Implementing Performance Engineering

Presentation #4015

Erik Wettersten
Panoptic Solutions Corporation
Box 600 856 Wapiti Drive
Fraser CO, 80442-0600
970-726-5676
pano@ix.netcom.com

Abstract:

Avoiding performance problems is obviously preferable to having to fix them. However, the complexity of client-server applications combined with the always-present promise of faster hardware, has created a tendency to wait until code is proven to be of poor performance, then fixing it with hardware or brutal last-minute changes. This habit, in turn, results in enormous costs, schedule slippage and even cancellations.

To counter this reactive pattern, Software Performance Engineering (SPE) evolved, attempting to understand the performance of software earlier in the lifecycle – thereby giving designers the opportunity to see the effect on performance of design changes while they are able to do something about it. SPE found a home in situations where the hardware commitment was not flexible (and very expensive), but has not caught-on in mid-range systems. This is largely because it requires too much discipline to be implementable and is therefore attempted by only a few dedicated individuals with an aptitude for number crunching.

Knowing performance information when it is most needed is the ideal situation, but if current methods are too complicated to be practiced, they are not valuable. In an effort to find a workable solution, this paper will take SPE principles and make them implementable, balancing these practices with other performance alternatives and thus creating a methodology to efficiently engineer performance *into* an application. This is Performance Engineering. In order to strike an equilibrium between SPE and hardware alternatives, this paper will first demonstrate the necessity to wean development efforts off the "fix it later" approach and then move toward an implementable Performance Engineering methodology. A methodology that does not abandon other choices, but provides a framework for designers, developers, and architects to understand the performance impact of decisions while there is an opportunity to act on the information.

1. Introduction

1.1 The Problem

"Make it run, make it pretty, make it fast" – **the *unwritten*, *unspoken*** credo of software development.

In the history of computer software development, developers have learned that functionality comes first, with usability, reliability, performance, etc. falling at a distant second. The problem is when it comes time to 'fix' these secondary priorities, particularly performance, very little can be done about it unless a major (costly) redesign is attempted – that or the more traditional approach of tuning to make it as good as possible and buying more hardware to do the rest.

This "fix it later" approach does not make sense, but is still practiced – particularly in the client-server world where the distributed, complex nature of applications makes it that much more difficult to anticipate the performance impact of design decisions until it is too late. Performance, once was achieved by intuition, is now neglected due to complexity. As scientists, the architects and designers should know better – and probably most do – but without the fundamental practices and incentives of *Performance Engineering*, nothing will improve.

1.2 The Cost

Assuming that there will be a performance problem (almost by definition, performance could always be better), the next issue centers around the cost of not addressing it. In addition to the obvious cost of user dissatisfaction, the costs of not dealing with performance proactively include:

1. the **increased expense** of additional hardware and maintenance costs,
2. the **lost profit** due to a delayed product, and
3. the **increased stress** of trying to capacity plan without performance information and pull off last-minute performance miracles.

Summarizing the extent of this cost, one estimate [Infoworld, 95] is that "Eighty percent of all client/server projects have to be redesigned because of performance issues, not because they didn't meet the [functional] needs of the users." That should be reason enough for anyone.

1.2.1 The Future

In the future, the issue of performance will become even more important. With the advent of code generators, code is developed more quickly and the produced code is more usable (the "make it run and make it pretty" factors), the primary differentiator will shift away from functionality and shift toward performance. This is the same transition that hardware has already gone through. *Componentized modules* with *known performance characteristics* will follow the industry's move towards object-oriented development. The movement from custom application development to the integration of 'canned' products will mean that a greater understanding will be necessary in order to create modules that both perform optimally and can be combined with other modules into an ***optimal system***. This dictates that the performance of a system must be known *before* it is put into production.

It would be hard to imagine a car that is built ***then*** driven around to see how fast it will go and what kind of mileage it will get. When engineers design a car, these characteristics are part of the design and design decisions are made accordingly. The carburetor matches the engine which matches the transmission, etc. so that the car *as a whole* performs optimally – instead of being an sub-optimal collection of optimized parts. Not designing performance *into* code makes about as much sense as allowing each component of a car to be created independent of the final objective.

1.3 Why hasn't it been Fixed Yet?

If a problem exists and it is going to cost a significant amount of resources, but is still isn't being addressed, there must be a reason. Actually, there are several reasons, but fundamentally, there are really two:

1. **Obviousness** – It's very clear to the developer when functionality is ***wrong*** but it's not obvious when *performance* is ***wrong*** until the application is finally assembled. This is the "in your face" effect – if it cannot be seen, must not be there. Client server performance issues do not automatically show up until very late in the product lifecycle – unless they are sought out intentionally.
2. **Complexity** – By definition, client-server applications mean that duplicating the final product requires multiple nodes and multiple transactions. This added complexity of makes it difficult to

look for problems and equally difficult to interpret the effects of development decisions on the final product.

1.4 The Paradox

The problem with performance – especially client server performance – is that the *easiest* time to find out when and where performance problems exist also happens to be the time when the least can be done to combat problems. This effect, call it "applied procrastination", would normally be discouraged but in effect is ***encouraged*** by the very nature of the problem: problem prevention is a thankless act, but problem resolution is a hero-maker – even when the prevention is far superior to the resolution. Consider the following:

1. After discovering that an application will not perform at the desired level, a performance "swat team" is formed. After a couple months of streamlining, reducing functionality, and increasing hardware, performance is brought close enough to the specification to ship.
2. The swat team and development teams are celebrated for their Herculean effort and saving the project from certain fate and embarrassment.

Conversely, suppose a structured approach to performance engineering was taken, whereby the response time issue surfaced while the code was being developed and the addressed at that time. The resulting product is one that meets the original specification thanks to slightly more up front work in the form of definitions. The product ships, there are no heroes, but the end result is on-time, did not require additional hardware or consulting, and furthermore will not have the maintainability and reliability issues of the former effort due to all of the ad-hock efforts that went in at the last minute. Not to discredit the eleventh hour work of red-eyed computer heroes, but it should not be necessary.

In these two cases, clearly the latter was more desirable, but the former was the one that received all of the positive reinforcement – all for improving something that was not supposed to be a problem. *And even though it was at a significantly higher cost!* This is analogous to being unhealthy; dramatic recoveries and incredible cures are celebrated, but the even greater success of prevention is hardly acknowledged – regaining health is not, ***ever***, as good as being healthy, it's just more dramatic.

1.5 Bad Habits and Beliefs

The "applied procrastination" approach to performance has nurtured a collection of nasty habits and myths which in turn contribute to the self fulfilling prophecy of bad performance. A couple of these habits and myths include the following:

Belief 1: Performance can be added later – This is like saying a car built without the plan can be made efficient or fast by adjusting the carburetor and removing the antenna. Tuning is **not** designing and *performance is the effect of design*.

Belief 2: Faster hardware makes performance issues "go away" – Although it is true that faster hardware *can* be used to increase performance (at a price) – it has a cruel side-effect *because it can also **cause** bad performance*. This is because of three reasons:

1. The knowledge that fast hardware can be used *encourages* bad coding practices. This is unavoidable human nature, just like the promise of an upcoming raise *increases* current spending.
2. The increased concurrency enabled by faster hardware makes performance more susceptible to variations in workload. Faster development environments backfire when resources have to be shared. As an analogy, faster automobiles help **create** traffic jams because people know they *can* travel 20 miles in 20 minutes. This capability increases the "burstiness" of traffic (i.e., everyone leaves at 7:40 to be at work at 8:00), which in turn, increases contention for shared resources..

3. Faster hardware allows us to create more complex systems which inherently create performance problems which are more difficult to solve [Smith, 90].

Furthermore, solving a problem with hardware also raises the financial stakes of a project. Consider the cycle of:

1. create a poor performing application,
2. get by on performance by increasing the hardware investment at the last minute,
3. add functionality without proper attention to performance in the next release,
4. increase hardware to fix performance problems incurred by the added functionality,
5. repeat steps 3 and 4 indefinitely.

This system is analogous to the blackjack theory of "doubling up" whereby the gambler doubles their bet each time they lose – thus winning all their losses back when they finally do win[1]. The problem is that neither gamblers nor businesses have unlimited resources. With either method, the investor eventually finds themselves faced with a situation where a huge investment has been made, and the only way out of it is to make another, even larger, investment. Sooner or later, a company will simply not be able to afford the next investment – thus canceling a project that has exhausted the company's research investment for several times the original budget. This windfall for hardware vendors can completely tap a company's IT budget – *and it does happen!*

The underlying consideration is that *hardware fixes a symptom*, not a problem; fast hardware is nothing more than a *design alternative* – it is not a cure-all.

Belief 3: Paying later is <u>better</u> than paying now – This is probably the most subtle and most dangerous of all. Developers have learned that it's easier to ask for forgiveness than it is to ask for permission; this has a negative impact to performance design. It is very difficult to budget for performance at the onset of a project, so instead, development waits until functionality is done and then more resources are requested to go back and fix performance issues. Even though the budget is exhausted, since the project is nearing functional completion, it's very difficult for management to say "no" to additional resources – no matter how much – since they have already made such a large investment. What could have been addressed for a fraction of the cost at the onset is ***encouraged*** to be addressed at several times the cost after the fact.

2. A Different Approach

2.1 Generic Problem Solving

Instead of looking for a silver bullet to solve this problem, consider how most day to day problems get solved. In its most simplified form, problem solving is a matter of:

1. Set a goal,
2. Take action towards that goal,
3. Obtain feedback as to the effect of the actions towards that goal, and
4. Repeat steps 2 and 3 until the goal is reached.

Although this is embarrassingly basic, the really embarrassing part is that it is forgotten in the quest for silver bullets. Applying this basic formula to software engineering with the intent of achieving

[1] Also called "Martingale", an example of a double up progression would be: starting with one unit and doubling up until the player finally wins; after five losses this would yield 32 - (1+2+4+8+16) = 32 - 31 = 1 [Humble 87]. Casinos don't mind, and in fact *like* this practice, because they know that eventually the player won't be able to place the last bet (due to limited resources or house limits) and lose their entire bankroll.

performance, yields ***performance engineering***. Specifically, performance engineering (PE) means the **application** of:

1. setting a *quantifiable* performance goal,
2. designing and developing code with a performance objective,
3. quantifying the effect of code design(s) relative to the performance goal, and
4. repeating.

The real key here is ***applying*** this formula – that is *implementing the method* and *utilizing the technology*.

2.2 The PE Methodology

The fundamentals of the PE methodology take the goal and action steps above and defines two criteria for each:

GOALS:

1. Know the intended environment.
2. Know the performance criteria.

ACTION:

1. Always know ***how*** the system is performing.
2. Always know ***why*** the system performs in this way.

The methodology endorsed by performance engineering is *constant knowledge* with steadily *increasing accuracy*. In the early stages of development, there is a known performance expectation but it is understood that this expectation is subject to a high degree of error. To this end, the performance estimates are qualified to a number of assumptions and are usually "book-ended" as a best-case/worst-case possibility. As the stages of development proceed, there is really very little *new* information, but rather a lot of refinement to *old* information – estimations are replaced by measurements and usage assumptions become more understood.

Balancing PE with last minute hardware adjusting and tuning should handle 80% of the issues ***before*** final release, then the last minute "bag of tricks" can be used to handle the remaining 20% that were not practical during development. In this way, performance becomes the effect of design, and hardware is used to scale.

3. Implementing Performance Engineering

The remainder of this paper will illustrate an approach for implementing performance engineering by describing a methodology that couples the "best available measurements" with the "best available extrapolations". This methodology concentrates on:

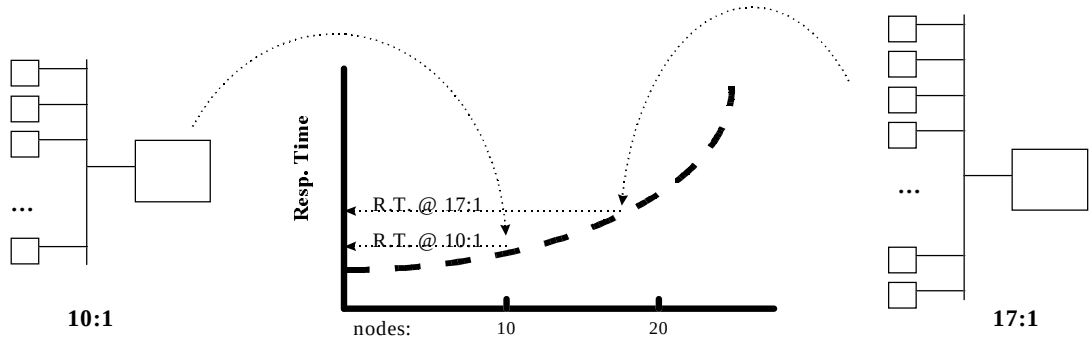
- Understanding the performance of the system as a whole by focusing on overall capacity first, knowing that response time cannot be brought under control until the capacity is under control.
- Focusing on the service time of operations so comparisons and extrapolations can be made accurately
- Using rules of thumb and modeling to equate service time to capacity and cost.

In other words this approach takes whatever information and tools are available and uses that information to systematically "shift the timeline" – that is, be able to see the ***effects*** of performance decisions and the ***cause*** of performance problems while there is still time to act.

3.1 Concepts

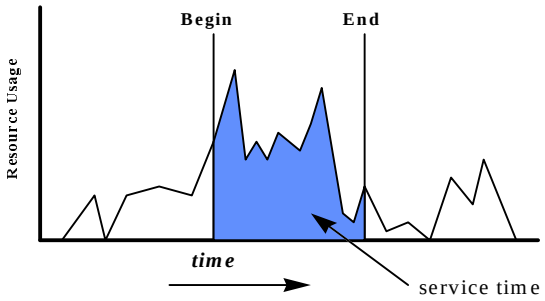
Following are the terms and concepts used in performance engineering and in this paper.

- **Response Time** – Service time plus delay(s). This how the end-user "views" performance
- **Response Time Model** – The expected response time as the node to server ratio changes. This provides a means to visualize the performance of a system as a whole.
- **Response Time Curve "Knee"** – The area on the response time curve where the node to server ratio is maximized, but contention has not yet made response time unacceptable. A system must be operating to the left of this "knee" from a capacity standpoint before user perceived response time issues can be addressed.



Expected Response Time As Node To Server Ratio Increases

- **Contention** – The result of multiple requests requiring the same resource. This is what causes the "knee" in a response time curve.
- **Node to Server Ratio** – This is the number of end users that can simultaneously use a shared computer resource, and is a significant indication of relative system cost (the higher the ratio, the more cost-effective).
- **Service Time** – This is the time a resource takes to process a request. This is basically the "area under the curve" if resource percentages were graphed over time. Contention for service time is what determines where the "knee" of the response time curve will occur, and is the most consistent means of comparing relative performance.



Since the goal of PE is to balance hardware costs with performance – which is largely driven by node to server ratios – most discussion herein will focus on the effects of service time and use response time curves to represent the performance of the system as a whole.

3.2 Goal

Goal setting is the most important part of any endeavor. The problem with the goal setting from a performance standpoint is that it is typical to set goals that are too amorphous to be useful. Consider

the following response time goal: "*Responses must take less than 4 seconds*". That would seem pretty strait forward, but in reality is almost meaningless as it does not specify the two ingredients of a performance goal, namely the **conditions** and the **criteria**.

3.2.1 Conditions

Conditions describe the environment and consist of three components:

1. **Load** – How active and efficient are the users? (How many operations per unit of time will be processed by the computer?)
2. **Mix** – How often will the different parts of the application be run? (Or how often will each application of a multiple application environment be run?)
3. **Target Configuration** – What is the intended node to server ratio, and how will the machines likely be configured. This is an important step as it discourages the "fix it with hardware approach".

Without these conditions, there is no way to proactively address performance issues.

3.2.2 Criteria

The criteria should specify the following two components:

1. **Metrics** – The above statement assumes that 100% of the responses should be within four seconds; in a distributed environment, it is nearly impossible to guarantee **all** responses within a time frame because of contention for shared resources. Because contention causes an exponential response time distribution, guaranteeing 99% response time criteria is ***ten times*** as difficult as guaranteeing 90% response time at the same level (and 99.9% is another ten times more difficult than 99%!).
2. **Circumstances** – Is the above metric accumulated over all responses or just at peak? Are the criteria relaxed under special circumstances such as a failover?

3.3 Specifying the Goal

Ideally, developers could develop to a goal that defines both the response time criteria and conditions exactly. This is not practical since conditions are typically not that well understood (especially for new development). To compromise, goal setting (especially early-on) should center on the typical conditions – that is, the 80%-tile. Find out what the likely conditions are and use that as the basis for design assumptions. The same is true for response time; find the response times that are important and define them. It is acceptable to have a catch-all such as "of the slowest responding transaction, no more than 10% should take more than XXX seconds [under YYY conditions]". Notice that the transaction is still defined (the "slowest one") – if it had said "of all transactions" it would be an easy target to hit since there is probably a very large percentage of fast transactions (i.e., 90%) which makes the specification a mute point.

Thus consider the following example:

Conditions

1. Users are assumed to be 90% efficient (i.e., actively working 54 minutes out of 60 on average) during peak load and all response time criteria are assumed to be under peak working conditions. This equates to 25 operations per user, per hour.
2. The operation mix is as follows:

Operation	*Spring%*	*Fall%*
Operation A	15	17
Operation B	40	68
Operation C	45	15

3. The system is to support a 30:1 user to server ratio under normal conditions, and a 45:1 ratio under failover conditions.

In keeping with the pattern of increasing granularity over time, it is understood that the above operation mix may be broken down further be defining each operation more specifically.

Criteria

1. "Over any one-hour period, responses for the slowest transaction shall be within 10 seconds 90% of the time."
2. "Under a failover situation, the system must operate with a 50% increase in the number of users and response time criteria is relaxed to 80% in the above criteria."

The combination of good conditions and clear criteria create a true goal. By having such a target, development can proceed with an achievable performance *objective* – instead of a vague performance *intent*.

3.3.1 Goal Refinement – Converting Conditions to Workloads

With a goal defined (one that includes operating conditions), the next step is to convert operating conditions and assumptions into work loads which will be used to drive (i.e., benchmark) the application. There should be one workload for each of the above defined transactions with each workload typifying the normal usage of that operation. These workloads are essentially a "usage flow" and can be thought of as what the end user does ***to the computer.***

If it is not possible to describe a normal operation with just one workload (e.g., sometimes an operation accesses a large volume of data, but most times just a small amount), then it should be broken out further. For example if "Operation B" above accesses a large amount of data some of the time and a small amount at other times, then it should be broken into two operations, such as "Operation B-small" and "Operation B-large" with two representative workflows. This might produce a mix like the following:

Operation	*Spring%*	*Fall%*
Operation A	15	17
Operation B-small	28	42
Operation B-large	12	26
Operation C	45	15

In general, such a refinement should take place for any operation that has characteristically different behavior (such as the above example) or represents an inherently large percentage of the workload mix (since it will have such a great effect, accuracy and therefore refinement, are more important).

3.4 Action

The next step to achieving performance is to obtain resource service time for each workflow. A lot of extrapolations can be made from service time data (as it ultimately determines the performance of the system) so it is crucial that this information be very accurate; a small error in data can result in a large error when the data is extrapolated to the real world. If accuracy can not be assured then

multiple measurements representing a best/worst/likely case should be used to "book-end" the range of probable performance.

Collecting service time data is simply a matter of running a workflow in a controlled environment with an appropriate measurement tool that collects service time (or estimating the impact in the early stages). Ideally, the workflows can be scripted using a playback or simulation tool which assure consistency but this can also be done via written scripts that the tester follows verbatim each time[2]. The performance data collected during this procedure would include information about the various resources, particularly CPU, Disk, memory, and LAN traffic.

Continuing with the above example, a collection of service time performance data (CPU, and disk only) might look like the following:

	CPU-seconds	Disk-seconds	Elapsed Time
Op A	1.61	1.69	3.9
Op B-small	1.54	1.01	4.2
Op B-large	1.89	1.54	4.9
Op C	1.82	1.02	3.6

This matrix reflects the service times for the CPU and Disk resources of the operations in question. Weighting this data with the anticipated usage patterns defined in the operating conditions yields the following:

	Spring					***Fall***				
	Percent	**Wtd. CPU**	**CPU %**	**Wtd Disk**	**Disk %**	**Percent**	**Wtd. CPU**	**CPU %**	**Wtd Disk**	**Disk %**
Op A	15.0%	0.2415	14.1%	0.2535	21.5%	17.0%	0.2737	16.2%	0.2873	22.7%
Op B-small	28.0%	0.4312	25.1%	0.2828	24.0%	42.0%	0.6468	38.4%	0.4242	33.5%
Op B-large	12.0%	0.2268	13.2%	0.1848	15.7%	26.0%	0.4914	29.2%	0.4004	31.7%
Op C	45.0%	0.8190	47.7%	0.4590	38.9%	15.0%	0.2730	16.2%	0.1530	12.1%
Ave Wtd Usage Per Op:		**1.7185**		**1.1801**			**1.6849**		**1.2649**	

This matrix now reflects the anticipated load based on both the usage assumptions and measured data and will be used to establish the "*how*" and "*why*" of the system's performance.

3.5 Feedback

In order to make the collected performance information useful, it has to be applied to the application. There are two facets to this:

1. **Projected performance** – using the data to track the overall performance of the system (the ***"how"***); this effectively "shifts the timeline" by giving an indication of the future performance.
2. **Performance improvements** – using the data to find where the most improvements can be gained (the "***why***").

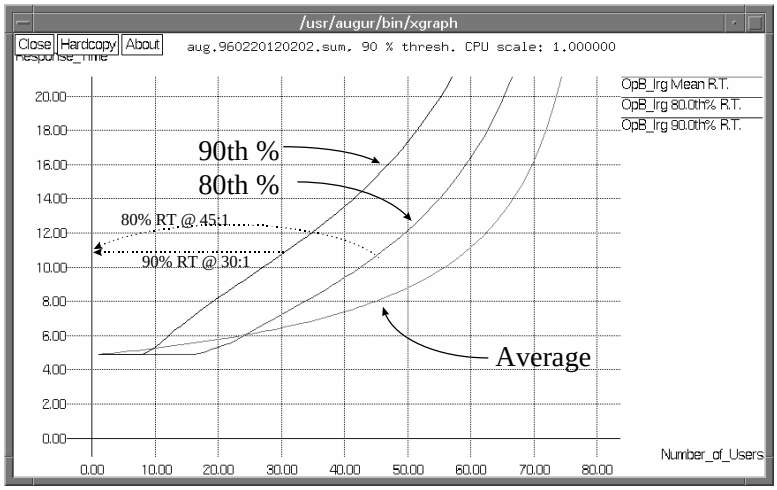
Both of these involve the combination of the user environment with the collected data. This can be achieved via spreadsheet calculations, but for practical reasons, can be simplified through the use of modeling tools. For comparative purposes, this paper will show the results of both methods.

3.5.1 Projected Performance (the "*How*")

It is recommended to use available performance modeling software to projected performance – such tools allow multiple scenarios to be examined quickly and also provide performance anticipation

[2] Typically, the testing will be done via user-driven scripts in the early stages, but will utilize automated tests once the application matures and becomes more stable.

capabilities that cannot be achieved by other means. A response time model for operation "Op B-large" ("*OpB_lrg*" in the graph) under the Spring usage assumptions is as follows:



As the model indicates, with 30 users on the system, 90% of the responses for operation "OpB_lrg" would NOT occur within the prescribed 10 seconds. Under the failover situation whereby there are 45 users, it would appear as though 20% of the responses would also require 11 or more seconds. Thus, the "look into the future" indicates that the performance criteria will not be met.

3.5.2 Projecting Without Analytical Models

In the absence of modeling tools, performance can also be targeted using a "50/40/30" rule of thumb. This means calculating the aggregate impact on resources and keeping the amount below 50%, 40%, and 30% of available CPU, disk, and LAN (respectively). As the utilization approaches the 50/40/30 threshold, variability begins to be an issue, so systems that have tight variability constraints (such as the current example) would be advised to stay somewhat below these limits.

Although this technique will not display the performance deterministically, it does allow feedback regarding the expected capacity. The above example would be calculated as follows (using CPU and Disk only):

					Normal		Failover
CPU Utilization:							
	1.7185 *secs/op*	750 *ops/hr*	0.000278 *hrs/sec*	=	**35.80%**	***Spring***	**53.70%**
	1.6849 *secs/op*	750 *ops/hr*	0.000278 *hrs/sec*	=	**35.10%**	***Fall***	**52.65%**
Disk Utilization:							
	1.1801 *secs/op*	750 *ops/hr*	0.000278 *hrs/sec*	=	**24.59%**	***Spring***	**36.88%**
	1.2649 *secs/op*	750 *ops/hr*	0.000278 *hrs/sec*	=	**26.35%**	***Fall***	**39.53%**

Thus, from this basic information, it is concluded that the desired node to server ratio is probably obtainable, but variability will be a potential issue, particularly in failover situations. This highlights both the advantage and disadvantage of the 50/40/30 technique – while it is very simple, it is not nearly as accurate as modeling and can only provide an ***indication*** as to when average response time begins to noticeably degrade.

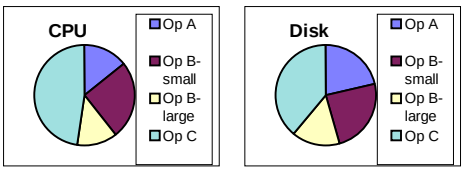
Working backwards from the 50/40/30 rule of thumb, the user can estimate the maximum number of nodes before the threshold is reached. Solving for the desired target in this example yields:

Target	*Spring*	*Fall*
Max nodes before CPU reaches 50%	41.9	42.7
Max nodes before Disk reaches 40%	48.8	45.5

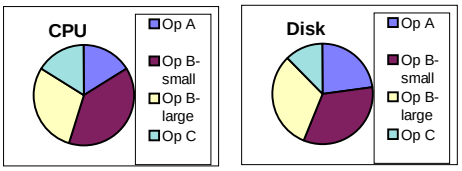
Using this data to document the performance of the system as a whole, it would be said that under the anticipated workload for the spring, the system's response time will experience noticeable variability with 42 users, and 43 users in the fall. Again, this would be for a system with only modest variability constraints – in some cases (including this example) it would be wise to target a more conservative rule of thumb down to even 40/30/20.

3.5.3 Performance Improvements (the "*Why*")

Once the performance picture is established, the next step is to provide insight as to where improvements need to be made. This is done by inspecting the service time data weighted according to the expected usage assumptions. Graphically this would be as follows:



Spring Usage



Fall Usage

Knowing that the average weighted value for CPU is greater than that for disk, it is natural to concentrate on the CPU-intensive resources. Besides the obvious approach of optimizing any operations that exceed the base response time specification, the above graphs highlight where improvements can be made to the overall performance. Since the Spring usage was the most restrictive (at 42 nodes), operation "Op C" would be a likely target for improving the overall performance situation. Further inspection shows that improving the "Op B" operation (in particular the "Op B-small") would have a significant effect in both the spring and the fall. In general, decreasing the service time of any operation would improve the response time of all operations (since the contention for the shared CPU is the primary cause of response time degradation). This is particularly applicable for applications concerned with maximizing node to server ratios or average response times of all transactions.

3.5.4 Reiteration

Continuing with the above example, assume that based on the analysis, the database was optimized for operation "Op B" (both large and small). This was not done originally as it was suspected to have an adverse effect on operation "Op A". After measuring the modified configuration, the new service time data is as follows:

	CPU-seconds	Disk-seconds	Elapsed Time
Op A	1.71	1.69	3.9
Op B-small	1.39	1.01	4.2
Op B-large	1.70	1.54	4.9
Op C	1.82	1.02	3.6

This new data reflects an improvement of about 10% for both "Op B-small" and "Op B-large", however it was

at the cost of a service time degradation to "Op A". Weighting the service times according to the usage patterns yields the following:

	Spring					*Fall*				
	Percent	**Wtd. CPU**	**CPU %**	**Wtd Disk**	**Disk %**	**Percent**	**Wtd. CPU**	**CPU %**	**Wtd Disk**	**Disk %**
Op A	15.0%	0.2565	15.4%	0.2535	21.5%	17.0%	0.2907	18.3%	0.2873	22.7%
Op B-small	28.0%	0.3892	23.3%	0.2828	24.0%	42.0%	0.5838	36.7%	0.4242	33.5%
Op B-large	12.0%	0.2040	12.2%	0.1848	15.7%	26.0%	0.4420	27.8%	0.4004	31.7%
Op C	45.0%	0.8190	49.1%	0.4590	38.9%	15.0%	0.2730	17.2%	0.1530	12.1%
Ave Wtd Usage Per Op:		**1.6687**		**1.1801**			**1.5895**		**1.2649**	

This shows an overall improvement in average weighted service time for the CPU in both cases. Using this data with the 50/40/30 maximum node estimate yields the following:

Target	***Spring***	***Fall***
Max nodes before CPU reaches 50%	43.1	45.3
Max nodes before Disk reaches 40%	48.8	45.5

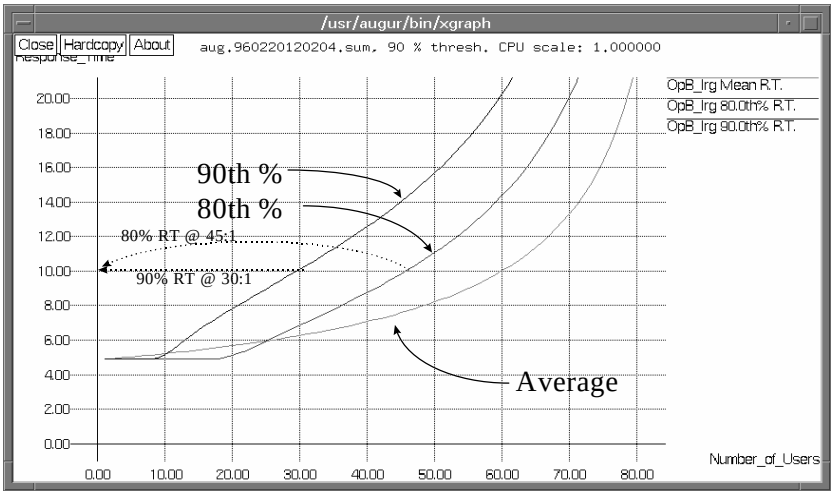
Thus the net effect of this modification is an increase of at least two nodes in the node to server ratio – which would equate to as much as a 5% decrease in the total server investment – a sizable improvement.

Subsequent iterations making similar comparisons will occur any time alternate design architectures are being evaluated. If the performance impact of architectural decisions is not obvious, then mock-ups are created, measured, and used to determine which alternative is optimal.

This step also highlights the three most important aspects of this method:

- By always knowing the expected performance, it is now possible to detect and quantify the effects of any changes made to the system.
- Service time calculations are very powerful, but require very accurate measurements since the slightest change can have a large effect on extrapolations.
- At all times, a rough indication of the capacity is understood – thus avoiding surprises at the end of development.

Remodeling with the new data allows the effects of this change to be visualized:



This model shows the true effects of the improvements – namely the response time criteria for both normal and failover conditions are met (90% of the responses at 30 users are within 10 seconds, and 80% of the responses with 45 users are within 10 seconds).

3.6 Tracking

Clearly modeling has a large advantage over the 50/40/30 rule of thumb, but regardless of the means, it's the methodology that is important. By establishing the "how" and documenting the "why", the only remaining step is to repeat the process and improve on the information. Using the example herein and in Appendix A, and adding to that the conditions of the test (Software revision, hardware configuration, date, etc.), a "Performance Report" is created which allows performance to be tracked. As this process is repeated, the testing should improve on the data collected by:

1. increasing the number and "coverage" of the tests,
2. including more detailed information (LAN, memory, etc.),
3. testing alternate configurations, and
4. breaking out information at increased levels of detail including by process.

3.7 The PE Lifecycle

The process described above is designed to follow the development lifecycle from high-level design all the way through to implementation. Although the above example relies on detailed service time measurements, the same methodology can be used with estimates, again relying on "best available data" to make a qualified performance assessment.

3.7.1 "Book-ending" to Establish a Performance Bound

In the early stages of design – before there is measurable code – a generic statement based on intuition, historical data, or mock-ups about service time can be used. Such an example might resemble: "*Operations will take at least one second of service time but probably no more than two seconds*". Using this off-the-cuff estimate, a bound can still be established by "book-ending" the performance. By using the 50/40/30 technique the following is calculated (based on the 25 operations per-user per-hour usage data):

Case	*Service Time*	*Max Nodes*
Best	1 Second	72
Worst	2 Seconds	36

Thus, the achievable node to server ratio would be assumed to be between somewhere between 72:1 and 36:1.

This is improved upon by the addition of new data. If, for example, a very simple test was created which measured 1.25 seconds of service time, then this would replace the best case and the "book-ends" are brought closer together as follows:

Case	*Service Time*	*Max Nodes*
Best	1.25 Seconds	58
Worst	2 Seconds	36

This sequence is repeated, eventually replacing estimates with measurements and then replacing general measurements with ones representing a true workflow.

3.7.2 Late in the Lifecycle

As development nears completion, results can be obtained from simulations or a beta-test which can be compared to the predicted results. As an example, if the code was placed at a beta-site, and after a four-hour period with 12 users running, the total CPU time used by the application was 1,100 seconds. If these 12 users handled 600 calls in this period of time, then the average weighted CPU usage would be 1.83 seconds per call. This data can be compared against previous test results (thanks to tracking) to see if the beta-site data coincides with the lab data. Before doing this of course, the observed transaction mix of the beta-site should be used for weighting the lab data to make a direct comparison valid. This exercise will highlight any errors in calculation as well as provide an indication as to how test results compare to a true production system. Further, if the observed transaction mix approximates the expected production mix, then the measured CPU time can be used as an input to modeling or the 50/40/30 technique to establish the "likely-case" scenario of expected performance.

3.8 The Final Step

Once the application nears completion, then any final performance adjustments can be made matching the hardware to the physical (or other) constraints. By using the historical performance data – capacity planning can now be done scientifically. As an example, suppose an office was to have 60 users – the capacity planners could now decide if it would be more practical to use three *downgraded* servers and support a 20:1 ratio with 30:1 during failover verses implementing two *upgraded* servers which could support a 30:1 ratio, but handle a 60:1 ratio under failover conditions. Without the scientific data of performance engineering, this could only be done via trial and error – usually when it's too late.

Hardware offers the opportunity to tune the system without modifying the design – and it is a great advantage to have that ability, but it is very important to remember that performance is the result of design and hardware can only address a symptom of a bad design.

3.9 Methodology Summary

The methodology herein is a means to combat the un-obviousness and complexity issues of client-server performance. This is accomplished methodically by implementing a system which:

1. Describes the "***how***" of performance either with modeling or a simplified rule of thumb;
2. Documents the "***why***" of performance so problems can be addressed;
3. Tracks this data so that changes are detected, documented, and understood; and
4. Continuously improves by increasing the inherent accuracy and coverage.

This is achieved by using the "best available data" to make a qualified judgment about the performance of a system. This information is understood to not be precise, but accurate enough to be used – knowing that it will be improved upon as time goes by.[3]

4. Summary

Performance is just another aspect of quality, and like usability or reliability, it is the effect of design – not something that can be added later. By establishing a real performance objective and implementing a methodology to support that objective, performance engineering will enable development to avoid the significant cost of bad performance. Breaking catch-22 of client-server performance ("you cannot know until it's done, but you cannot change anything once it's done") requires a methodology that can "shift the timeline". This means a means an approach that allows

[3] As Aristotle surmised: "[It is the mark of an instructed mind to not demand more accuracy than the subject emits]".

development to see the effects of decisions while there is still time to act, yet is still basic enough to be accomplished.

Implementing performance engineering is a doable process but relies on these key components:

- there must be a ***real goal***,
- there must be ***feedback***,
- there must be willingness to ***accept performance indications*** as opposed to waiting until the data is firm, but the alternatives are few.

Using this performance engineering process, it is possible to design performance into software. Thus what was once accomplished by intuition, but then abandoned for the "fix it later" approach due to the complexity of client-server can be accomplished with a little planning.

References

[Infoworld, 95] *Infoworld*, "Stop Network Bottlenecks before They Happen", April 10, 1995. (Statement by Shahla Butler, director of Performance and Measurement Laboratory for the Center of Advanced Technologies at American Management Systems Inc.)

[Smith, 90] Connie U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[Humble, 87] Lance Humble and Carl Cooper, *The world's greatest blackjack book*, Doubleday & Company, Inc.

Appendix A – Summary of Performance Calculations

Input Data

Usage Estimates:

Mix

	Spring	*Fall*
Op A	15.0%	17.0%
Op B-sma	28.0%	42.0%
Op B-larg	12.0%	26.0%
Op C	45.0%	15.0%
	100.0%	100.0%

Load

Ops Per hour:	750	*Ops/hour*
Ops Per Node:	25	*Ops/node-hour*
Failover Mult.	150%	*of normal*

Measured Data:

	CPU-seconds	Disk-seconds	Elapsed Time
Op A	1.61	1.69	3.9
Op B-small	1.54	1.01	4.2
Op B-large	1.89	1.54	4.9
Op C	1.82	1.02	3.6

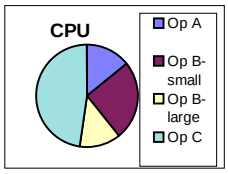
Calculations

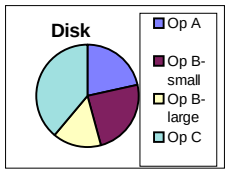
Spring

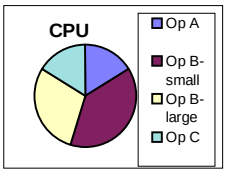
	Percent	Wtd. CPU	CPU %	Wtd Disk	Disk %
Op A	15.0%	0.2415	14.1%	0.2535	21.5%
Op B-small	28.0%	0.4312	25.1%	0.2828	24.0%
Op B-large	12.0%	0.2268	13.2%	0.1848	15.7%
Op C	45.0%	0.8190	47.7%	0.4590	38.9%
Ave Wtd Usage Per Op:		**1.7185**		**1.1801**	

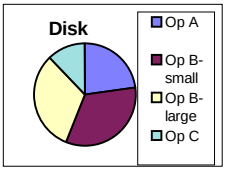
Fall

	Percent	Wtd. CPU	CPU %	Wtd Disk	Disk %
Op A	17.0%	0.2737	16.2%	0.2873	22.7%
Op B-small	42.0%	0.6468	38.4%	0.4242	33.5%
Op B-large	26.0%	0.4914	29.2%	0.4004	31.7%
Op C	15.0%	0.2730	16.2%	0.1530	12.1%
		1.6849		**1.2649**	









Estimated Resource Consumption

*Resource utilization (P) = cost per op * ops per time-interval*

CPU Utilization:

					Normal		*Failover*
1.7185 *secs/op*	750 *ops/hr*	0.000278 *hrs/sec*	=	**35.80%**	***Spring***	**53.70%**
1.6849 *secs/op*	750 *ops/hr*	0.000278 *hrs/sec*	=	**35.10%**	***Fall***	**52.65%**

Disk Utilization:

1.1801 *secs/op*	750 *ops/hr*	0.000278 *hrs/sec*	=	**24.59%**	***Spring***	**36.88%**
1.2649 *secs/op*	750 *ops/hr*	0.000278 *hrs/sec*	=	**26.35%**	***Fall***	**39.53%**

Estimated Max nodes before exceeding utilization (*P*) = 50/40/30)

*nodes = P/((Ops per user-interval) * (cost per op))*

Max nodes before CPU exceeds 50%

		Spring			***Fall***		
		25 *ops/node-hr*			25 *ops/node-hr*		
	*	1.7185 *secs/op*			1.6849 *secs/op*		
	*	0.000278 *hrs/sec*			0.000278 *hrs/sec*		
50.00%	/	0.011934	=	**41.9** *nodes*	0.011701	=	**42.7** *nodes*

Max nodes before Disk exceeds 40%

		25 *ops/node-hr*			25 *ops/node-hr*		
	*	1.1801 *secs/op*			1.2649 *secs/op*		
	*	0.000278 *hrs/sec*			0.000278 *hrs/sec*		
40.00%	/	0.008195	=	**48.8** *nodes*	0.008784	=	**45.5** *nodes*

		Primary Bottleneck
Max nodes, spring:	**42**	***CPU***
Max nodes, fall:	**43**	***CPU***