Thinking Clearly About Performance

Cary Millsap
Method R Corporation, Southlake, Texas, USA
cary.millsap@method-r.com

Revised 2010/01/22

Creating high-performance as an attribute of complex software is extremely difficult business for developers, technology administrators, architects, system analysts, and project managers. However, by understanding some fundamental principles, performance problem solving and prevention can be made far simpler and more reliable. This paper describes those principles, linking them together in a coherent journey covering the goals, the terms, the tools, and the decisions that you need to maximize your application's chance of having a long, productive, high-performance life. Examples in this paper touch upon Oracle experiences, but the scope of the paper is not restricted to Oracle products.

TABLE OF CONTENTS

1 AN AXIOMATIC APPROACH

When I joined Oracle Corporation in 1989, performance—what everyone called "Oracle tuning"—was difficult. Only a few people claimed they could do it very well, and those people commanded nice, high consulting rates. When circumstances thrust me into the "Oracle tuning" arena, I was quite unprepared. Recently, I've been introduced into the world of "MySQL tuning," and the situation seems very similar to what I saw in Oracle over twenty years ago.

It reminds me a lot of how difficult I would have told you that beginning algebra was, if you had asked me when I was about 13 years old. At that age, I had to appeal heavily to my "mathematical instincts" to solve equations like $3x + 4 = 13$. The problem with that is that many of us didn't *have* mathematical instincts. I can remember looking at a problem like "$3x + 4 = 13$; find x" and basically stumbling upon the answer $x = 3$ using trial and error. Maybe I tried $x = 2$ first, yielding a left-hand side value of 10, which was smaller than 13, so I tried a larger number. And, happily, $x = 3$ happened to work.

The trial-and-error method of feeling my way through algebra problems worked—albeit slowly and uncomfortably—for easy equations, but it didn't scale as the problems got tougher, like "$3.1x + 4 = 13$." Now what? It was hard. My problem was that I wasn't "thinking clearly" yet about algebra. My introduction at age fifteen to Jimmy Harkey put me on the road to solving that problem.

Mr. Harkey taught us what he called an *axiomatic approach* to solving algebra equations. He showed us a set of steps that worked every time (and he gave us plenty of homework to practice on). In addition to working every time, by executing those steps, we necessarily *documented* our thinking as we worked. Not only were we thinking clearly, using a reliable and repeatable sequence of steps, we were *proving* to anyone who read our work that we were thinking clearly.

Our work for Mr. Harkey looked like this:

$3.1x + 4$	$= 13$	*problem statement*
$3.1x + 4 - 4$	$= 13 - 4$	*subtraction property of equality*
$3.1x$	$= 9$	*simplification*
$3.1x/3.1$	$= 9/3.1$	*division property of equality*
x	≈ 2.903	*simplification*

This was Mr. Harkey's axiomatic approach to algebra, geometry, trigonometry, and calculus: one small, logical, provable, and auditable step at a time. It's the first time I ever really *got* mathematics.

Naturally, I didn't realize it at the time, but of course *proving* was a skill that would be vital for my success in the world after school. In life, I've found that it's not nearly as important to be right as it is to be able to convince others that you're right.

My goal since the mid-1990s has been to create a similarly rigorous approach to Oracle performance optimization. Lately, I am expanding the scope of that goal beyond Oracle, to: "Create an axiomatic approach to computer software performance optimization." I've found that not many people really like it when I talk like that, so let's say it like this:

> My goal is to help you *think clearly* about how to optimize the performance of your computer software.

2 WHAT IS PERFORMANCE?

If you google for the word *performance*, you get over a half a billion hits on concepts ranging from bicycle racing to the dreaded employee review process that many companies these days are learning to avoid. When I googled for *performance*, most of the top hits relate to the subject of this paper: the time it takes for computer software to perform whatever task you ask it to do.

And that's a great place to begin: the *task*. A ***task*** is a business-oriented unit of work. Tasks can nest: *print invoices* is a task; *print one invoice*—a sub-task—is also a task. When a computer user talks about *performance* he usually means the time it takes for the system to execute some task. ***Response time*** is the execution duration of a task, measured in time per task, like "seconds per click." For example, my Google search for the word *performance* had a response time of 0.24 seconds. The Google web page rendered that measurement right in my browser. That is evidence, to me, that Google values my perception of Google performance.

Some people are interested in another performance measure: ***Throughput*** is the count of task executions that complete within a specified time interval, like "clicks per second." In general, people who are responsible for the performance of *groups* of people worry more about throughput than people who work in a solo contributor role. For example, an individual accountant is usually more concerned about whether the response time of a daily report will require him to stay late after work today. The manager of a group of accounts is additionally concerned about whether the system is capable of processing all the data that all of her accountants will be processing today.

3 RESPONSE TIME VS THROUGHPUT

Throughput and response time have a generally reciprocal type of relationship, but not exactly. The real relationship is subtly complex.

> ***Example:*** Imagine that you have measured your throughput at 1,000 tasks per second for some benchmark. What, then, is your users' average response time?
>
> It's tempting to say that your average response time was 1/1,000 = .001 seconds per task. But it's not necessarily so.
>
> Imagine that your system processing this throughput had 1,000 parallel, independent, homogeneous service channels inside it (that is, it's a system with 1,000 independent, equally competent service providers inside it, each awaiting your request for service). In this case, it is possible that each request consumed exactly 1 second.
>
> Now, we *can* know that average response time was somewhere between 0 seconds per task and 1 second per task. But you cannot derive response time exclusively[1] from a throughput measurement. You have to measure it separately.

[1] I carefully include the word *exclusively* in this statement, because there are mathematical models that can compute response time for a given throughput, but the models require more input than just *throughput*.

The subtlety works in the other direction, too. You can certainly flip the example I just gave around and prove it. However, a scarier example will be more fun.

> ***Example:*** Your client requires a new task that you're programming to deliver a throughput of 100 tasks per second on a single-CPU computer. Imagine that the new task you've written executes in just .001 seconds on the client's system. Will your program yield the throughput that the client requires?
>
> It's tempting to say that if you can run the task once in just a thousandth of a second, then surely you'll be able to run that task at least a hundred times in the span of a full second. And you're right, *if* the task requests are nicely serialized, for example, so that your program can process all 100 of the client's required task executions inside a loop, one after the other.
>
> But what if the 100 tasks per second come at your system at random, from 100 different users logged into your client's single-CPU computer? Then the gruesome realities of CPU schedulers and serialized resources (like Oracle latches and locks and writable access to buffers in memory) may restrict your throughput to quantities much less than the required 100 tasks per second.
>
> It might work. It might not. You cannot derive throughput exclusively from a response time measurement. You have to measure it separately.

Response time and throughput are not necessarily reciprocals. To know them both, you need to measure them both.

So, which is more important: response time, or throughput? For a given situation, you might answer legitimately in either direction. In many circumstances, the answer is that *both* are vital measurements requiring management. For example, a system owner may have a business requirement that response time must be 1.0 seconds or less for a given task in 99% or more of executions *and* the system must support a sustained throughput of 1,000 executions of the task within a 10-minute interval.

4 PERCENTILE SPECIFICATIONS

In the prior section, I used the phrase "in 99% or more of executions" to qualify a response time expectation. Many people are more accustomed to statements like, "*average* response time must be *r* seconds or less." The percentile way of stating requirements maps better, though, to the human experience.

> ***Example:*** Imagine that your response time tolerance is 1 second for some task that you execute on your computer every day. Further imagine that the lists of numbers shown in Exhibit 1 represent the measured response times of ten executions of that task. The average response time for each list is 1.000 seconds. Which one do you think you'd like better?

List A	List B
.924	.796
.928	.798
.954	.802
.957	.823
.961	.919
.965	.977
.972	1.076
.979	1.216
.987	1.273
1.373	1.320

Exhibit 1. The average response time for each of these two lists is 1.000 seconds.

> You can see that although the two lists have the same average response time, the lists are quite different in character. In List A, 90% of response times were 1 second or less. In List B, only 60% of response times were 1 second or less. Stated in the opposite way, List B represents a set of user experiences of which 40% were dissatisfactory, but List A (having the same average response time as List B) represents only a 10% dissatisfaction rate.
>
> In List A, the 90th percentile response time is .987 seconds. In List B, the 90th percentile response time is 1.273 seconds. These statements about percentiles are more informative than merely saying that each list represents an average response time of 1.000 seconds.

As GE says, "Our customers feel the variance, not the mean."[2] Expressing response time goals as percentiles make for much more compelling requirement specifications that match with end user expectations:

> The *Track Shipment* task must complete in less than .5 seconds in at least 99.9% of executions.

5 PROBLEM DIAGNOSIS

In nearly every performance problem I've been invited to repair, the problem statement has been a statement about response time. "It used to take less than a second to do *X*; now it sometimes takes 20+." Of course, a specific problem statement like that is often buried behind veneers of other problem statements,

[2] General Electric Company: "What Is Six Sigma? The Roadmap to Customer Impact" at *http://www.ge.com/sixsigma/SixSigma.pdf*

like, "Our whole [adjectives deleted] system is so slow we can't use it."[3]

But just because something has happened a lot for me doesn't mean that it's what will happen next for you. The most important thing for you to do first is state the problem clearly, so that you can then think about it clearly.

A good way to begin is to ask, what is the goal state that you want to achieve? Find some specifics that you can *measure* to express this. For example, "Response time of *X* is more than 20 seconds in many cases. We'll be happy when response time is 1 second or less in at least 95% of executions."

That sounds good in theory, but what if your user doesn't have a specific quantitative goal like "1 second or less in at least 95% of executions?" There are two quantities right there (1 and 95); what if your user doesn't know either one of them? Worse yet, what if your user *does* have specific ideas about his expectations, but those expectations are impossible to meet? How would you know what "possible" or "impossible" even is?

Let's work our way up to those questions.

6 THE SEQUENCE DIAGRAM

A ***sequence diagram*** is a type of graph specified in the Unified Modeling Language (UML), used to show the interactions between objects in the sequential order that those interactions occur. The sequence diagram—with a few extra features that the authors of the UML probably never intended—is an exceptionally useful tool for visualizing response time. Exhibit 2 shows a standard UML sequence diagram for a simple application system composed of a browser, an application server, and a database.

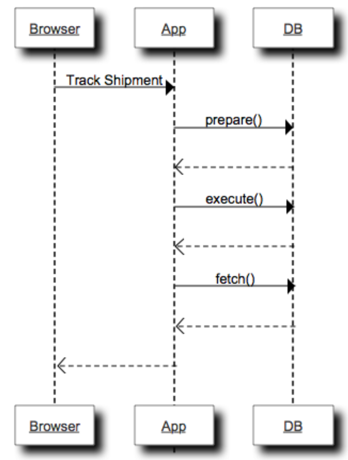

Exhibit 2. This UML sequence diagram shows the interactions among a browser, an application server, and a database.

Imagine now drawing the sequence diagram to scale, so that the distance between each "request" arrow coming and its corresponding "response" arrow going out were proportional to the duration spent servicing the request. I've shown such a diagram in Exhibit 3.

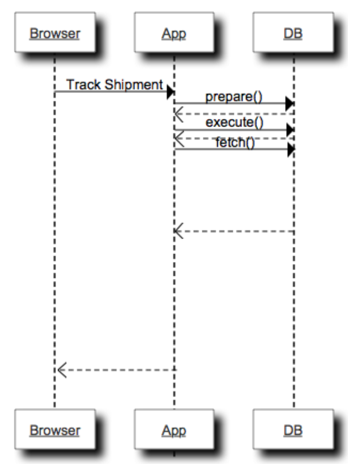

Exhibit 3. A UML sequence diagram drawn to scale, showing the response time consumed at each tier in the system.

With Exhibit 3, you have a good graphical representation of how the components represented in your diagram are spending your user's time. You can "feel" the relative contribution to response time by looking at the picture.

Sequence diagrams are just right for helping people conceptualize how their response is consumed on a given system, as one tier hands control of the task to the next. Sequence diagrams also work well to show how simultaneous threads of processing work in parallel. Sequence diagrams are good tools for

[3] Cary Millsap, 2009. "My whole system is slow. Now what?" at *http://carymillsap.blogspot.com/2009/12/my-whole-system-is-slow-now-what.html*

analyzing performance outside of the information technology business, too.[4]

The sequence diagram is a good conceptual tool for talking about performance, but to think clearly about performance, we need something else, too. Here's the problem. Imagine that the task you're supposed to fix has a response time of 2,468 seconds (that's 41 minutes 8 seconds). In that roughly 41 minutes, running that task causes your application server to execute 322,968 database calls. Exhibit 4 shows what your sequence diagram for that task would look like.

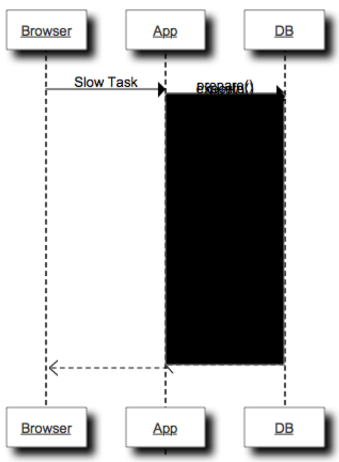

Exhibit 4. This UML sequence diagram shows 322,968 database calls executed by the application server.

There are so many request and response arrows between the application and database tiers that you can't see any of the detail. Printing the sequence diagram on a very long scroll isn't a useful solution, because it would take us weeks of visual inspection before we'd be able to derive useful information from the details we'd see.

The sequence diagram is a good tool for conceptualizing flow of control and the corresponding flow of time. But to think clearly about response time, we need something else.

7 THE PROFILE

The sequence diagram doesn't scale well. To deal with tasks that have huge call counts, we need a convenient aggregation of the sequence diagram so that we understand the most important patterns in how our time has been spent. Exhibit 5 shows an example of a table called a *profile*, which does the trick. A ***profile*** is a tabular decomposition of response time, typically listed in descending order of component response time contribution.

	Function call	*R* (sec)	Calls
1	DB: fetch()	1,748.229	322,968
2	App: await_db_netIO()	338.470	322,968
3	DB: execute()	152.654	39,142
4	DB: prepare()	97.855	39,142
5	Other	58.147	89,422
6	App: render_graph()	48.274	7
7	App: tabularize()	23.481	4
8	Browser: render()	0.890	2
	Total	**2,468.000**	

Exhibit 5. This profile shows the decomposition of a 2,468.000-second response time.

> ***Example:*** The profile in Exhibit 5 is rudimentary, but it shows you exactly where your slow task has spent your user's 2,468 seconds. With the data shown here, for example, you can derive the percentage of response time contribution for each of the function calls identified in the profile. You can also derive the average response time for each type of function call during your task.

A profile shows you *where your code has spent your time* and—sometimes even more importantly—where it has *not*. There is tremendous value in not having to guess about these things.

From the data shown in Exhibit 5, you *know* that 70.8% of your user's response time is consumed by *DB:fetch()* calls. Furthermore, if you can drill down in to the individual calls whose durations were aggregated to create this profile, you can know how many of those *App:await_db_netIO()* calls corresponded to *DB:fetch()* calls, and you can know how much response time each of those consumed. With a profile, you can begin to formulate the answer to the question, "How long *should* this task run?" …Which, by now, you know is an important question in the first step (section 5) of any good problem diagnosis.

8 AMDAHL'S LAW

Profiling helps you think clearly about performance. Even if Gene Amdahl hadn't given us Amdahl's Law back in 1967, you'd have probably come up with it yourself after the first few profiles you looked at. Amdahl's Law states:

[4] Cary Millsap, 2009. "Performance optimization with Global Entry. Or not?" at *http://carymillsap.blogspot.com/2009/11/performance-optimization-with-global.html*

> Performance improvement is proportional to how much a program uses the thing you improved.

So if the thing you're trying to improve only contributes 5% to your task's total response time, then the maximum impact you'll be able to make is 5% of your total response time. This means that the closer to the top of a profile that you work (assuming that the profile is sorted in descending response time order), the bigger the benefit potential for your overall response time.

This doesn't mean that you always work a profile in top-down order, though, because you also need to consider the *cost* of the remedies you'll be executing, too.[5]

> ***Example:*** Consider the profile in Exhibit 6. It's the same profile as in Exhibit 5, except here you can see how much time you think you can save by implementing the best remedy for each row in the profile, and you can see how much you think each remedy will cost to implement.

	Potential improvement % and cost of investment	*R* (sec)	*R* (%)
1	50% super expensive	1,748.229	70.8%
2	90% dirt cheap	338.470	13.7%
3	Impossible to improve	152.654	6.2%
4	99% dirt cheap	97.855	4.0%
5	1% super expensive	58.147	2.4%
6	80% dirt cheap	48.274	2.0%
7	Impossible to improve	23.481	1.0%
8	Impossible to improve	0.890	0.0%
	Total	**2,468.000**	

Exhibit 6. This profile shows the potential for improvement and the corresponding cost (difficulty) of improvement for each line item from Exhibit 5.

> What remedy action would you implement first? Amdahl's Law says that implementing the repair on line 1 has the greatest potential benefit of saving about 874 seconds (50% of 1,748.229 seconds). But if it is truly "super expensive," then the remedy on line 2 may yield better *net* benefit—and that's the constraint to which we really need to optimize—even though the potential for response time savings is only about 305 seconds.

A tremendous value of the profile is that you can learn exactly how much improvement you should expect for a proposed investment. It opens the door to making much better decisions about what remedies to implement first. Your predictions give you a yardstick for measuring your own performance as an analyst. And finally, it gives you a chance to showcase your cleverness and intimacy with your technology as you find more efficient remedies for reducing response time more than expected, at lower-than-expected costs.

What remedy action you implement first really boils down to how much you trust your cost estimates. Does "dirt cheap" really take into account the risks that the proposed improvement may inflict upon the system? For example, it may seem "dirt cheap" to change that parameter or drop that index, but does that change potentially disrupt the good performance behavior of something out there that you're not even thinking about right now? Reliable cost estimation is another area in which your technological skills pay off.

Another factor worth considering is the political capital that you can earn by creating small victories. Maybe cheap, low-risk improvements won't amount to much overall response time improvement, but there's value in establishing a track record of small improvements that exactly fulfill your predictions about how much response time you'll save for the slow task. A track record of prediction and fulfillment ultimately—especially in the area of software performance, where myth and superstition have reigned at many locations for decades—gives you the credibility you need to influence your colleagues (your peers, your managers, your customers, …) to let you perform increasingly expensive remedies that may produce bigger payoffs for the business.

A word of caution, however: Don't get careless as you rack up your successes and propose ever bigger, costlier, riskier remedies. Credibility is fragile. It takes a lot of work to build it up but only one careless mistake to bring it down.

9 MINIMIZING RISK

The subject of the risk of how 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.[6] Roughly

[5] Cary Millsap, 2009. "On the importance of diagnosing before resolving" at *http://carymillsap.blogspot.com/2009/09/on-importance-of-diagnosing-before.html*

[6] The Oak Table Network is a network of Oracle practitioners who believe in using scientific methods to improve the development and administration of Oracle-based systems (*http://www.oaktable.net*).

ten people sat 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's 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 badly-written Java programs that make unnecessarily many database calls (and hence unnecessarily many network I/O calls as well). If everybody's 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.

Even if everyone on the entire system is suffering, you should still focus first on the program that the business needs fixed first. The way to begin is to ensure that 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:

Example: 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).

Example: 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).

Example: A SQL statement[7] touches the database buffer cache 7,428,322 times to return a 698-row result set. An extra filter predicate could have returned the 7 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 (e.g., ill-conceived index, badly set parameter, poorly configured hardware), then you should fix it. But don't tune a system to accommodate programs that are inefficient. There is a lot more leverage in curing the program inefficiencies themselves. Even if the programs are commercial, off-the-shelf applications, it will benefit you better 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 be as efficient as it can with inherently inefficient workload.

Improvements that make your program more efficient can produce tremendous benefits for everyone on the system. It's easy to see how top-line reduction of waste helps the response time of the task being repaired. What many people don't 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.

10 LOAD

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

One measure of load is *utilization*. ***Utilization*** 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 toll booth.

With computer software, the software you use doesn't actually "go slower" like your car does when you're going 30 mph in heavy traffic instead of 60 mph 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: *queueing delay*, and *coherency delay*. I'll address each as we continue.

[7] My choice of article adjective here is a dead giveaway that I was introduced to SQL within the Oracle community.

11 QUEUEING DELAY

The mathematical relationship between load and response time is well known. I've already written quite a bit about one mathematical model called "M/M/m" that relates response time to load in systems that meet one particularly useful set of specific requirements.[8] One of the assumptions of M/M/m is that the system you are modeling has theoretically perfect scalability. Having "theoretically perfect scalability" 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 like the one about perfect scalability, M/M/m has a lot to teach us about performance. Exhibit 7 shows the relationship between response time and load using M/M/m.

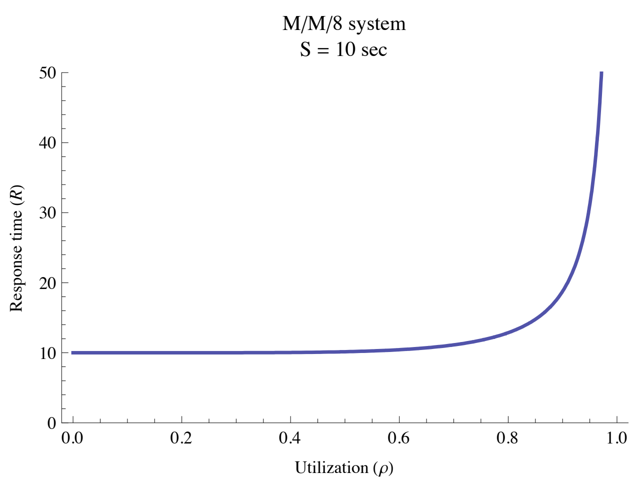

Exhibit 7. This curve relates response time as a function of utilization for an M/M/8 system.

In Exhibit 7, you can see mathematically what you can feel when you use a system under different load conditions. At low load, your response time is essentially the same as your response time was at *no* load. As load ramps up, you sense a slight, gradual degradation in response time. That gradual degradation doesn't 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, exponential.

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

So, when you order lunch at Taco Tico, your response time (R) for getting your order is the queueing delay time (Q) that you spend queued 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. Queueing 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).

12 THE KNEE

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

1. You want the best response time you can get: you don't want to have to wait too long for tasks to get done.
2. And you want 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 other one requires you to maximize it. You can't 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*. The ***knee*** is the utilization value for a resource at which throughput is maximized with minimal negative impact to response times. 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 that 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 straightedge, as shown in Exhibit 8.

[8] Cary Millsap and Jeff Holt, 2003. *Optimizing Oracle Performance*. O'Reilly. Sebastopol CA.

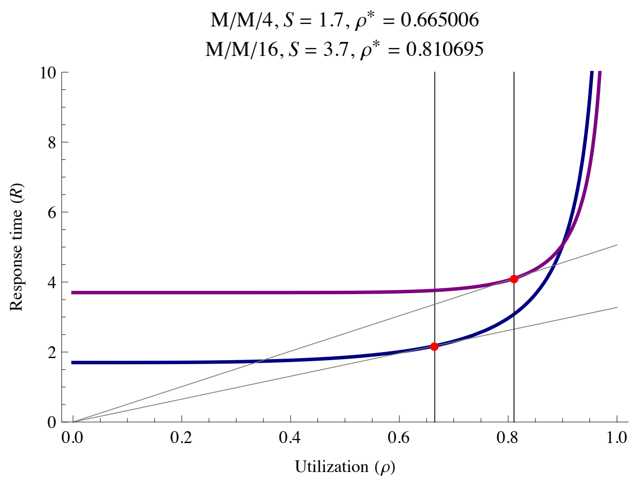

Exhibit 8. The knee occurs at the utilization at which a line through the origin is tangent to the response time curve.

Another nice property of the M/M/*m* knee is that you only need to know the value of 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 such resources, like a booth in a toll plaza or a CPU in an SMP computer.

The italicized lowercase *m* in the name M/M/*m* is the number of service channels in the system being modeled.[9] The M/M/*m* knee value for an arbitrary system is difficult to calculate, but I've done it for you. The knee values for some common service channel counts are shown in Exhibit 9.

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

Exhibit 9. M/M/m knee values for common values of m.

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 severely degraded 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.

13 RELEVANCE OF THE KNEE

So, 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're thinking about scales perfectly. I know what you're thinking: It doesn't.

But what M/M/*m* gives 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 I've given you in Exhibit 9. Your system 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 I've given you in Exhibit 9. (I said *values* and *knees* in plural form, because you can model your CPUs with one model, your disks with another, your I/O controllers with another, and so on.)

Allow me to recap.

If you have a system with random request arrivals, then:

- 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 Exhibit 9. The more imperfectly your system scales, the smaller (worse) your knee value will be.

[9] By this point, you may be wondering what the other two 'M's stand for in the M/M/*m* queueing 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* for even more.

- If you allow your sustained utilization for any resource on your system to exceed your knee value for that resource, then you'll have performance problems.

Therefore, it is vital that you manage your load so that your resource utilizations will not exceed your knee values.

14 CAPACITY PLANNING

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

1. 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.
2. If you keep your utilizations less than your knees, your system behaves roughly linearly: no big exponential surprises.
3. However, if you're letting your system run any of its resources beyond their knee utilizations, then you have performance problems (whether you're aware of it or not).
4. 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 either rescheduling load, eliminating load, or increasing capacity.

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

15 RANDOM ARRIVALS

You might have noticed that several times now, I have mentioned the term "random arrivals." Why is that important?

Some systems have something that you probably don't have right now: completely deterministic job scheduling. Some systems—it's 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 "one task per second," I don't mean at an average rate of one task per second (for example, 2 tasks in one second and 0 tasks in the next), I mean one task per second, like 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 random arrivals tend to cluster up on you and cause temporary spikes in utilization. Those spikes need enough spare capacity to consume so that users don't have to endure noticeable queueing delays (which cause noticeable fluctuations in response times) every time one of those spikes occurs.

Temporary spikes in utilization beyond your knee value for a given resource are ok as long as they don't exceed just a few seconds in duration. How many seconds is too many? I believe (but have not yet tried to prove) that you should at least ensure that your spike durations do not exceed 8 seconds.[10] 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.

16 COHERENCY DELAY

Your system doesn't have theoretically perfect scalability. Even if I've never studied *your* system specifically, it's a pretty good bet that no matter what computer application system you're 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.[11] ***Coherency delay*** is the duration that a task spends communicating and coordinating access to a shared resource. Like response time, service time, and queueing delay, coherency delay is measured in time per task execution, as in *seconds per click*.

I won't describe here 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 like the following are examples of coherency delay:

enqueue

[10] You'll recognize this number if you've heard of the "8-second rule," which you can learn about at *http://en.wikipedia.org/wiki/Network_performance#8-second_rule*.

[11] Neil Gunther, 1993. Universal Law of Computational Scalability, at *http://en.wikipedia.org/wiki/Neil_J._Gunther#Universal_Law_of_Computational_Scalability*.

buffer busy waits
latch free

You can't model coherency delays like these with M/M/*m*. That's because M/M/*m* assumes that 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'll be your turn to be serviced. However, coherency delays don't work like that.

> ***Example:*** 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's because the design makes it possible—quite likely, actually—for a user to click Update, look at his calendar, realize uh-oh he's 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 had 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's why you can't model this kind of thing in M/M/*m*, and it's 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.

17 PERFORMANCE TESTING

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

You can model. And you can test.[12] However, nothing you do will be perfect. 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. Don't get trapped in that mentality. The following points are certain:

- You'll catch a lot more problems if you try to catch them prior to production than if you don't even try.
- You'll never catch all your problems in pre-production testing. That's why you need a reliable and efficient method for solving the problems that leak through your pre-production 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 it comes time to fix the performance problems that will inevitably occur during production operation.

18 MEASURING

People feel throughput and response time. Throughput is usually easy to measure. Measuring response time is usually 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's easy to measure, which is not necessarily what they *should* be measuring. It's a bug. When it's not easy to measure what we need to measure, we tend to turn our attention to measurements that *are* easy to get. 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 of surrogate measures include subroutine call counts and samples of subroutine call execution durations.

I'm ashamed that I don't have greater command over my native language than to say it this way, but here is what I think about surrogate measures:

> Surrogate measures suck.

[12] The Computer Measurement Group is a network of professionals who study these problems very, very seriously. You can learn about CMG at *http://www.cmg.org*.

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're 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's not. That's what statisticians call *type I error*, the false positive. And they can tell you that something is a problem when it's not. That's what statisticians call *type II error*, the false negative. I've seen each type of error waste years of people's time.

When it comes time 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've 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.

19 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.[13] Performance, like any other feature, doesn't 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.

However, like many other features, you can't know exactly how performance is going to work out while it's still early in the project when you're 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.[14] Writing an application that's easy to fix in production begins with an application that's easy to measure in production.

Most times, 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 extra code path to measure timings be slower than the same software without that extra code path?

I like an answer that I heard Tom Kyte give once in response to this question.[15] He estimated that the measurement intrusion effect of Oracle's extensive performance instrumentation is *negative* 10% or less.[16] 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 how first 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."[17] The software designer who integrates performance measurement into his product is much more likely to create a fast application and—more importantly—an application that will become faster over time.

20 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

---

[13] FogBugz, which is software that I enjoy using, does this.

[14] David Garvin, 1993. "Building a Learning Organization" in *Harvard Business Review*, Jul. 1993.

[15] Tom Kyte, 2009. "A couple of links and an advert…" at *http://tkyte.blogspot.com/2009/02/couple-of-links-and-advert.html*.

[16] …Where *or less* means *or better*, as in –20%, –30%, etc.

[17] Donald Knuth, 1974. "Structured Programming with Go To Statements" in ACM Journal *Computing Surveys*, Vol. 6, No. 4, Dec. 1974, p268.

keeps the company going and for the lunchtime conversations that keep my imagination going. And thank you Tom Kyte for your continued inspiration and support.

21 ABOUT THE AUTHOR

Cary Millsap is well known in the global Oracle community as a speaker, educator, consultant, and writer. He is the founder and president of Method R Corporation (*http://method-r.com*), a small company devoted to genuinely satisfying software performance. Method R offers consulting services, education courses, and software tools—including the Method R Profiler, MR Tools, the Method R SLA Manager, and the Method R Instrumentation Library for Oracle—that help you optimize your software performance.

Cary is the author (with Jeff Holt) of *Optimizing Oracle Performance* (O'Reilly), for which he and Jeff were named *Oracle Magazine*'s 2004 Authors of the Year. He is 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 called Hotsos. Cary is also an Oracle ACE Director and a founding partner of the Oak Table Network, an informal association of "Oracle scientists" that are well known throughout the Oracle community. Cary blogs at *http://carymillsap.blogspot.com*, and he tweets at *http://twitter.com/CaryMillsap*.