

Performance Metric Validation in Chaos

Charles L. (Chuck) Burmaster
The Mitre Corporation

R. Daniel (Dan) Johnson
ThomsonConsulting

Robert H. (Bob) Johnson
Landmark Systems Corporation

Abstract

Whether you run a database-backed web site on a farm of distributed NT or UNIX boxes, a mainframe-based data center with more emerging online applications than you thought possible, or something in between, if you didn't know about Power Law distributions before, you had better learn about them now. They're out there, they're everywhere you look, and they're coming to get you!

*If you were a careful observer at CMG97 last year, you may have noticed several presentations with ideas that, developed further, had the potential to knock the socks off conventional performance measurement wisdom. One was on cache: *New Perspectives in DASD Subsystem Cache Performance* by David L. Peterson of StorageTek Corporation. Another was a Sunday session, a sleeper presented by Lipsky and loaded with math, demonstrating that the standard deviation is useless when computed over a Power Law distribution and that a good many of our measurements have such a distribution.*

In other words, everything we believe may be wrong. Okay, maybe not everything, but in light of these recent observations about data that are very close to home, we should question our methods and decide whether our arsenals could stand some improvements: new tools, mathematical and otherwise. To get you started, we enthusiastically present this paper that:

- *Explains why the statistics that we use most often, the mean and standard deviation, are misleading and inappropriate for most of our performance analysis purposes*
- *Provides strategies for coping with a new world that nearly transforms itself on a monthly basis*
- *Presents alternative mathematical and procedural techniques, such as statistical classification and nonparametric statistics, to improve the accuracy of our conclusions and, therefore, the benefit to our companies*
- *Suggests that the World Wide Web, instead of being a complication to performance analysis, can simplify it drastically.*

1. Business and Background

It may come as a surprise to some of us, but our businesses don't really care all that much about our technology. Businesses and people alike want data in a useful medium and they want responsive systems; they don't want to have to "tune" their system to make it run. If you don't believe that, look at your average web surfer. Kids, housewives, and grandparents are on the Web. Some of them probably know more about the Web than those reading this paper! If you have not already been asked, you probably soon will be asked to manage or tune a web server.

Compounding all this is accelerating change. A decade ago, performance analysts and capacity planners could keep up with developments by reading publications, attending a few conferences, measuring their systems, and making decisions. Today, the rate of technology's growth and adaptation is accelerating, sometimes overtaking applications and rendering them obsolete before their development is complete. Time-to-market is shrinking, the general public is being invited and lured to become users of our systems, and commodity hardware amplifies the platforms that we must manage. Time is now measured in web-years, or about a month to us mere mortals.

We are asked to evaluate and manage these constantly changing systems, yet we can no longer afford adequate time to study and understand them. Worse, the statistics that we use and the rules of thumb that we apply may no longer be valid under today's new conditions, yet businesses are making decisions based on these measurements without understanding their probability of error. You may have heard of decisions to add software or hardware that were financial disasters. This paper begins with the assumption that new systems are going to be monitored and business decisions are going to be based on that monitoring before a complete understanding of the system or the metrics can grow with time and long-term study.

Mathematics to the rescue! The second part of the paper describes the application of Statistical Pattern Recognition (SPR) to computer metrics. The application of classification algorithms has proven helpful in many other scientific disciplines and can also be used to improve our understanding of computer metrics in real-world environments.

Analysis of a batch of data when the system is "good" and when it is "bad" can lead you to understand metrics and the probability that they are a leading indicator of "good" or "bad." The second part shows these techniques used in a benchmarking setting to evaluate different configurations. This part shows these techniques used in evaluation of a standard performance problem, such as: Do we need more central storage? The third part gives you a technique for measuring performance in this new world and a look at how the Web may be on your side.

If you can measure it, you can manage it is an old adage in this business. Measuring implies that you have a tool with which to measure, that is, to supply data about the system. Managing implies you make sound decisions based on those measurements. Therein lies the problem.

Many CMG presentations have concentrated on the measurement and management aspects of a relatively stable computer environment. Over the years, new technology was brought online and conferences like CMG were used to publish and discuss how to measure the technology. Groups like CMG, SHARE, and Guide are vital to the vendors as far as meeting with real customers who are trying to use and manage the technology. We wound up with metrics that were understood and experiences that gave us a good clue as to how to manage the system based on those measurements. This paradigm worked until the 1990s.

Each year brings better and faster hardware technology, better and more complicated operating system software, better and more complicated application software. Having learned that people need metrics to manage their systems, hardware and software vendors began to describe and externalize metrics. IBM has always provided more data than we really need. Original equipment manufacturer (OEM) vendors provide extended metrics for their subsystems (for example, StorageTek's Iceberg). Microsoft provides hundreds of metrics for its desktop systems (Windows NT and 95) and an open implementation that lets additional Microsoft and non-Microsoft components contribute their own metrics.

Today, architecture is changing so fast that there is not enough time to study the architecture, hypothesize what metrics will be used to manage it, and compare our predictions to practical installations. For example, with Windows NT, there are two completely different books and CD-ROMs to describe the metrics: the server and workstation resource guides, as well as hundreds of books by publisher-proclaimed experts. We are dealing not only with different processor platforms and architectures within the same system but also with virtual machine interpreters and just-in-time compilers as well as conventionally compiled programs.

1.1. Mainframe Metrics

To get us started, let's look back at history to see what we used to do. (What is past is prologue.)

Most OS-390-MVS and VM/390 metrics were gathered by the operating system and written (or "punched," in the case of VM) to a common data set. The software allowed us to specify the intervals and the types of data to be collected, and it was up to us to manage the data. Rarely did the collector portions give us any display of the information. As we wrote the applications, we looked at the raw data to determine how it could best be displayed.

Generally, we used averages for the numbers. This was an easy representation to get with spreadsheets or general-purpose programs like SAS. The next thing we did was to plot these averages and determine if there was a pattern of "good performance" and "bad performance." The averages were used as an indicator. *Rules of thumb* were developed to help us predict problem status:

- Over 30 ms per I/O operation was "bad."
- CPU averages over 80% busy was "bad."

- TSO¹ response time over 1.5 seconds was “bad.”

1.1.1. The Trouble with Tribbles (Averages)

But most times, when performance analysts reported averages, the end-user environment dragged one of us over to the terminal and showed us much worse response times. “Your numbers are wrong,” they would say. Thus, we developed additional criteria to get a better indicator. Take the last one above, “TSO response time.” That number (or its CICS² equivalent) is not really what the end user sees, but the value measured inside the system. In the case of TSO, it is the measured time a terminal I/O is received and the TSO address space is dispatched until the TSO address space sends screen updates back out. There is no disabled wait time included. There is no network time included. The end user sees a very different response time. This led to a technique many of us have used for years:

Add one standard deviation to the average to estimate the “end-user’s viewpoint.”

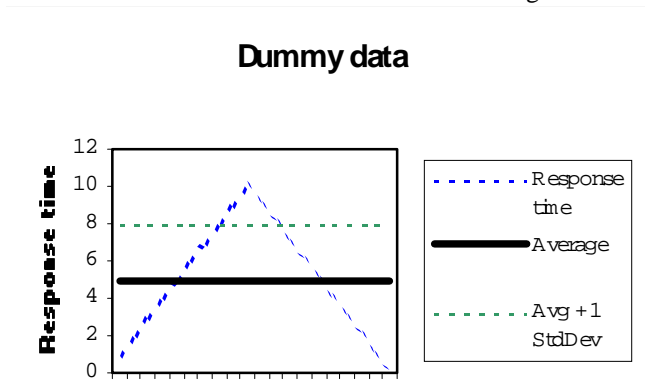


Figure 1. Dummy Data to Show the Problem: Average is 5 Seconds, Plus One Standard Deviation is 7.9 Seconds.

Figure 1 shows this technique for a set of dummy numbers. Each of 20 time values has a different “response time.” The *average* is 5 seconds. But exactly half of the responses are *above* 5 seconds. Adding one standard deviation gives a mathematical number of 7.9 seconds such that only 10% of the “users” are seeing a number over the 7.9 seconds. But of what real value is this number?

1.1.2. The Trouble with Standard Deviations

Adding one standard deviation was an attempt to get to the hidden response time. It made sense, mathematically, because adding one standard deviation above the average gives you a smaller error or better estimate for end-user time.

What is a standard deviation? It is used to calculate the spread of an observation from the average. A calculation of standard deviation is fairly simple:

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

1. TSO is the IBM time sharing option used by mainframe developers and data center personnel. It provides a robust environment to edit files, measure what is going on in the environment, monitor batch work, and develop programs and procedures.
2. ICS is IBM’s Customer Information Control System, a major online terminal program and application environment. CICS is used in almost all MVS/OS-390 environments and can operate on many other platforms.

There is no problem with the standard deviation by itself, but there is a problem with the way we use it. It is okay to state that 66% of all the response times fall within one standard deviation of the mean (that is, the mean $\pm 1s$), and it is okay to state that 95% fall within two standard deviations and 99% within three – provided that the response times are distributed “normally.” These Gaussian or normal distributions have been studied to death, but in general terms, a set of values is said to be normally distributed if, when the values are plotted as a frequency histogram, the shape of the histogram is the familiar bell curve. The important characteristics of this curve are that it has a hump in the middle, which narrows on either side into thin tails, and that it is symmetrical. Our response times are *not* distributed normally, nor are they usually distributed symmetrically around the mean. If you say that 66% of your response times would fall under the mean plus one standard deviation, then you’re relying on circumstances that don’t generally exist. If that percentage happens to be right, it’s more due to luck than scientific reasoning.

Another reason a standard deviation is misleading is that it sums squares. Very high numbers (*outliers*) included in the population that is squared cause huge swings in the standard deviations. Calculate 1,000 values of 5 and a single 10,000 metric and you will see that the standard deviation is really high (316)! 1,000 happy customers and you just reported 331 seconds response time! Boo!

The “TSO response time” was just the time that the operating system saw from when it passed control to TSO to when TSO was in terminal wait. But each transaction could be completely different from the last one. Morino Associates (later Legent, later Computer Associates) had a product, TSOMON, which allowed the performance manager to specify a number of parameters to segment response times. For example, a TSO transaction that read a very large file had much different expectations than one that was just going down one page of a file already in storage.

Several papers address the problem of using a standard deviation. See [Lipsky97]. This paper describes a study from network response data. The authors showed that the standard deviation was infinitely variable.

Just imagine the overwhelming joyful response you would get from management if you showed them a TSO or CICS response number of 3.9 seconds and demanded they buy more real storage. After the money was spent and installed, you showed them a 4.9 second response. Joy would hardly describe that meeting!

The third problem with the standard deviation is that we are discovering that the measurements we take of current systems consist primarily of noise. As you graph it, it looks noisy. If you decrease the time granularity and graph it again, it still looks just as noisy. This is called *pink noise*. Think about this. If you take a set of data and graph it for a time period with each section being 100 milliseconds (ms), you will get one shape of a curve. If you expand the time period for each of the bars, you will get another shape. As you expand to 500 ms and then to 1 second, you would expect that the noise would be muffled by all of the measurements in a single bar. The value will flattened out to become the mean. That works for places like a large mainframe data center where workloads are planned and scheduled. It does not work (on networks) where the outside world (indeed, with the Web, the rest of the world) determines the arrival rate and load.

1.1.3. A Plan

We must find other ways to deal with data. One of the most-often asked questions is: “What metric do I use, what should it be, and what do I do when it gets outside ‘normal’ boundaries?”

You must have a function that measures the consequences of indirect activity. It is a binary “good” or “bad” for your system. In most cases, this will be response time. We recognize that this metric is not generally available in these new environments. We emphasize that these techniques depend on your ability to have one metric in each sample interval (row) indicate good or bad.

We need nonparametric ways to describe the data. Many of the statistical patterns observed in non-mainframe environments are not Gaussian or Poisson. We can no longer depend upon the traditional statistical assumptions when we begin to classify and describe the data. Traditional statistical approaches are having difficulty with “fat tailed” distributions [Lipsky97]. This brave new world is new to statistics and new to us.

There are statistical tools that we can use to reduce our error somewhat. One is to substitute for mean and standard deviation the geometric mean and standard deviation. This is just like the regular flavor except you take the logarithm of each value before computing the mean and standard deviation, then you take the antilog of the mean and standard deviation once these are computed.

Perhaps a better way would be to use plain old percentiles. These are nonparametric statistics: They do not depend on the distribution of our data and they describe the distribution in clear ways that are easily explained to just about everyone. A percentile is a statement about an entire set of numeric values. Given a set of values, the n^{th} percentile is the value for which n percent of the values in the set are above the stated value and m percent of the values are below that value. The sum of n plus m equals 100%. For example, in the list: 1, 2, 3, 4, 5, 6, 7, 8, 9, the 50th percentile is 5 because there are an equal number of values (4) above and below 5. The 25th percentile is 3 because a quarter of the values are below 3 (conversely, 75% of the values are above it). Likewise, the 75th percentile is 7 because 75% of the values are below 7 (conversely, 25% are above it).

Percentiles describe the distribution of the values with just a few data points and are valid regardless of the data's distribution. Useful percentiles for this purpose are 0% (or the minimum), 5%, 25% (or the first quartile), 50% (or the median), 75% (or the third quartile), 95%, and 100% (or the maximum). We may only be interested in the upper half of our distribution, so the median, 75%, 95%, and maximum may suffice.

Why do we get into trouble? Because we use tools that worked in the past that are not responsive to our needs. Same old tools give same old results.

One example is our experience with trying to quantify real storage constraint on UNIX response time. Carrying our mainframe experiences forward, we looked for a parameter called "demand page-in" because that is what works on the mainframe. Demand page-ins cause wait states for applications and elongate response times. Not true with all variants of UNIX. In Sun Solaris (and others), demand page-in counts are used for file I/O.

It is less important to crank through formulas than it is to find simpler measures and apply thought and experience to these measures. Rules of thumb fail. Plug and play does not work.

Too many factors affect our systems. Today, you must know the architecture (for example, Solaris) to determine which metrics to monitor and what the numbers should be. With Windows NT, it is just the same.

Another situation occurs with building systems. There is no way to get "adequate" response times before the system is implemented. You must be there when it is implemented, positioned to help them discover -- and describe -- response times.

So what are we left with as a plan

- 1 Understand/identify metrics:
 - For specific architectures, consult the expert (like Adrian Cockcroft or Brian Wong at Sun). Let them tell you what metrics to watch and why.
 - Use another mathematical system to identify good and bad metrics. (For how, see the next part.)
- 2 Describe the system and see if you need to buy or fix something to meet your performance criteria.

2. SPR: Motivation and A Little Mathematics

The measurement of computer metrics generally occurs to determine (hopefully) whether or not a particular computer system is working well or if the system is on the brink of failure. Often the problem is associated with monitoring legal commitments to service levels in terms of measurable "trusted metrics" such as "response time" or "thruput." When such measurements are obtained for a local system, the service levels agreed to can be observed directly. However, when such measurements are desired for services provided at a distant location, say, in another state, and measurements are only available locally, how does one relate local metrics to distant performance? Such decision-making can be helped by taking advantage of ideas originating in the field of Statistical Pattern Recognition (SPR). In particular, we will consider (1) how to identify useful local metrics on any given system and (2) how we might use selected local metrics to flag a system (including a remote system) that is on the brink of failure. Identifying Useful Local Metrics

2.1. Identifying Useful Local Metrics

The first step we propose is to examine measurements of individual computer metrics using a data picture obtained via log-histogram plots. A *log-histogram plot* is simply one showing log to the base 10 of the histogram count plus one for each bin. The vertical scale becomes a base 10 order of magnitude value. The log-histogram can be used to retain a meaningful histogram plot when a small number of bins contain most of the data and outliers are relatively few.

Instead of forming a log-histogram of all the data for a given metric, we first separate the measurements of the metric into two groupings. The first set of measurements is obtained when the system is operating poorly and is labeled “bad.” The second set of measurements is obtained when the system is operating satisfactorily and is labeled “good”. In Figure 2, the overall system response time is used to define “good” and “bad.” In particular, when the overall system response time was less than or equal to 5 seconds, we classified the associated metric values obtained as “good” and, when the overall system response time was greater than 5 seconds, we classified the associated metric values as “bad.” While this is arbitrary, it was deemed reasonable for this application and representative of alternative choices that could have been made.



Figure 2. Split Log-Histograms of User CPU Time (msec) for Good and Bad Response Times.

Next we wish to illustrate the results of using an available local metric (User CPU Time) to characterize or classify system performance in lieu of the trusted system metric (response time). To do so, we define a User CPU Time threshold and say the system response time is “good” when the User CPU Time is below this threshold and “bad” when above this threshold.

The problem in attempting to flag a good or bad system with any such metric is the overlap in the “good” and “bad” histograms. Any threshold for a particular metric selected to delineate between “good” and “bad” system performance will be correct part of the time and fail part of the time. In general, there are four possibilities we encounter in utilizing such a classification scheme. We can:

- Correctly call a system “good” when it really is good.
- Falsely call a system “good” when it really is bad.

- Correctly call a system “bad” when it really is bad.
- Falsely call a system “bad” when it really is good.

Note: It may be useful for understanding to note that the sum of the probabilities of conditions (1) and (4) must total 1 and the sum of the probabilities of conditions (2) and (3) must also total 1. The sum of probabilities associated with conditions (1) and (2) or with conditions (3) and (4) do not generally add to one. Hence, we effectively know all the probabilistic consequences of selecting a given threshold if we know the probabilities associated with condition pair (1) and (2) or condition pair (3) and (4). In Figure 3, we illustrate the effect of a moving threshold on the probability of correctly calling a system “good” when it really is good and on the probability of falsely calling a system “good” when it really is bad.

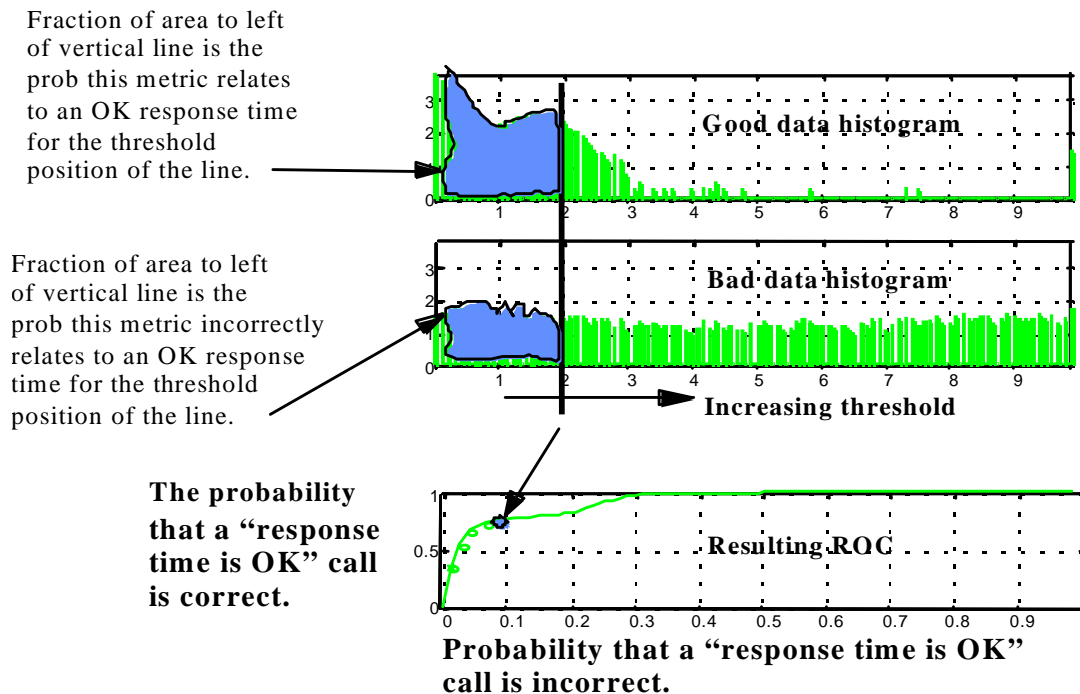


Figure 3. Creating a Receiver Operating Curve (ROC) or System Classification Curve with a Single Metric.

2.2. Evaluating Individual Computer Metrics

ROC plots from split histograms are intended to illustrate the use of a single metric to classify a system as “good” or “bad”. We see in Figure 2 that low values of User CPU Time are generally (but not always) associated with “good” system response times while larger values may (but not always) indicate “bad” system response times.

In classic detection theory, a plot of the probabilities in a ROC plot is often used to describe the range of detection performance of a given receiving system as a detection threshold is changed. We will still use the term ROC even though we are not considering “Receiver Operating Curves” but rather are examining the classification performance of individual and (later) combinations of computer metrics.

If we computed a ROC using our trusted metric of system response time, we would obtain a plot of what the “best” possible ROC would look like. This is shown in Figure 4.

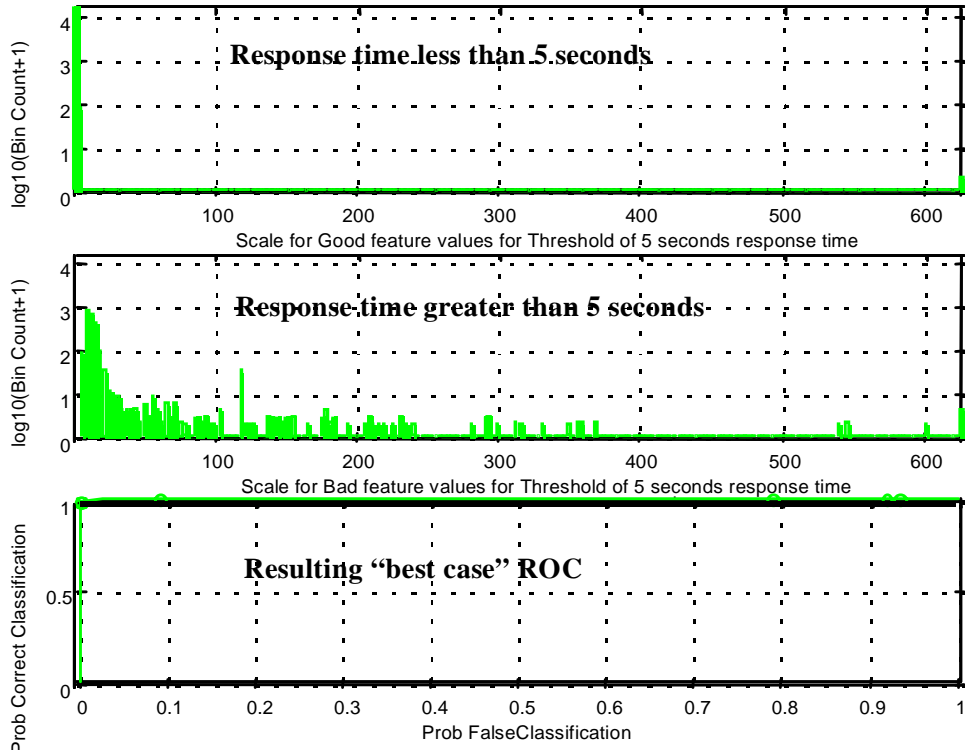


Figure 4. Split Log-Histogram of System Response Time: the “Best Case” ROC Plot.

In general, we would like to find a metric that has a ROC plot similar to the ROC of our trusted metric; that is, one that plots in the vicinity of the upper left corner of such a probability-performance space so that we can choose a threshold that allows correct classification *almost* all the time with false classification *almost* never occurring. The closer to the upper left a ROC plots, the better the single-metric threshold can represent overall system performance.

What we usually have, however, is not just one metric but many metrics. In the next plot, six different metrics have been chosen to illustrate how ROC plots can be used to identify good vs. bad metrics with regard to their representation of system response time (we could have used some other trusted metric). The metrics used for illustration are:

- System CPU Time (msec)
- User CPU Time (msec)
- I/O Kbytes/Sec
- Total I/O Kbytes
- Write Ops
- Real User ID

ROC plots for these six metrics and an indication of whether or not the individual metrics are good or bad in their ability to classify overall system response are shown in Figure 5. Metrics with ROC plots in the upper left corner of the probability-performance space are individually useful in characterizing overall system response time while those that move on or below the diagonal are not desirable for classification applications. Individual correlations with system response time are included for comparison.

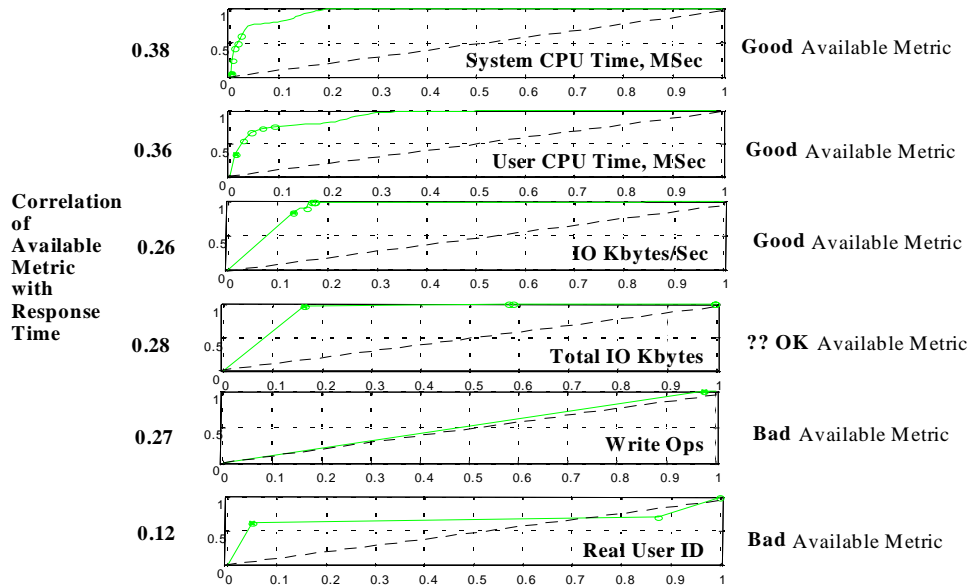


Figure 5. ROC Plots of “Good” vs “Bad” Metrics for the Six UNIX Metrics Relative to System Response Time.

The correlation with response time (or some other trusted metric) is useful in indicating metric utility. However, only a ROC plot describes the degree of utility in a probabilistic sense.

2.3. Combining Individual Metrics Using a Discriminant Function

So far, we have used, in a probabilistic sense, the degree of utility of the relationship between a single local metric and a trusted metric to determine whether or not the local metric might be useful in representing the trusted metric. However, it is difficult to obtain *individual* metrics that perform well in this capacity. As an alternative, it is possible to combine the metrics in a mathematical sense, using the Theory of Statistical Pattern Recognition [Fukun90]. To illustrate one such application, we will compute a linear transformation (or *discriminant function*) of the six data vectors into a new single combined metric.

If our metrics were Gaussian variables, the transformation would be a linear optimal transform with the least Bayes error. However, our metrics are not Gaussian and this application is, therefore, suboptimal. We will see how well the transform works for this illustration.

For the purposes of transforming the data to plot in a histogram pair, we apply the transform to the good and bad data separately. The following uses matrix algebra with superscript “T” being the transpose operator:

$$TX1 = V^T X1 + v0$$

$$TX2 = V^T X2 + v0 \text{ Eqn (1)}$$

with

$$V = [P_1 \Sigma_1 + P_2 \Sigma_2]^{-1} (M_2 - M_1)$$

$$v0 = -V^T [P_1 M_1 + P_2 M_2]$$

where:

X = vector of data [in our six-metric example, "X" is a 6 x 1 vector]

P1 = fraction of sample measurements called "bad"

P2 = fraction of sample measurements called "good"

M1 = mean of "bad" vector metrics [in our six-metric example, "M1" is a 6 x 1 vector]

M2 = mean of "good" vector metrics [in our six-metric example, "M2" is a 6 x 1 vector]

Σ_1 = covariance(X1)

Σ_2 = covariance(X2)

X1 = vector of "bad" data [in our six-metric example, "X1" is a 6 x 1 vector]

X2 = vector of "good" data [in our six-metric example, "X2" is a 6 x 1 vector]

Should we ultimately implement this discriminant function, we will simply calculate:

$$\begin{array}{ccc}
 & \text{decision "bad"} & \\
 H(X) = V^T X + v0 & > & \text{Threshold} \\
 & \leq & \\
 & \text{decision "good"} &
 \end{array}$$

using a threshold selected to achieve a certain operating point

Using the transform shown in equation (1), we use all six metrics to (hopefully) improve upon the performance predictions available with any single metric.

As seen in Figure 6, the numeric values resulting from this transformation range roughly from -6 to +82. The decision threshold can now be selected to obtain the performance desired. However, it is not yet readily apparent that the overall ROC has been improved upon over what was available previously using the ROC plot of the best individual metric shown in Figure 4.

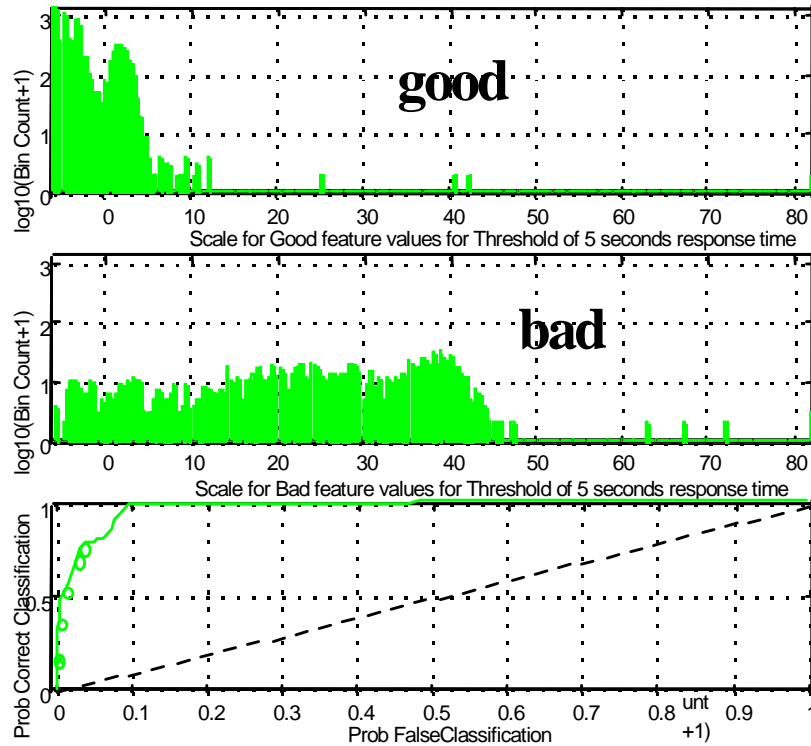


Figure 6. Six Metrics Transformed into One.

If we divide our data sets in half and train on one half and test on the second half, we obtain some indication of how robust our transform performs. See Figure 7.

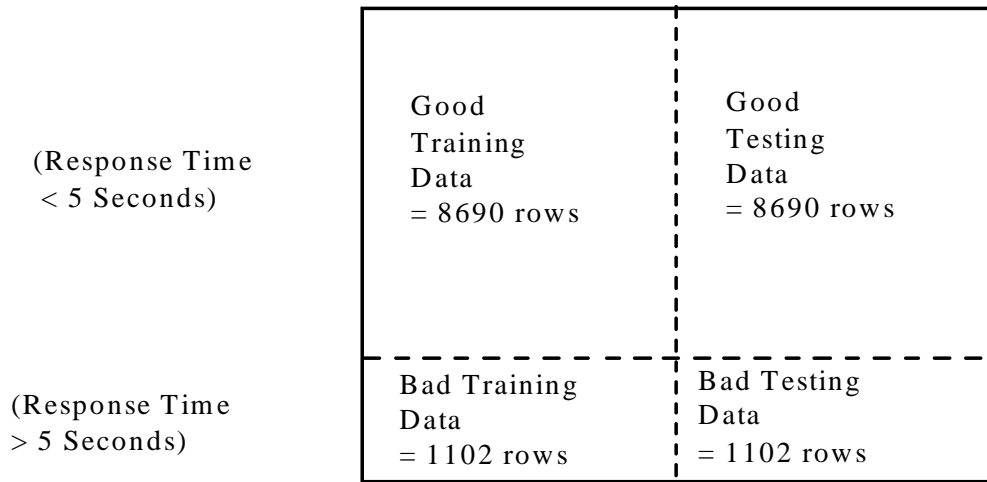


Figure 7. Partitioned Data for Training and Testing.

Doing this, we found 8,690 rows for each of the two samples where response time was less than 5 seconds and 1,102 rows where response time was worse than 5 seconds.

In Figure 8, we compare the ROC plots obtained with the training data and with the testing data. We see a slight drop (shift) in the performance plot. How close the testing ROC comes to the training ROC is an indication of how robust the results might be.

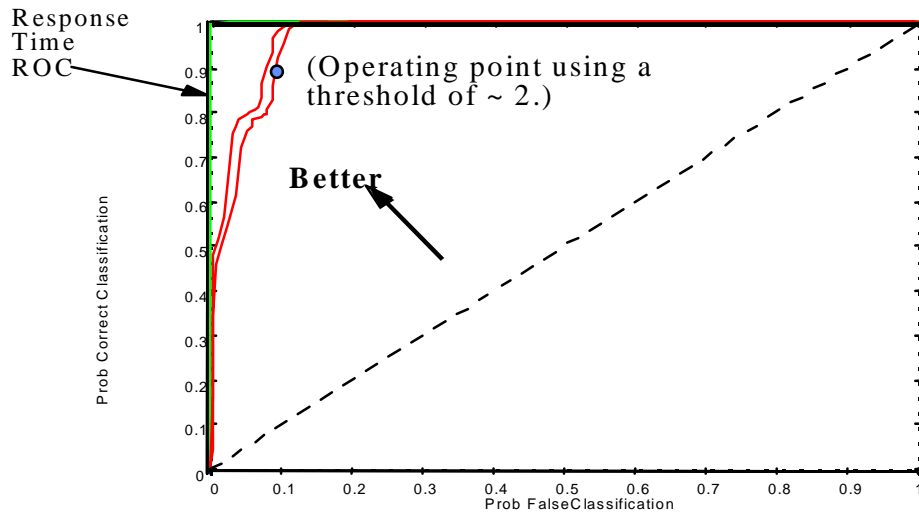


Figure 8. ROC Plots of Training and Testing Data.

When we methodically train and test in a random fashion and then plot the resulting operating points, we see more clearly the robustness of our classification performance. Figure 9 illustrates the method used to repeatedly partition the data for such an estimate.

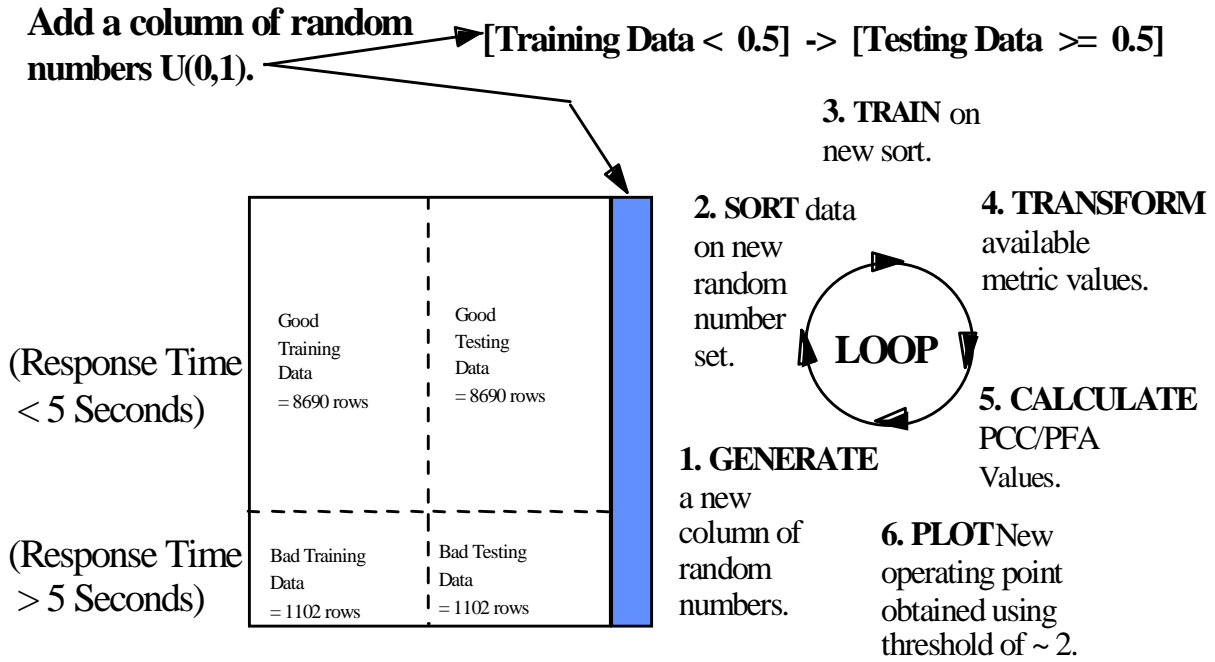


Figure 9. Estimating Uncertainty in the Operating Point.

Including “bad” metrics (features) in the transform will cause the overall performance to be degraded. Out of the 100 Monte Carlo operating points shown in Figure 10, several occur at unacceptable probabilities of misclassification.

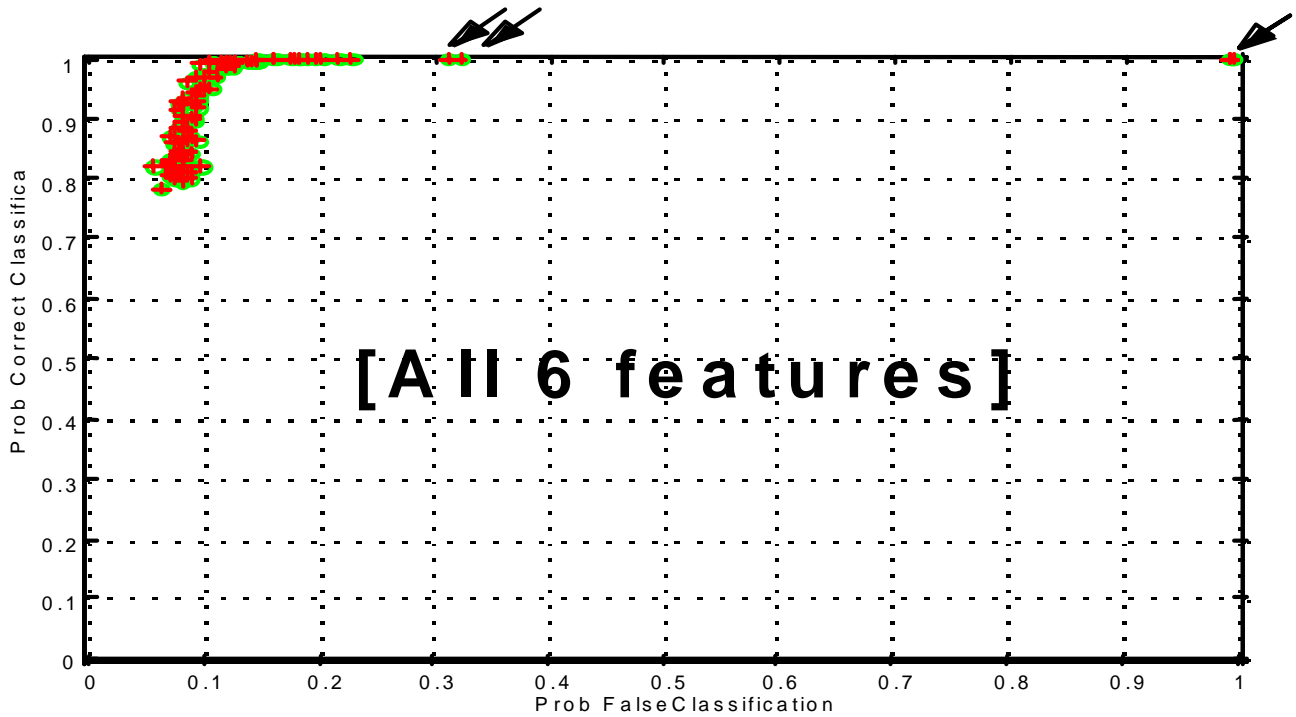


Figure 10. Too Many Metrics can Degrade Results.

When the “bad” metrics are removed from the transform, the stability of the operating point is greatly improved and the performance outliers are gone, as shown in Figure 11.

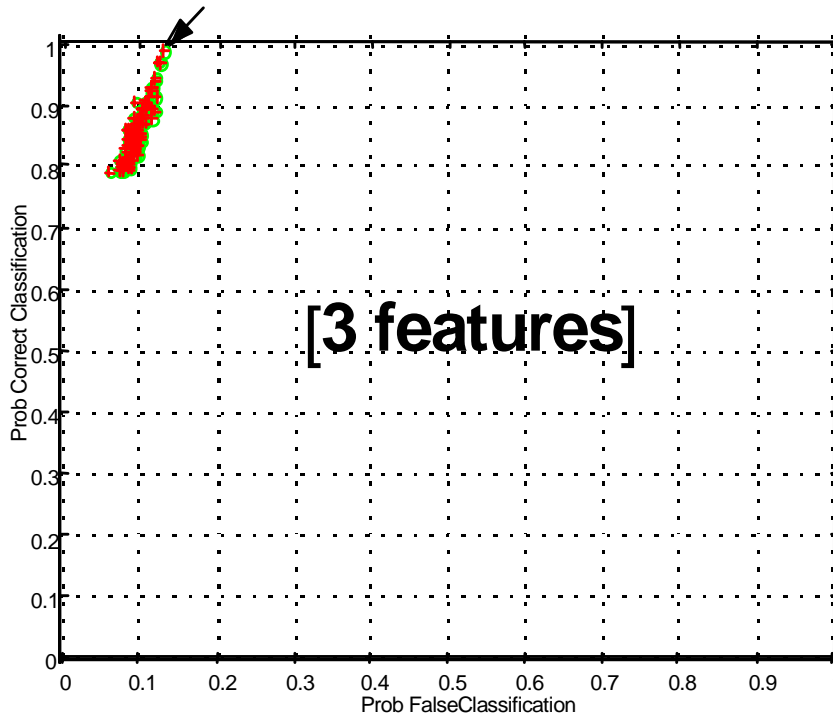


Figure 11. Better Results are Seen With the Three Good Features.

2.4. Statistical Pattern Recognition in Summary

We can use methods of Statistical Pattern Recognition to accomplish three goals useful in computer performance monitoring.

- **Train.** Evaluate the impact/usefulness of available computer metrics on service levels by relating them to a “trusted metric.”
- **Evaluate.** Combine available metrics to obtain an indication (or estimate) of service level achievement.
- **Troubleshoot.** Use ROC plots and operating points of available metrics obtained in (1) to troubleshoot system components when service levels are not met.

3. Practical Applications

One of the best “trusted metrics”

to use with Statistical Pattern Recognition (SPR) in performance management is response time. It is relevant to end users, and because of this, it is easy to express or reach consensus about value judgements based on response time. Response time is not very easy to measure, though, and when it is possible to measure it, it is rarely possible to measure it all the time.

Once you have used SPR to evaluate metrics readily obtainable, these available metrics serve as a surrogate for continuous response time measures. Those metrics identified as “good” also serve to indicate, indirectly, those resources upon which an application most depends. That in itself is useful in configuration management and capacity planning.

How can we measure response times for the purpose of SPR? Is it realistic to measure just response times to monitor the performance health of a system and just do away with SPR and other analysis techniques?

3.1. Holy Grail: Measured Response Time

The proposed Application Response Measurement protocol, now past its second revision, promises to instrument all of our applications and server components. Unfortunately, it is not – and cannot be – the ultimate answer to performance measurement in the future. ARM will likely fail to pervade online applications, not because it is a bad idea, but for two other reasons. First, there are other less intrusive ways to accomplish similar ends. Second, facilities similar in concept to ARM exist and have not been used either, such as the Trace Normal Form (TNF)¹ facility in Solaris.

Marketing reasons aside, what seems to motivate ARM is the holy grail of actual end-user response time measures together with the well-intentioned desire to split response time into component segments. Wouldn't the world be simple if we had actual response time measures and could tell how much of that time was spent in the net and how much in tier 1 server processing, tier 2, and so on? That's what ARM seeks to provide. It is a noble goal.

The first problem is as we said at the beginning: Development schedules are becoming more aggressive and time-to-market is becoming shorter. Programmers barely have enough time to implement and test function before deployment, leaving little or no time (or negative time!) left over for the instrumentation design and implementation that ARM requires. (Anyone who thinks that instrumentation can be dropped into code as an afterthought underestimates the requirements of the task.) The second problem is that ARM depends on the components also being universally instrumented. The third is dependence on a server to collect ARM data in a defensibly low-overhead manner, a server that you can't purchase without investing in one or another vendor's performance management software suite.

When most programmers are presented with the task of determining the performance of a distributed system, their natural first reaction is to attempt to record timestamps as a transaction progresses from component to component through the system. That may work *if* every component is instrumented and *if* you can collect and gather the timestamps for each transaction and *if* the time-of-day clocks on all your distributed systems are synchronized and *if* all this data collection doesn't affect the system you're testing.

ARM provides an API, a protocol to collect the data, and assurances that the overhead will be low. There are clock synchronization mechanisms such as NTP² for UNIX. But the complexity is as high as is the possibility of introducing an undetected error.

ARM is a fine idea, though it is not clear that it is worth its cost or complexity. What can we do instead if our applications aren't or can't be instrumented, if our commercial off-the-shelf (COTS) software isn't instrumented, if we don't have an expensive performance management solution available to us?

3.2. The Web to the Rescue!

One of the immeasurably valuable things that the World Wide Web has done for us performance analysts is to architecturally separate user interfaces – and users – from implementations and interpose a standard protocol between them. (Client-side Java has the potential to weaken this advantage, but that is a subject for another CMG presentation!)

We have indeed come full circle. We began with dumb terminals connected to central computers. We progressed to decentralizing processing all the way out to autonomous desktop computers. Here we are again, this time with “dumb” web browsers talking to not just one but arbitrary “central computers”. This time, however, instead of ASCII or 3270 data streams, we have HTTP³.

1. NF is an extensible application program interface (API) for reporting events and data about them along with a mechanism for storing event data in a trace buffer and a visualization tool. The Solaris kernel itself is instrumented with trace instrumentation points. Sun does not guarantee their support of TNF.

2. TP is the Network Time Protocol that has been around since the '80s but has gone through several revisions. See RFC-2030 for the latest, Simple Network Time Protocol (SNTP). Time synchronization devices and host computers participate in this protocol to track and correct deviations in system clock time while taking into account such influences as propagation delays. The protocol seems to work quite well, keeping adjacent UNIX system times within a few microseconds of each other.

3. TTP is the Hypertext Transport Protocol, a proposed standard defined in Internet RFC-2068. This is the protocol used over TCP between web browsers and web servers.

The beauty of HTTP is that a user and web browser are not required to generate it. We can simulate users and their web browsers a lot easier than we were able to simulate users at dumb terminals and infinitely easier than we were able to simulate “fat” applications running on desktops and accessing central computing resources.

One of the authors has had good success with a program written in C++ that simulates hoards of web users.¹ HTTP transactions corresponding to end-user actions are sent to the front-end web server, and the simulator measures and records actual response times. This idea is similar to Teleprocessing Network Simulator (TPNS), which the mainframe crowd might remember, but HTTP’s higher level of abstraction makes it less onerous to construct tests than it was for TPNS or other terminal emulators.

To populate tests, HTTP transaction groups can be constructed by hand, but a far more powerful technique has been to sample actual system users and “replay” their actions at amplified load levels. See Figure 12. To accomplish this, conventional tools are used to capture inbound network datagrams at the web server. A program that implements enough of the protocol layers that the datagrams represent (for example, Ethernet, IP, TCP, and HTTP) is sufficient to produce a fresh test configuration. So, take a sample, convert it, replay it, and Shazaam! You have just selected a user load, possibly at future levels, subjected your server(s) to it, and measured end-user response time!

1. We call this program *WebVitesse*[™], *vitesse* being French for *speed*. Cute, no? Actually, the HTTP implementation is a plug-in component implemented as a dynamically loaded, shared library. *Vitesse*[™] by itself is a performance test harness providing timing, control, and data recording functions. The plug-in supplies low-level transaction handlers. We have written CORBA handlers for *Vitesse*, for example, to test the performance of non-web servers.

(Writing your own automated test systems is highly recommended, because you learn a great deal about the underpinnings of the systems that you are testing. If you can't afford the time, there are vendors who will sell you products that do similar things by monitoring and replaying end-user actions. Ours supports, of necessity, features that current commercial packages do not, features that we happen to require for subscription-based content-delivery systems.)

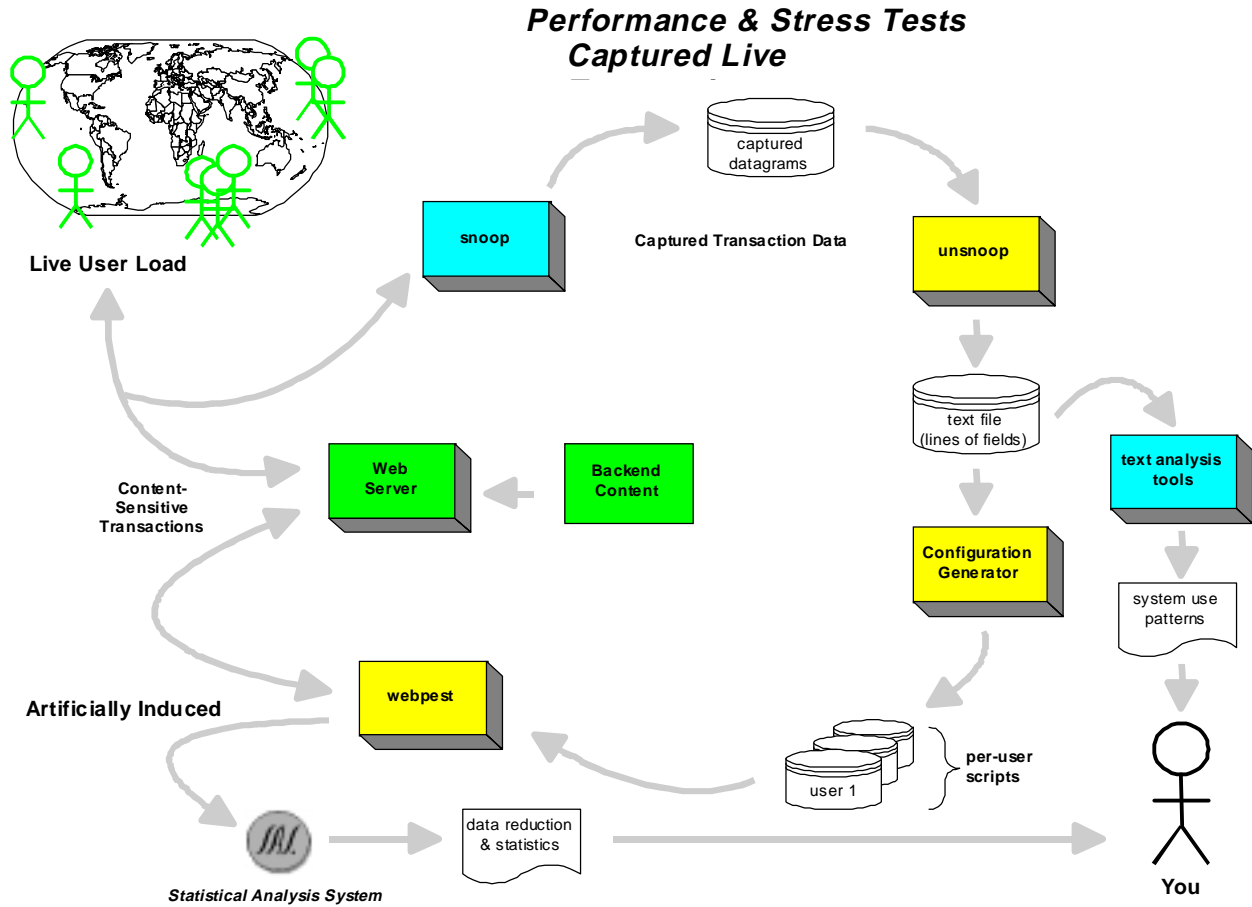


Figure 12: Flow to Capture and Replay Live Transactions.

3.3. "User" – Beware!

But wait – is this a benchmark, or are we solving a perceived performance problem on our production machines? In today's world, not only *can* it be both, but arguments could be made that it *should* be. When servers are deployed on commodity systems, it is probable that staging configurations – test systems – are available for testing the results of the developers' latest race to the finish line. Sometimes there are periods during which you can stress the actual production servers. You are not, after all, making intrusive changes to the production system, just creating a bunch of users "out there".

What is a *user* anyway? What load does a user present to the system? How much load does the entire user base present at any time to the server(s)? It is often difficult to reach a consensus on these questions among everyone in an organization who is concerned with system performance. One of the authors uses a rule of thumb that has proven itself remarkably accurate for UNIX systems:

- 10% of the registered users are online at any time.
- 10% of these are actively interacting with the system.
- Each of these active users initiates one action every 60 seconds on average.

For example, if your application has a total registered user population of 50,000, then 5,000 are “online” with the system and 500 are actually doing anything with the system, with the ultimate result of 500 end-user actions per minute appearing at the server. So you can run a load of 500 actions per second, measure the response times, and declare that the system can sustain the reported response time for a population of 50,000 users. It is the *registered* users that marketing and the other managers understand when they say *users*.

Everyone has some understanding of the concepts behind the rules of this model, and they form a good basis for consensus – or a starting point for reaching one. Most important from a performance reporter’s perspective is that results based on this model can be easily rescaled for an individual’s trust of the model’s accuracy.

Be careful with the Web, though. Each *user* action frequently results in many web (HTTP) transactions. Clicking a button on a web page may result in five or 10 HTTP GET transactions: possible redirection GETs, a frame GET, page GETs for each frame, GETs for each image, and so on. If you are testing a web application and need to generate 500 end-user operations per minute, this may translate to 5,000 HTTP transactions (or “hits” as they are popularly reported) each minute. Design your tests to preserve these units!

3.4. Response Time and SPR

Whether we use a technique that leverages web browsers, ARM, or some other method to obtain response times or some other trusted metric, the trusted metric must be obtained in correlation with customary system metrics. In so doing, we can use SPR to adapt the available metrics to the hardware and software configuration that is being measured. Instead of trying to explain the behavior of individual metrics as a way of describing the system (and employing rules of thumb as shortcuts), we are free to treat the entire collection of metrics as a unit.

What a powerful concept this is! Instead of relying upon rules of thumb that may or may not apply or be valid for every site, the technique produces, in effect, the rules of thumb that work for that site alone, “rules” that are derived from the site itself.

4. Future Directions

This paper is finished, but the work has barely started on applying Statistical Pattern Recognition and other nontraditional techniques to performance analysis. We can foresee opportunities for advancement in several areas:

- SPR tools and frameworks for Matlab, Microsoft Excel, SAS, the HP-48/GX calculator, etc.
- Application of the techniques to many platforms and many systems – a portable methodology!
- Automation including dynamic filters to record metrics in an adaptive manner and alerts for imbedded performance feedback.
- New web server instrumentation and metrics to feed into SPR analysis.

The most important opportunity that we should all seize is collaboration among CMG members. CMG and similar user groups were built from the ground up. That means that members shared information and techniques, tried them on their home systems, and reported the results back to the group. In this way, worthy techniques were preserved and unworthy ones discarded.

5. Summary

Today’s world of distributed systems and online commerce requires changes to the practice of performance analysis. The familiar statistics and rules of thumb become less effective as the world changes under them.

A collection of small systems does not behave like a large mainframe, and neither do powerful but small systems. The workloads have changed and the systems have changed. Conform to them instead of expecting them to conform to you.

It is a fallacy to assume that you can successfully throw hardware at performance problems, even with extraordinarily inexpensive commodity hardware. Even if it can be shown to work in the short run, it rarely ends up being cost-effective in the long run.

Know your data and describe it in meaningful ways. Learn the limitations of the mean and use non-parametric statistics like the median and percentiles when the data warrants it. Learn how to apply other techniques such as Statistical Pattern Recognition both for their increased accuracy and their ease-of-use.

Respond to the threat of new technology by leveraging it. Learn how to take advantage of the World Wide Web's architectural influence, both as a way of segregating the interface from the implementation and as a way of factoring effort over a wide range of systems.

6. References

[Freun68] Freund, John E. and Williams, Frank J., *Modern Business Statistics* Prentice-Hall, Englewood Cliffs, NJ.

[Fukun90] Fukunaga, Keinosuke, *Introduction to Statistical Pattern Recognition, Second Edition* Academic Press, San Diego, CA, 1990, ISBN 0-12-269851-7.

[Jain91] Jain, Raj, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991, ISBN 0-471-50336-3

[Lipsky97] Lipsky, Lester, *How to Model or Simulate Telecommuting Systems Where Power-tail (Bursty, Chaotic, Heavy-Tail, Self-Similar, etc.) Behavior is Observed*. Computer Measurement Group, December 7, 1997, Orlando, FL.

[Peterson97] Peterson, David L., *New Perspectives in DASD Subsystem Cache Performance* Computer Measurement Group, December 7, 1997, Orlando, FL.

[VanTrees68] VanTrees, H.L., "Detection, Estimation, and Modulation Theory: Part I," Wiley, New York, 1968.

[Melsa78] Melsa, J. L., and Cohn, D.L., "Decision and Estimation Theory." McGraw-Hill 1978.

[Bishop97] Bishop, C.M., "Neural Networks for Pattern Recognition." Clarendon Press, Oxford, 1995 (reprinted 1996, 1997)

[Hahn97] Hahn, Brian D., *Essential MATLAB for Scientists and Engineers*, (ISBN 0-340-69144-1, 0-470-25013-5 (Wiley) 1997)