
The Guerrilla Manual

1.1 Weapons of Mass Instruction

The following distillations have been extracted from the chapters in this book, my training classes of the same name, as well as my other book, *Analyzing Computer System Performance with Perl::PDQ* (Gunther 2005a).

Why Go Guerrilla? The planning horizon is now 3 months, thanks to the gnomes on Wall Street. Only Guerrilla-style *tactical planning* is crazy enough to be compatible with that kind of insanity.

Selling Prevention: Capacity planning is about prevention and someone told me “You can’t sell prevention!” Then explain the multibillion dollar dietary-supplements industry!

Why Capacity Planning is Nontrivial: Capacity planning is complicated by your brain thinking *linearly* about a computer system that operates *nonlinearly*.

Capacity planning techniques, such as the *universal scalability model* (in Sect. 1.3), help us to describe and predict these nonlinearities.

The Performance Homunculus: Capacity management is to systems management as the homunculus (sensory proportion) is to the human body (geometric proportion). See Fig. 1.1 in Chap. 1.

Capacity management can rightly be regarded as just a subset of systems management, but the infrastructure requirements for successful capacity planning (both the tools and knowledgeable humans to use them) are necessarily out of proportion with the requirements for simpler systems management tasks like software distribution, security, backup, etc. It is self-defeating to try doing capacity planning on the cheap.

Self Tuning Applications: Self-tuning applications are not ready for prime time. How can they be when human performance experts get it wrong all the time!?

Performance analysis is a lot like a medical examination, and medical *Expert Systems* were heavily touted in the mid 1980s. You do not hear about them anymore. And you know that if it worked, HMOs would be all over it. It is a laudable goal but if you lose your job, it will not be because of some expert performance robot.

Squeezing Capacity: Capacity planning is not just about the future anymore.

Today, there is a serious need to squeeze more out of your current capital equipment.

When Wrong Is Right: Capacity planning is about setting expectations. Even *wrong* expectations are better than no expectations!

Planning means making predictions. Even a wrong prediction is useful. It means either (i) the understanding behind your prediction is wrong and needs to be corrected, or (ii) the measurement process is broken somewhere and needs to be fixed. Start with a SWAG. Next time, try a G. If you aren't making iterative predictions throughout a project life cycle, you will only know things are amiss when it is too late!

The Overengineering Gotcha: Hardware *is* cheaper today, but a truckload of PCs will not help one iota if all or part of the application executes single-threaded.

My response to the oft-heard platitude: "We don't need no stinkin' capacity planning. We'll just throw more cheap iron at it!" The *capacity* part is easy. It is the *planning* part that is subtle.

Network Performance: It is never the network!

If the network is out of bandwidth or has interminable latencies, fix it! Then we will talk performance of your application.

Can't Beat This! If the measured round-trip times (RTTs) for an application produce a relatively flat or concave curve (like that in Fig. 1.5) as a function of increasing load, SHIP IT! Only if you do not understand basic queueing theory would you press on in spite of such data.

Modeling Errors: When I am asked, "But, how accurate are your performance models?" my canonical response is, "Well, how accurate are your performance *data*!?"

Most people remain blissfully unaware of the fact that *all* measurements come with errors, both systematic and random. An important capacity planning task is to determine and track the magnitude of the errors in your performance data. Every datum should come with a " \pm " attached (which will then force you to put a number after it).

Data are Not Divine: Treating performance data as something divine is a sin.

Data comes from the devil, only models come from God.

Just Digging the Hole Deeper: Busy work does not accrue enlightenment.

Western culture too often glorifies hours clocked as productive work. If you do not take time off to come up for air to reflect on what you are doing, how are you going to know when you are wrong?

Little Things: Little's law means a lot! Learn $Q = XR$ by heart.

I use it almost daily to *cross-check* that throughput and delay data are consistent, no matter whether those data come from measurements or models. More details about Little's law can be found in (Gunther 2005a, Chap. 2), *Analyzing Computer System Performance with Perl::PDQ*. Another use of Little's law is calculating service times, which are notoriously difficult to measure directly. See the Rules of Thumb in Sect. 1.2.

Bigger is Not Always Better: Beware the SMP wall!

The *bigger* the symmetric multiprocessor (SMP) configuration you purchase, the *busier* you need to run it. But only to the point where the average run-queue begins to grow. Any busier and the user's response time will rapidly start to climb through the roof.

If They Snooze, You lose: Spend as much time on developing the presentation of your capacity planning conclusions as you did reaching them.

If your audience does not get the point, or things go into the weeds because you did not expend enough thought on a visual, you just wasted a lot more than your presentation time-slot.

Bottlenecks: You never *remove* a bottleneck, you just shuffle the deck.

Benchmarks: All benchmarks represent institutionalized cheating.

Consolidation: Gunther's law of consolidation: Remove it and they will come!

Control Freaks Unite! Your own applications are the last refuge of performance engineering.

Control over the performance of hardware resources, e.g., processors and disks, is progressively being eroded as these things simply become commodity black boxes, viz., multicore processors and disk arrays. This situation will only be exacerbated with the advent of Internet-based application services. Software developers will therefore have to understand more about the performance and capacity planning implications of their designs running on these black boxes. (see Sect. 1.3)

Best Practices: Best practices are an admission of failure.

Copying someone else's apparent success is like cheating on a test. You may make the grade but how far is the bluff going to take you?

1.2 Capacity Modeling Rules of Thumb

Here are some ideas that might be of help when you are trying to construct your capacity planning or performance analysis models.

Keep It Simple: A performance model should be as simple as possible, but no simpler!

I now tell people in my GCaP classes, despite the fact that I repeat this rule of thumb several times, you *will* throw the kitchen sink into your performance models; at least, early on as you first learn how to create them. It is almost axiomatic: the more you know about the system architecture, the more detail you will try to throw into the model. The goal, in fact, is the opposite.

More Like The Map Than The Metro: A performance model is to a computer system as the BART map (Fig. 1.2) is to the BART rail system.

The BART map is an abstraction that has very little to do with the physical train. It encodes only sufficient detail to enable transit from point A to point B. It does not include a lot of irrelevant details such as altitude of the stations, or even their actual geographical proximity.

A performance model is a similar kind of abstraction.

The Big Picture: Unlike most aspects of computer technology, performance modeling is about deciding how much detail can be *ignored*!

Look for the Principle: When trying to construct the performance representation of a computer system (which may or may not be a queueing model), look for the principle of operation. If you cannot describe the principle of operation in 25 words or less, you probably do not understand it yet.

As an example, the principle of operation for a time-share computer system can be stated as: *Time-share gives every user the illusion that they are the ONLY user active on the system.* All the thousands of lines of code in the operating system, which support time-slicing, priority queues, etc., are there merely to support that illusion.

Guilt is Golden: Performance modeling is also about spreading the guilt around.

You, as the performance analyst or planner, only have to shine the light in the right place, then stand back while others flock to fix it.

Where to Start? Have some fun with blocks; *functional blocks*!

One place to start constructing a PDQ model is by drawing a *functional block diagram*. The objective is to identify where time is spent at each stage in processing the workload of interest. Ultimately, each functional block is converted to a queueing subsystem like those shown above. This includes the ability to distinguish sequential and parallel processing. Other diagrammatic techniques e.g., UML diagrams, may also be useful. See (Gunther 2005a, Chap. 6).

Inputs and Outputs: When defining performance models (especially queueing models), it helps to write down a list of INPUTS (measurements or estimates that are used to parameterize the model) and OUTPUTS (numbers that are generated by calculating the model).

Take Little's law $Q = XR$, for example. It is a performance model, albeit a simple equation or operational law, but a model nonetheless.

All the variables on the *right* side of the equation (X and R) are INPUTS, and the single variable on the *left* is the OUTPUT. A more detailed discussion of this point is presented in (Gunther 2005a, Chap. 6).

No Service, No Queues: You know the restaurant rule: “No shoes, no service!” Well, this is the PDQ modeling rule: no service, no queues. In your PDQ models, there is no point creating more queueing nodes than you have measured service times for.

If the measurements of the real system do not include the service time for a queueing node that you think ought to be in your PDQ model, then that PDQ node cannot be defined.

Estimating Service Times: Service times are notoriously difficult to measure directly. Often, however, the service time can be calculated from other performance metrics that are easier to measure.

Suppose, for example, you had requests coming into an HTTP server and you could measure its CPU utilization with some UNIX tool like *vmstat*, and you would like to know the service time of the HTTP Gets. UNIX will not tell you, but you can use Little’s law ($U = XS$) to figure it out. If you can measure the arrival rate of requests in Gets/sec (X) and the CPU %utilization (U), then the average service time (S) for a Get is easily calculated from the quotient U/X .

Change the Data: If the measurements do not support your PDQ performance model, change the measurements.

Closed or Open Queue? When trying to figure out which queueing model to apply, ask yourself if you have a finite number of requests to service or not. If the answer is yes (as it would be for a load-test platform), then it is a *closed* queueing model. Otherwise use an *open* queueing model.

Opening a Closed Queue: How do I determine when a closed queueing model can be replaced by an open model?

This important question arises, for example, when you want to extrapolate performance predictions for an Internet application (open) that are based on measurements from a load-test platform (closed).

An open queueing model assumes an infinite population of requesters initiating requests at an arrival rate λ (lambda). In a closed model, λ (lambda) is approximated by the ratio N/Z . Treat the thinktime Z as a free parameter, and choose a value (by trial and error) that keeps N/Z constant as you make N larger in your PDQ model. Eventually, at some value of N , the OUTPUTS of both the closed and open models will agree to some reasonable approximation.

Steady-State Measurements: The steady-state measurement period should on the order of 100 times larger than the largest service time.

Transcribing Data: Use the timebase of your measurement tools. If it reports in seconds, use seconds, if it reports in microseconds, use microseconds. The point being, it is easier to check the digits directly for any

transcription errors. Of course, the units of ALL numbers should be normalized before doing any arithmetic.

Workloads Come in Threes: In a mixed workload model (multiclass streams in PDQ), avoid using more than three concurrent workstreams whenever possible.

Apart from making an unwieldy PDQ report to read, generally you are only interested in the interaction of two workloads (pairwise comparison). Everything else goes in the third (AKA “the background”). If you cannot see how to do this, you are probably not ready to create the PDQ model.

1.3 Scalability on a Stick

The following points explain how to quantify notions of scalability:

1. A lot of people use the term “scalability” without clearly defining it, let alone defining it quantitatively. Computer system scalability must be quantified. If you cannot quantify it, you cannot guarantee it. The *universal law of computational scaling* provides that quantification.
2. One the greatest impediments to applying queueing theory models (whether analytic or simulation) is the inscrutability of service times within an application. Every queueing facility in a performance model requires a service time as an input parameter. As noted in Sect. 1.2, *No service time, no queue*. Without the appropriate queues in the model, system performance metrics like throughput and response time, cannot be predicted. The *universal law of computational scaling* leapfrogs this entire problem by NOT requiring ANY low-level service time measurements as inputs.

1.3.1 Universal Law of Computational Scaling

The relative capacity $C(N)$ (the dashed line in Figs. 6.3 or 6.5) is given by:

$$C(N) = \frac{N}{1 + \alpha N + \beta N(N - 1)}$$

where N is either:

1. The number of users or load generators on a fixed hardware configuration. In this case, the number of users acts as the independent variable while the CPU configuration remains constant for the range of user load measurements.
2. The number of physical processors or nodes in the hardware configuration. In this case, the number of user processes executing per CPU (say, 10) is assumed to be the same for every added CPU. Therefore, on a 4 CPU platform you would run 40 virtual users.

with α the *contention* parameter, and β the *coherency-delay* parameter. The latter accounts for the retrograde throughput seen in Fig. 6.3, for example.

- The objective of using $C(N)$ is *not* to produce a curve that passes through every data point. That is called curve fitting and that is what graphics artists do with splines. As von Neumann said, “Give me four parameters and I will fit an elephant. Give me five and I will make its trunk wiggle!” (At least I only have two).
- When the coherency-delay parameter vanishes i.e., $\beta = 0$, $C(N)$ reduces to Amdahl’s law, as expected. See Eq.(4.15) in Chap. 4.

1.3.2 Areas of Applicability

This universal model has wide spread applicability. Some areas are:

- Modeling such effects as VM thrashing, and cache-miss latencies.
- Modeling disk arrays, SANs, and multicore processors.
- Modeling certain types of network I/O.
- User-load performance testing is one of the most common applications.
- Using it in combination with measurement tools like LoadRunner, Benchmark Factory, etc.

That is why $C(N)$ is called *universal*.

1.3.3 How to Use It

Virtual Load Testing: The universal model for $C(N)$ allows you take a sparse set of load measurements (4–6 data points) and determine how your application will scale under larger user loads than you may be able to generate in your test lab. This can all be done in a spreadsheet like Excel. See, e.g., Fig. 1.3 in Chap. 1 and Fig. 5.3 in Chap. 5.

Detecting measurement problems: $C(N)$ is not a crystal ball. It cannot foretell the onset of broken measurements or intrinsic pathologies. When the data diverge from the model, that does not automatically make the model wrong. You need to stop measuring and find where the inconsistency lies.

Performance Heuristics: The relative sizes of the α and β parameters tell you respectively whether contention effects or coherency effects are responsible for poor scalability.

Performance Diagnostics: What makes $C(N)$ easy to apply also limits its diagnostic capability. If the parameter values are poor, you cannot use it to tell you what to fix. All that information is in there alright, but it is compressed into the values of those two little parameters. However, other people, e.g., application developers (the people who wrote the code), the systems architect, may easily identify the problem once the universal law has told them they need to look for one.

<http://www.springer.com/978-3-540-26138-4>

Guerrilla Capacity Planning

A Tactical Approach to Planning for Highly Scalable
Applications and Services

Gunther, N.J.

2007, XX, 253 p. 108 illus., Hardcover

ISBN: 978-3-540-26138-4