Performance Management: Myths & Facts

Cary V. Millsap
Oracle Corporation

June 28, 1999

Performance management consists of problem diagnosis and repair, resource management, application optimization, and capacity planning. In constructing a reliable performance management method for hundreds of Oracle database sites, my colleagues and I have encountered bits of interesting folklore that, ironically, *block* progress toward the ultimate goal of lasting system performance satisfaction. In this paper, I will take a fun look at the dangers of several popular but bad guidelines, and I will offer alternative advice that helps you avoid the risk of costly performance management mistakes while not losing sight of the friendly goals: fast, easy, and cheap.

Contents

1. Introduction

In my colleagues' and my construction of a reliable performance management method for Oracle systems, we have encountered bits of interesting folklore that, ironically, *block* progress toward the ultimate goal of lasting system performance satisfaction. A rule of thumb is by definition a useful compromise between precision and simplicity that errs in favor of simplicity. However, there is a big difference between a rule of thumb, which is useful; and a myth, which is treacherous. We need rules of thumb that are fast, easy, and cheap, but we *don't* need guidelines that mislead us.

Certainly, it is much easier to poke holes into other people's rules of thumb and to call them "folklore" or "myths" than it is to create accurate guidelines that are fast, easy, and cheap. In this paper, I shall try to hold myself to two standards. First, I will criticize a statement only if it meets two criteria:

1. The statement is *often wrong*. It is not sufficient to show that a statement is wrong under only a contrived or bizarre set of conditions. It must be incorrect under sufficiently many normal circumstances as to pose a material risk to the system designers who would need to rely upon the guideline.
2. When the statement *is* wrong, the consequence of having relied on it is *highly painful*.

Second, for every tradition that I characterize as a "myth," I will offer alternative advice that mitigates the specific risks I describe, as quickly, easily, and inexpensively as possible. My mission in this paper is to help you avoid the risk of costly performance management mistakes while not losing sight of the friendly goals: fast, easy, and cheap.

2. The CPU Upgrade Myth

MYTH: Installing a faster CPU always helps performance.

This myth illustrates the value of analytical performance models with drama. The statement is almost irresistible to the intuition, especially when you're desperate for a performance boost and your hardware provider is encouraging you along. However, this statement is wrong with astonishing frequency, and the damage that it can inflict upon IT credibility is tremendous—sometimes irreversible.

People commonly fail to perceive even the *possibility* that a CPU upgrade could be a performance *risk*. Why, surely a CPU upgrade couldn't *hurt* my performance![1] And even if there's a chance that a CPU upgrade won't help my system, isn't the likelihood so great that my expected benefits greatly outweigh the cost of the upgrade?

In actual fact, upgrading CPUs can *hurt* system performance. And even when an upgrade helps, there is often a better way to spend the same money that would have yielded a greater system performance return on investment. How can something so counterintuitive be not only possible, but *likely*? Neil Gunther presents a splendid example [Gunther (1998) 117–122] that clearly illustrates how the "CPU upgrade always helps" statement can be untrue, and how it can be untrue with such astonishing frequency.

He begins by setting up a simple system, like many typical Oracle applications, that hosts both batch jobs and interactive users.

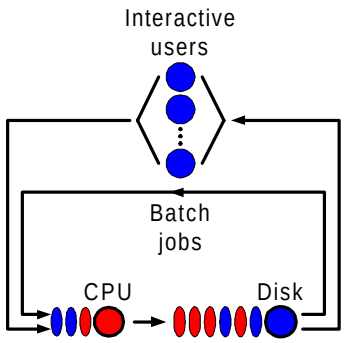


Figure 1. A Simple Model of a Computer

Each batch job requests service from a CPU, for which it queues behind competing requests that arrived before it. Once it passes through the CPU queue and receives CPU service, it queues for service

[1] I'm not going to trick you with some notion like, "If the new faster CPU provides lower reliability because it is so new and therefore untested…." It's something to consider, but the point I shall make in a moment is much more fundamental.

from a disk. After it clears the disk queue and receives I/O service, the batch job then is either finished, or it immediately hits the CPU queue again. When a batch job is finished, there's another job waiting to run.

Interactive users go through the same routine, with just one difference. When a user request has been served by the CPU and disk, then instead of immediately queueing again for the CPU, the user consumes "think time" before making the next request.

Gunther supplies example parameters to this model, such as how many batch jobs and users there are, how long the users' think times are, the speeds with which the CPU and disk can service requests, and the completion rate of jobs through the system. Each of these parameters is a real-life, operationally measurable quantity. He then uses a well-known and widely accepted algorithm[2] to predict the effect of upgrading to a five-times-faster CPU. To the reader's presumed astonishment, the upgrade results in what Gunther advertises as a 20 percent *degradation* in response time for the online users.[3]

Gunther's realistic example is a dramatic illustration of what happens when you tune something that's not the bottleneck. The bottleneck for Gunther's batch jobs was the CPU, but the interactive jobs were bottlenecked on the disk. With the introduction of a five-times-faster CPU, the batch jobs ran faster. But because they cleared the CPU five times more quickly, batch jobs issued disk service requests more quickly. The intensified competition for disk I/O service made the users' already existing bottleneck even worse, making their workdays miserable compared to what they had before the upgrade.

We don't need a lot of fine print to convert the "CPU upgrade always helps" myth into an accurate and usable rule of thumb:

Installing a faster CPU will help performance only if CPU is the bottleneck.

When you add capacity, add it to the bottleneck device.

CPU upgrades are often the result of desperation. The system is too slow, and we all know that in our young industry, there aren't enough people available who know how to diagnose or optimize complex application performance problems. And rather than trying to find the system's true bottleneck when you don't know how, the CPU upgrade plan often emerges as the savior.

However, "Just buy a faster CPU" is a truly treacherous myth, because the strategy it represents so often blocks the analysis required to maximize system performance per dollar of investment. Even in cases in which a CPU upgrade has improved performance for individual users by a factor of two (a nice gain for a CPU upgrade!), a plan of application optimization could often have provided opportunity to improve performance for those users by a factor of tens, hundreds, or even thousands.

System performance problems can impact reputations all the way to your customers, your board, and your shareholders. The great danger of the "CPU upgrade always helps" myth is that choosing the wrong plan in an environment of tremendous stress and equally tremendous expectations can cost your career. "CPU upgrade always helps" is a bad guideline.

Before any upgrade, determine—by testing or at least by modeling—whether it will have an unexpectedly adverse effect on your system's performance.

3. The User Count Myth

MYTH: The number of users accessing a system is a good predictor of CPU requirements.

It used to happen every week. Someone would describe an application, a machine, how much memory, the disk model. Then would come the inevitable question: "600 users. How many CPUs?" It's just not possible to give a reliable answer to that one.

The reason? Because total workload depends on a lot more than just the number of users:

{total workload}
= {# jobs} × {avg. load per job}
+ {# users} × {avg. load per user}.

[2] See [Kleinrock (1975), Kleinrock (1976), Jain (1991), Allen (1994), Menascé (1994), Tanner (1994), Gunther (1998)], and others for a description of the queueing theory models that Gunther uses.

[3] An incomplete self-assigned bit of homework is to verify with Dr. Gunther whether the 20% figure he quotes isn't actually a typographical error. I presently believe that the degradation is actually 153% (even worse than Gunther's already dramatic claim!)—from an average online response time of 3.99 seconds before the upgrade to 10.11 seconds after.

In systems that serve a mixture of online and batch processing, the batch loads dominate the total workload of the system, often overwhelmingly. One batch job can generate as much workload as a hundred or more interactive users. Thus, not only is user count one of four factors of total workload, it's possibly the *least significant* of four factors. Your commitment to *managing* your interactive and batch workload is the chief determinant of the total required capacity of a new system.

Unfortunately, there is no reliable user-count rule of thumb for sizing most Oracle systems. Furthermore, such a rule of thumb *can never exist* for an application that is either flexibly configurable or that does not impose a strict batch workload discipline upon its users. Consequently, capacity planners seeking a simple user-count rule of thumb are doomed never to find one for most applications.

For example, Oracle Applications™ (e.g., Financials and Manufacturing) generate dynamic SQL that is unique to an implementation-specific set of customization options. And our Concurrent Manager™ batch queue management system provides an extremely flexible means for an Oracle Applications owner to execute whatever batch workload discipline he or she likes—including no discipline at all. It is impossible to make precise workload forecasts for an Oracle Applications system without being able to test the specific setup options, and without imposing strict constraints upon the allowable amount of batch processing.

Use-case policies also drive huge workload variance. For example, do you generate aged trial balance reports daily? or monthly? Do you generate manufacturing alerts for each process status change? or in one-hour intervals? Is your system 100% out-of-the-box? or do you use six custom forms and twenty custom reports? Each of these differences in *how* the application is used—not *how many* are using it—significantly impacts the amount of system capacity required to drive it to satisfactory performance levels.

If we can't rely on user counts to help gauge the capacity requirement of an application, then what can we use? The practical answer is often this:

Buy as much hardware as you can afford, and then create a workload management program to ensure that your workload will fit within the constraints of the capacity that you've purchased.

How can you maximize your chances that the hardware you can afford will provide sufficient capacity? For systems that will resemble others that are already in production, architects often borrow initial specifications from similar sites. For systems that are unique in their size or performance requirements, or which otherwise bear extra risk, architects must mitigate those risks by inserting tasks for configuration analysis and workload forecasting into their implementation project plans.

To determine how much hardware you'll need, test performance.

And remember, sometimes you have to be flexible. If your financial limits require you to put your workload on a budget, then put your workload on a budget.

The chief determinant of your new system's total capacity requirement is your commitment to your own system performance goals.

Commit to reducing batch job workload if your batch workload begins to jeopardize interactive response time performance.

And commit to reducing interactive workload if your interactive workload begins to jeopardize batch throughput performance.

You can reduce workload in three ways: (1) You can prohibit users from doing some of the things they want to do; (2) you can require users to reschedule some of the things they want to do to off-peak times; or (3) you can improve the efficiency of the applications they're using. Improving application efficiency makes everybody happy, and from experience I can tell you, the art of doing so can make for a happy and rewarding career.

4. The Benchmark Myth

MYTH: TPC-Cs and TPC-Ds are good capacity and performance predictors for my project.

This one is closely related to the user-count myth, because it doesn't address the different possibilities for workload variance. Industry standard benchmarks simply don't bear a close enough resemblance to most real-life workloads to provide a useful performance correlation.

However, TPC benchmarks *are* good for doing what they were designed to do: to highlight the performance differences between competing hardware or software vendors' products. And they're actually excellent at showcasing the skills and dedication of a vendor's pre-sales technical staff whose job it is to inflict embarrassment by making their benchmark run faster than all their competitors' benchmarks.

In the previous section, we explored why software configuration and use-case differences drive workload variances. These variances can make even an identical application implementation an unreliable predictor for a new system. Imagine how unreliable, then, a contrived benchmark will be as a predictor of system performance. There is no reliable conversion formula that will allow you to translate a TPC benchmark workload into an equivalent workload for a real-life application.

Use industry-standard benchmarks only for their intended purpose: to compare the performance of hardware or software vendors' products on as "level" a playing field as you will probably find.

5. The Permanent Satisfaction Myth

MYTH: Once a system is sized and tuned, it should meet the performance goals of the project.

This is an easy one. Real systems change *continuously*, demanding continual reaction and adaptation. I am amazed by the number of database application owners who express their disappointment that, in spite of the tuning we did together two years ago, their system just isn't performing up to expectations today. Never mind that in the two years since we tackled their original problems, they've grown their data size by two orders of magnitude, they've added seventeen custom reports and six custom forms, they've increased their user base by 50%, and they haven't upgraded their hardware in two years.

Performance management requires continual commitment to diagnosis and problem resolution, resource management, application optimization, and capacity planning. The commitment may cease only when the application system is decommissioned.

6. The CPU Utilization Myth

MYTH: A high-performance system needs to have CPU utilization below x%.

A very popular performance management goal, unfortunately, at sites that I've seen is, "We shall be satisfied with the performance of our system as soon as it starts running below *x*% CPU utilization." The basis for this goal is that when CPU utilization gets above *x*%—and *x* is different for different systems—response times for both online and batch jobs starts to vary wildly as queueing jobs compete for the busy CPU. But measuring the success of your system performance management program by comparing your CPU utilization to a given threshold is the wrong thing to do. I'll show you why.

First, some background. *CPU utilization* is the number of CPU busy cycles in a desired interval divided by the total number of cycles in the interval. Remember that a CPU operates in tiny time slices that I'm calling "cycles." A 300 MHz CPU exploits 300 million such cycles per second. During each cycle, the CPU is either *busy* or *not busy*—in a single cycle, there is no in-between. The concept of a CPU utilization between 0% and 100% for a given time interval exists only because we are averaging the zeros and the ones of all the cycles during that interval. For example, here is a ten-cycle interval for which CPU utilization is 60%:

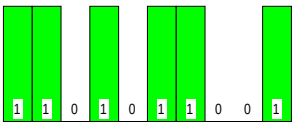


Figure 2. CPU Utilization of 60% in a 10-Cycle Interval

A *symmetric multiprocessing* (or *SMP*) system uses multiple CPUs in a single system. CPU utilization for an SMP system is a single statistic representing, again, the number of CPU busy cycles during an interval divided by the total number of cycles available in the interval. This is equivalent to the average of the individual utilization numbers chosen one from each CPU. Here is one way to achieve 60% CPU utilization on a 2-way SMP configuration:

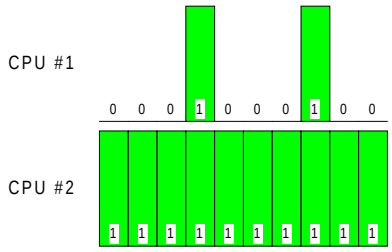
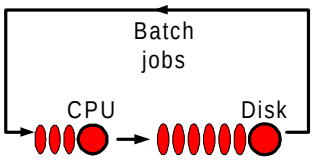


Figure 3. CPU Utilization of 60% on a 2-way SMP Machine

Note that a 2-CPU SMP machine can produce a leisurely sounding 60% utilization by having one CPU completely killed with work while another idles along at 20% utilization. Or a 2-CPU SMP system can produce 60% utilization by running both CPUs at 60% in a given interval. Single system-wide SMP CPU utilization statistics hide important information.

The secret to understanding the CPU utilization myth is contained in Figure 1. Consider the two workload classes separately for a moment. First, imagine that our system had no interactive users, only batch jobs, as follows:



Let's imagine that there is a large backlog of batch jobs that are permitted to run one at a time. In this system, a job would first request service from the CPU. Since this job is alone in the system, it would find no queue at the CPU (in the drawing, none of the "on-edge M&Ms" would be waiting for the "on-face M&M" CPU slot), so it would be serviced immediately. Upon CPU service completion, the job would request service from the disk. Again, because this job is the only job in the system, it would find no queue at the disk, and the job would be serviced immediately.

The first thing you might notice in this system is that the single job would finish completely unabated by queueing delays, with a job completion time restricted only by the raw speeds of the CPU and disk. The second thing you would probably notice is an opportunity. You have a large backlog of batch jobs waiting to be run while this single job was in the system. Yet if you had monitored the CPU utilization during the job run, you would have noticed that several CPU cycles went unused. Namely, every CPU clock tick that passed while the solitary job was receiving disk service was wasted as idle time. Likewise, you might have noticed that during every clock tick when the job was receiving CPU service, the *disk* sat idle.

Perhaps if you had submitted a second job to run in parallel with the first, you could have stuffed two batch jobs through the system in almost the same time as it took to finish just one job. Perhaps you could have crammed three or four jobs through the system without response time degradation. In actual fact, several man-years of field data collection have verified a valuable rule of thumb:

> *A single CPU can service roughly two Oracle batch jobs concurrently without degrading the time required to complete either job.*

We have tried this for a variety of different Oracle batch jobs, from reports to massive updates, from stock Oracle Applications products to exotic custom applications; and we've experimented with numerous combinations of CPUs and disks. The optimal concurrency number appears to vary only from about 1.5 jobs to about 2.5.

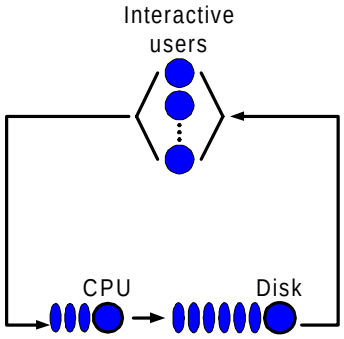
Note that for batch jobs, there is no delay—no "think time"—between when a job completes a disk I/O and either *that* job or a *new* job enters the CPU queue. It is not job think time that allows two jobs to run concurrently with minimal performance degradation; it is the phenomenon that a job leaves one resource idle while it hops off to request service from another resource.

> *In a batch-only system, CPU utilization below 100% is bad if there is a backlog of jobs to be run.*

Each CPU cycle that passes by unused is a cycle that you will never have a chance to use again; it is wasted forever. Time marches irrevocably onward.

Hopefully, you've enjoyed your reading so far, but how relevant is it to spend time thinking about a pure "batch-only" system? Even if your system is not batch-only, chances are good that it has intervals of pure batch processing, such as nighttime operations in a system that has interactive users only during the local business day. But we have more work to do.

Now let's look at the other extreme: an interactive-only system with no batch jobs at all. We shall meet in the middle to talk about mixed systems in a moment.



A key distinction between interactive users and batch jobs is that interactive users consume so-called *think time*. In this example, the users consume think time after disk service and before reentry into the CPU queue, denoted in the picture above by the angle-bracketed interactive user pool in the graph.

In Gunther's example, he modeled interactive user think time at 30 seconds. Real-life think times can range for different operations from just a few milliseconds to several minutes. For example, a user who types 60 words per minute will consume a think time of 0.004 seconds between keystrokes. On a 300 MHz CPU, each such delay accounts for 1.2 million CPU cycles. On an Internet-based email system, a user may consume several hours of think time between database calls from the server's perspective. Users attend meetings, make phone calls, write papers for conferences, and judiciously obey their biophysical requirements.

There are lots of reasons why CPU utilization would be below 100% on a system like this, even if there were a lot of concurrent users on the system. Remember that a batch-only system can handle about two concurrent batch jobs per CPU because a job leaves one resource idle while it is served by another resource. Interactive-only systems leave resources idle for this reason too, *plus* each interactive job leaves all sorts of idle resources because of think times that are gigantic in comparison to resource service times. Hence:

> *One batch job consumes the tens to hundreds of times the workload of a typical interactive user.*

How many interactive users you can put on a system before the queueing makes them miserable depends upon the speeds of the resources, the amount of work each interactive user is trying to do, and the think times.

> *A good way to estimate how many interactive users a system can handle is to use a queueing theory model like the ones described by Allen, Gunther, Jain, Kleinrock, Menascé, Tanner, and others.*

Next, we have a useful guideline for interactive-only systems that is going to take me out onto a limb:

> *To maximize performance stability for an interactive-only system, run at no more than about 70%–80% CPU utilization.*

Here's why I'm on a limb: Arnold Allen and Stephen Samson independently point out that *any* "Keep CPU utilization below *x*%" rule of thumb is of questionable virtue. When established experts say your rule of thumb is "of questionable virtue," you're on a limb. They object to the format of the rule on two separate grounds:

1. Allen points out that any such rule of thumb "overlooks the fact that it is sometimes very desirable for a computer system to run with 100% CPU utilization." [Allen (1994), 27]
2. Samson asserts that most functions of interest resemble the M/M/1 queueing function, which contains no "knee" in the curve. He states, "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." [Samson (1988)]

As I have described above during the batch-only discussion, Allen's point is one with which I agree wholeheartedly. By restricting the scope of my rule of thumb to interactive-only systems, I believe we have overcome this very valid objection. To be clear: for a batch-only system, you *want* to run at 100% utilization. For an interactive-only system, you don't. I promise: I'll address soon what happens when you mix workloads.

In response to the second point, first please remember the reason people so persistently want a rule of thumb in this format: they are trying to avoid the problem of having CPU utilization cranked up so high that their users suffer. The whole argument about whether or not there's a knee in the curve reminds me of a disturbing parable—hopefully a parable validated only by *thought* experiments—involving a frog and a pan of boiling water. The parable states that if you drop a frog into a pan of boiling water, he will quickly hop out, minimizing his burn injury. But, the story says that 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.

The intended lesson is that, because the condition of the frog's environment is never materially different from the condition in moments past, the frog is never stimulated to take bold action. But of course, in spite of the imperceptibly small degradation steps, over time the environment in the pan will deteriorate to the point at which life for the frog is no longer possible. M/M/1 curves remind me of the frog story (see Figure 4). Samson is right. There's no single utilization value on the curve as we move rightward at which the queueing time goes from being *ok* to being *not-ok*. But if you go right far enough, then you're in a region on the curve in which users are clearly suffering.

The question, then, is this: at what point does the suffering become so great that bold action is necessary—in the frog's case, a hop out of the pan; and in the system manager's case, a restricting workload? A valid "*x*%" rule of thumb would help to prevent suffering, but what is the right value of *x*? We must take care not to choose *x* so small that we waste perfectly useful CPU cycles. But if we choose *x* too large, we cause suffering.

I have two issues with Samson's line of reasoning. First, I believe there *is* a knee in the M/M/1 curve, and that a useful formal definition of *knee* is the point at which the vertical momentum of the curve begins to exceed its horizontal momentum. Figure 4 shows that even for the M/M/1 curve, there is a point near 70% utilization at which the curve begins to go "up" faster than it goes "out." Regardless of whether you will agree with me that there is a single "knee," it is certain that nobody wants to rely on a system whose queueing delays accumulate faster than the pace at which load is added to the system.

Second, Samson has based his argument upon the assumption that M/M/1 is the always right model. Although this was definitely true in the days predating SMP hardware, it is an incorrect assumption today, at least for Oracle applications. For any SMP computer implementation, the "functions of interest" resemble the M/M/*m* queueing function, for $m > 1$. The M/M/*m* model produces flatter hockey-stick shaped curves with knee-bends that become more pronounced as you use more CPUs. The more CPUs you have in your system (the larger your *m* is), the farther to the right your knee will occur, but also the more quickly your average queueing time will degrade as you add load. SMP machines with lots of CPUs scale exceptionally well for high concurrency applications, but response time performance practically falls off a cliff as utilization approaches 100%, as shown below.

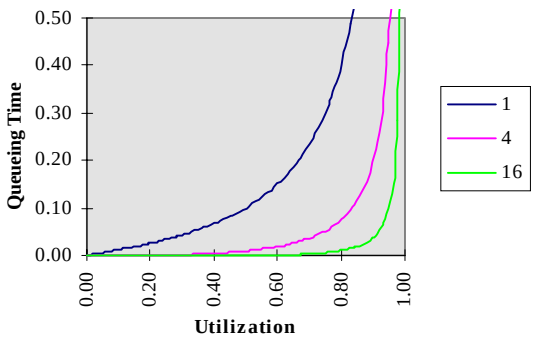


Figure 4. Queueing Time vs. Utilization for M/M/*m* Systems, for *m* = 1, 4, 16

Whether or not you can pinpoint the exact utilization beyond which queueing delays become intolerable, you definitely want to avoid running an interactive system at a CPU utilization that is "too high." I believe that the rule-of-thumb makers are right to highlight the knee-in-the-curve phenomenon for interactive-only systems, but not for systems with batch job activity.[4]

Having gained insight into batch-only and interactive-only systems, we are now prepared to tackle a performance management challenge of the highest order: mixing interactive users into a system with batch jobs. Most applications work this way. For example, Oracle Applications provide interactive services to users on the same system that runs background jobs managed by the Concurrent Manager.

The tradeoffs should now be clear. To maximize batch job throughput, you want to run your system at 100% CPU utilization. However, to optimize interactive user response times, you want some CPU utilization headroom so that users aren't exposed to high-variance queueing delays. The contradiction you see here is one reason why running an Oracle

[4] I invite the reader to race me to create a more useful rule of thumb with a threshold percentage customized for the number of CPUs in an SMP system. Here's how: If you adopt my formal definition of *knee*, then it becomes possible to *compute* the utilization at which the knee occurs. The way to do it is to compute the partial derivative $\partial w/\partial \rho$, where $w = f(\rho, \mu, m)$ is Jain's function $E[w] = E[n_q]/\lambda$ [Jain (1994), 529]. Once the derivative $\partial w/\partial \rho$ is obtained, then solve to find the value of ρ for the particular *m* of interest for which $\partial w/\partial \rho = 1$. The result would be a much more useful rule of thumb that would compensate appropriately for particular values of *m*.

Applications database is considered by many to be so difficult. The appropriate compromise is this:

> *If interactive response time constraints dominate your requirements, then determine how much CPU capacity is required by your interactive users. Then restrict the number of concurrent batch jobs so that the batch workload does not impose upon the capacity allocated to those users.*
>
> *Otherwise, if batch throughput constraints dominate your requirements, then determine how much CPU capacity is required by your batch jobs. Then restrict the number of concurrent users so that the interactive workload does not impose upon the capacity allocated to those batch jobs.*

Both of these guidelines are fast, easy, and cheap. And both prevent you from falling into the trap of letting spare CPU cycles go to waste simply because you're guided by a myth saying you should always run your system at less than 100% CPU utilization.

7. The Too-Exotic Myth

> *MYTH: Capacity planning is too intellectually exotic for me to take on.*

In the nineteenth century before A. K. Erlang discovered how to forecast wait times in a multiserver queueing network, accurate capacity planning would have been a real novelty. But today atop Erlang's original work we have a skyscraper of queueing theory and analysis techniques specialized to meet all the performance management goals of the modern computing system owner. Today we indeed have several prepackaged software tools to choose from that do all of the hard mathematics for us. Many come free with the purchase of the books that explain them [Gunther (1998), Menasacé (1994), Allen (1994)]. Lack of mathematical sophistication is certainly not a valid impediment to the modern capacity planner.

The biggest obstacle to accurate capacity planning is actually the difficulty in obtaining usable workload forecasts. Ironically, the difficulty here doesn't stem from a lack of mathematical sophistication at all, but rather from the inability or unwillingness of a business to commit to a program of performance management that includes testing, workload management, and the collection and analysis of data describing how a system is being used.

Even without sophisticated queueing theory models,[5] you can do a quite respectable job of predicting performance problems using simple linear models and rules of thumb. For example, if my system is 55% busy today and I expect my workload to double in the next year, then I need to expand my capacity this year if I want to stay ahead of my users' performance expectations.

> *Simple capacity planning is better than no capacity planning at all.*

8. Conclusion

The world is full of great ideas. But the Second Law of Thermodynamics ensures that for every great idea there are a thousand bad ones. I hope that this paper has helped you sort some of the good ideas from the bad ones and that you'll now more quickly, easily, and cheaply provide the highest levels of system performance that you wish to achieve.

References

ALLEN, A. 1994. *Computer Performance Analysis with* Mathematica. Academic Press, Cambridge MA.

ERLANG, A. K. 1917. "On the rational determination of the number of circuits," in *The Life and Works of A. K. Erlang,* 1948. BROCKMEYER, E.; HALSTROM, H. L.; JENSEN, A. (eds.). *Trans. Danish Academy of Tech. Sci.,* 216.

GUNTHER, N. 1998. *The Practical Performance Analyst.* McGraw-Hill, New York.

JAIN, R. 1991. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York.

KLEINROCK, L. 1975. *Queueing Systems Volume I: Theory.* Wiley-Interscience, New York.

KLEINROCK, L. 1976. *Queueing Systems Volume II: Computer Applications.* Wiley-Interscience, New York.

MENASCÉ, D.; ALMEIDA, V.; DOWDY, L. 1994. *Capacity Planning and Performance Modeling.* PTR Prentice Hall, Englewood Cliffs NJ.

SAMSON, S. 1988. "MVS performance legends," in *CMG '88 Conference Proceedings.* Computer Measurement Group, 148–159.

TANNER, M. 1994. *Practical Queueing Analysis.* McGraw-Hill, London.

[5] …Which are themselves perhaps sophisticated to build, but not sophisticated to use.

Acknowledgments

Thanks to Ellen Dudar for conversations leading to the wonderful idea of presenting thoughts about capacity planning this way, and for many of the ideas and much of the data behind this paper; to Espen Braekken, my group's chief capacity planning researcher and practitioner, who has ignited the engine that has turned capacity planning into a business that helps our customers; to Dominic Delmolino and my wife for helping me proofread my paper in a hurry; to Probal Shome for his encouragement; and to Mogens Nørgaard and Steen Rønsberg, my friends who I'm sure are descended from Erlang and who insisted that I propose this paper for presentation to the EOUG audience in Copenhagen.

About the Author

Cary Millsap is Vice President of the System Performance Group, which provides high performance and high availability to Oracle's largest customers worldwide. He is also responsible for the construction and global deployment of Oracle Consulting's system architecture and system management services.