Conference* (1988), 148–159.

Figure A. Gunther's maximum allowable utilization value ρ*T* is defined as the utilization producing the average response time *T*.

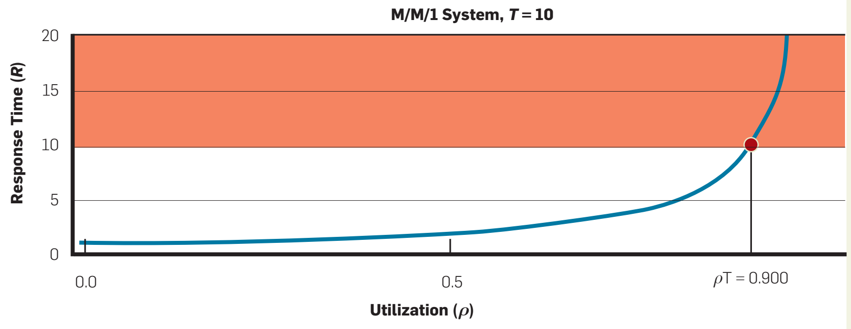
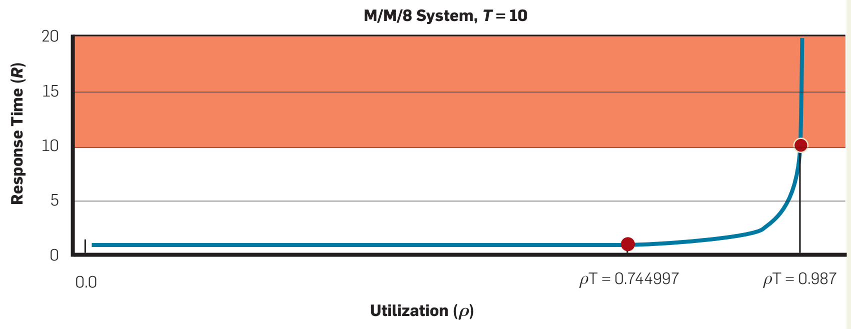


Figure B. Near ρ*T* value, small fluctuations in average utilization result in large response-time fluctuations.



isn't as good as the theoretical systems that M/M/*m* models. Therefore, the utilization values at which *your* system's knees occur will be more constraining than the values in Table 1. (I use the plural of *values* and *knees*, because you can model your CPUs with one model, your disks with another, your I/O controllers with another, and so on.)

To recap:

- Each of the resources in your system has a knee.
- That knee for each of your resources is less than or equal to the knee value you can look up in Table 1. The more imperfectly your system scales, the smaller (worse) your knee value will be.
- On a system with random request arrivals, if you allow your sustained utilization for any resource on your system to exceed your knee value for that resource, then you will have performance problems.
- Therefore, it is vital that you manage your load so that your resource utilizations will not exceed your knee values.

Capacity Planning

Understanding the knee can collapse a lot of complexity out of your capacity planning. It works like this:

- Your goal capacity for a given resource is the amount at which you can comfortably run your tasks at peak times without driving utilizations beyond your knees.
- If you keep your utilizations less than your knees, your system behaves roughly linearly—no big hyperbolic surprises.
- If you are letting your system run any of its resources beyond their knee utilizations, however, then you have performance problems (whether you are aware of them or not).
- If you have performance problems, then you don't need to be spending your time with mathematical models; you need to be spending your time fixing those problems by rescheduling load, eliminating load, or increasing capacity.

That's how capacity planning fits into the performance management process.

Random Arrivals

You might have noticed that I used the term *random arrivals* several times. Why is that important?

The reason the knee value is so important on a system with *random* arrivals is that these tend to cluster and cause temporary spikes in utilization.

Some systems have something that you probably do not have right now: completely deterministic job scheduling. Some systems—though rare these days—are configured to allow service requests to enter the system in absolute robotic fashion, say, at a pace of one task per second. And by this, I don't mean at an average rate of one task per second (for example, two tasks in one second and zero tasks in the next); I mean one task per second, as a robot might feed car parts into a bin on an assembly line.

If arrivals into your system behave completely deterministically—meaning that you know *exactly* when the next service request is coming—then you can run resource utilizations beyond their knee utilizations without necessarily creating a performance problem. On a system with deterministic arrivals, your goal is to run resource utilizations up to 100% without cramming so much workload into the system that requests begin to queue.

The reason the knee value is so important on a system with *random* arrivals is that these tend to cluster and cause temporary spikes in utilization. These spikes need enough spare capacity to consume so that users don't have to endure noticeable queuing delays (which cause noticeable fluctuations in response times) every time a spike occurs.

Temporary spikes in utilization beyond your knee value for a given resource are OK as long as they don't exceed a few seconds in duration. But how many seconds are too many? I believe (but have not yet tried to prove) that you should at least ensure that your spike durations do not exceed eight seconds. (You will recognize this number if you've heard of the "eight-second rule."[2]) The answer is certainly that if you're unable to meet your percentile-based response time promises or your throughput promises to your users, then your spikes are too long.

Coherency Delay

Your system does not have theoretically perfect scalability. Even if I have never studied your system specifically, it is a pretty good bet that no matter what computer application system you are thinking of right now, it does not meet the M/M/*m* "theoretically perfect scalability" assumption. *Coherency delay* is the factor that you can use to model the imperfection.[4] It is the duration that a task spends communicating and coordinating access to a shared resource. Like response time, service time, and queuing delay, coherency delay is measured in time per task execution, as in seconds per click.

I will not describe a mathematical model for predicting coherency delay, but the good news is that if you profile your software task executions, you'll see it when it occurs. In Oracle, timed events such as the following are examples of coherency delay:

- enqueue
- buffer busy waits
- latch free

You can not model such coherency delays with M/M/*m*. That is because M/M/*m* assumes all *m* of your service channels are parallel, homogeneous, and independent. That means the model assumes that after you wait politely in a FIFO queue for long enough that all the requests that enqueued ahead of you have exited the queue for service, it will be your turn to be serviced. Coherency delays don't work like that, however.

Imagine an HTML data-entry form in which one button labeled "Update" executes a SQL update statement, and another button labeled "Save" executes a SQL commit statement. An application built like this would almost guarantee abysmal performance. That is because the design makes it possible—quite likely, actually—for a user to click Update, look at his calendar, realize "uh-oh, I'm late for lunch," and then go to lunch for two hours before clicking Save later that afternoon.

The impact to other tasks on this system that wanted to update the same row would be devastating. Each task would necessarily wait for a lock on the row (or, on some systems, worse: a lock on the row's page) until the locking user decided to go ahead and click Save—or until a database administrator killed the user's session, which of course would have unsavory side effects to the person who thought he had updated a row.

In this case, the amount of time a task would wait on the lock to be released has nothing to do with how busy the system is. It would be dependent upon random factors that exist outside of the system's various resource utilizations. That is why you can not model this kind of thing in M/M/*m*, and it is why you can never assume that a performance test executed in a unit-testing type of environment is sufficient for a making a go/no-go decision about insertion of new code into a production system.

All this talk of queuing delays and coherency delays leads to a very difficult question: How can you possibly test a new application enough to be confident that you are not going to wreck your production implementation with performance problems?

Performance Testing

All this talk of queuing delays and coherency delays leads to a very difficult question: How can you possibly test a new application enough to be confident that you are not going to wreck your production implementation with performance problems?

You can model. And you can test.[1] Nothing you do will be perfect, however. It is extremely difficult to create models and tests in which you'll foresee all your production problems in advance of actually encountering those problems in production.

Some people allow the apparent futility of this observation to justify not testing at all. Do not get trapped in that mentality. The following points are certain:

- You will catch a lot more problems if you try to catch them prior to production than if you do not even try.
- You will never catch all your problems in preproduction testing. That is why you need a reliable and efficient method for solving the problems that leak through your preproduction testing processes.

Somewhere in the middle between "no testing" and "complete production emulation" is the right amount of testing. The right amount of testing for aircraft manufacturers is probably more than the right amount of testing for companies that sell baseball caps. But don't skip performance testing altogether. At the very least, your performance-test plan will make you a more competent diagnostician (and clearer thinker) when the time comes to fix the performance problems that will inevitably occur during production operation.

Measuring. People feel throughput and response time. Throughput is usually easy to measure, response time is much more difficult. (Remember, throughput and response time are *not* reciprocals.) It may not be difficult to time an end-user action with a stopwatch, but it might be very difficult to get what you really need, which is the ability to drill down into the details of why a given response time is as large as it is.

Unfortunately, people tend to measure what is easy to measure, which is not necessarily what they *should be* measuring. It's a bug. Measures that aren't what you need, but that are easy enough to obtain and seem related to what you need are called *surrogate measures*. Examples include subroutine call counts and samples of subroutine call execution durations.

I'm ashamed that I do not have greater command over my native language than to say it this way, but here is a catchy, modern way to express what I think about surrogate measures: *surrogate measures suck*.

Here, unfortunately, *suck* doesn't mean *never work*. It would actually be better if surrogate measures never worked. Then nobody would use them. The problem is that surrogate measures work *sometimes*. This inspires people's confidence that the measures they are using should work all the time, and then they don't. Surrogate measures have two big problems.

- They can tell you your system's OK when it is not. That's what statisticians call *type I error*, the false positive.
- They can tell you that something is a problem when it is not. That's a *type II error*, the false negative. I have seen each type of error waste years of people's time.

When the time comes to assess the specifics of a real system, your success is at the mercy of how good the measurements are that your system allows you to obtain. I have been fortunate to work in the Oracle market segment, where the software vendor at the center of our universe participates actively in making it possible to measure systems the right way. Getting application software developers to use the tools that Oracle offers is another story, but at least the capabilities are there in the product.

Performance is a Feature

Performance is a software application feature, just like recognizing that it's convenient for a string of the form "Case 1234" to automatically hyperlink over to case 1234 in your bug-tracking system. (FogBugz, which is software that I enjoy using, does this.) Performance, like any other feature, does not just happen; it has to be designed and built. To do performance well, you have to think about it, study it, write extra code for it, test it, and support it.

Like many other features, however, you can not know exactly how performance is going to work out while you're still writing, studying, designing, and creating the application. For many applications (arguably, for the vast majority), performance is completely unknown until the production phase of the software development life cycle. What this leaves you with is this: since you can't know how your application is going to perform in production, you need to write your application so that it's easy to *fix* performance in production.

As David Garvin has taught us, it's much easier to manage something that's easy to measure.[3] Writing an application that is easy to fix in production begins with an application that's easy to measure in production.

Usually, when I mention the concept of production performance measurement, people drift into a state of worry about the measurement-intrusion effect of performance instrumentation. They immediately enter a mode of data-collection compromise, leaving only surrogate measures on the table. Won't software with an extra code path to measure timings be slower than the same software without that extra code path?

I like an answer that I once heard Tom Kyte give in response to this question.[6] He estimated that the measurement-intrusion effect of Oracle's extensive performance instrumentation is −10% or less (where *or less* means *or better*, as in −20%, −30%, and so on). He went on to explain to a now-vexed questioner that the product is at least 10% faster now because of the knowledge that Oracle Corporation has gained from its performance instrumentation code, more than making up for any "overhead" the instrumentation might have caused.

I think that vendors tend to spend too much time worrying about how to make their measurement code path efficient without figuring out first how to make it effective. It lands squarely upon the idea that Knuth wrote about in 1974 when he said that "premature optimization is the root of all evil."[5] The software designer who integrates performance measurement into a product is much more likely to create a fast application and—more importantly—one that will become faster over time.

Acknowledgments

Thank you, Baron Schwartz for the email conversation in which you thought I was helping you, but in actual fact, you were helping me come to grips with the need for introducing coherency delay more prominently into my thinking. Thank you, Jeff Holt, Ron Crisco, Ken Ferlita, and Harold Palacio for the daily work that keeps the company going and for the lunchtime conversations that keep my imagination going. Thank you, Tom Kyte for your continued inspiration and support. Thank you, Mark Farnham for your helpful suggestions. And thank you, Neil Gunther for your patience and generosity in our ongoing discussions about knees.

References

1. CMG (Computer Measurement Group, a network of professionals who study these problems very, very seriously); http://www.cmg.org.
2. Eight-second rule; http://en.wikipedia.org/wiki/Network_performance#8-second_rule.
3. Garvin, D. Building a learning organization. *Harvard Business Review* (July 1993).
4. Gunther, N. Universal Law of Computational Scalability (1993); http://en.wikipedia.org/wiki/Neil_J._Gunther#Universal_Law_of_Computational_Scalability.
5. Knuth, D. Structured programming with Go To statements. *ACM Computing Surveys 6*, 4 (1974), 268.
6. Kyte, T. A couple of links and an advert…; http://tkyte.blogspot.com/2009/02/couple-of-links-and-advert.html.
7. Millsap, C. and Holt, J. *Optimizing Oracle Performance*. O'Reilly, Sebastopol, CA, 2003.
8. Oak Table Network; http://www.oaktable.net.

Cary Millsap is the founder and president of Method R Corporation (http://method-r.com), a company devoted to software performance. He is the author (with Jeff Holt) of *Optimizing Oracle Performance* (O'Reilly) and a co-author of *Oracle Insights: Tales of the Oak Table* (Apress). He is the former vice president of Oracle Corporation's System Performance Group and a co-founder of his former company Hotsos. He is also an Oracle ACE Director and a founding partner of the Oak Table Network, an informal association of well-known "Oracle scientists." Millsap blogs at http://carymillsap.blogspot.com, and tweets at http://twitter.com/CaryMillsap.