Dynamic Dependencies and Performance Improvement

Marc Chiarini and Alva Couch – Tufts University

ABSTRACT

The art of performance tuning is, alas, still an art; there are few tools to help predict the effects of changes that are motivated by performance needs. In this work, we present dynamic dependency modeling techniques for predicting the effects of configuration changes. These techniques utilize a simple, exterior model of the system under test, and only require codifying the nature of dependencies between subsystems. With a few examples, we demonstrate how to utilize these models to predict the impact of system changes. Using simple tools and a reasoning process, it is possible to answer many questions about possible performance enhancements. The impact of this work is to advance performance tuning a small amount – from being an art toward becoming a science.

Introduction

This paper was conceived in the context of a very practical problem. In a pilot course on service performance analysis, the class undertook a term project to determine how to improve performance at a reasonably large (3000+ simultaneous users) social networking site. The owner reported that the site exhibits self-limiting behavior, in the sense that the number of simultaneous users is limited by and coupled to server capacity. Thus, our problem was to increase server capacity as much as possible within reasonable economic limits. The site was implemented using a LAMP architecture, with the MySQL and Apache server executing on the same host. The owner of the site proposed an initial alternative, which was to move MySQL to an external enterprise cluster with much more CPU capacity. The initial goal of study was to determine whether this proposal was reasonable and – to the extent possible – to predict the benefits of this change before making it (because such a change is expensive to make).

The students began attacking the problem by understanding traditional performance analysis, as defined in Menascé's books [16, 17]. We found out quickly, however, that traditional methods for performance analysis failed to predict the behaviors we were observing. Part of the reason was that our model of application behavior was incorrect, and there were hidden factors in how the application was written that seriously affected performance. The classical theory thus failed to be useful, and the performance results we actually observed did not look anything like what we might have expected. Artifacts in our observations led us to realize that our model of system behavior was quite näive.

For example, memory caching had a dominant effect upon performance. After studying some confusing measurements, we discovered that the application makes extensive use of in-memory caching of MySQL query results (using the memcache PHP library). The critical bottleneck was not the normally CPU-intensive process of uncached query execution, but rather, memcache's intensive use of memory. We concluded that deploying a clustered MySQL server would have at best a minor effect upon performance: a lot of bucks for little or no bang. A better alternative would be to migrate MySQL onto a single server, move all static content to a third server, and give the dynamic content server as much memory in which to cache MySQL results as possible.

This experience was a hard lesson that changed our thinking about service performance prediction in fundamental ways. In a complex system or application, it can be difficult and costly to develop a detailed model of internals. This greatly reduces the effectiveness of traditional performance analysis theory. Single servers are difficult enough to understand, networks of heterogeneous systems even more. What is needed, therefore, are new theories and methodologies for analyzing performance that do not require detailed internal modeling.

This paper is a small first step toward that goal. We attempt to describe, package, and discuss the methodology by which we came to these conclusions. Nothing in this paper is new by itself; it is the combination of elements into a coherent tuning strategy that is new. We hope that this strategy will inspire other system administrators to both employ and improve it for their needs. The ideas here are a start, but in no sense a mature answer.

Overview

Performance modeling and tuning of a complex system by experimental means has a very different form than classical performance tuning [16, 17]. The

approach follows these steps, which we discuss in further detail in the remaining sections:

1. *Factor* the system under test into independent subsystems and resources subject to change.
2. *Synthesize* steady-state behavior via an observation plan.
3. *Determine* dependencies between performance and resources.
4. *Validate* the dependency model by observation.
5. *Reduce* resource availability temporarily.
6. *Measure* the difference in performance between current and slightly-reduced resources.
7. *Compute* critical resources from the measurements.
8. *Expand* the amount of a critical resource.
9. *Repeat* the methodology for the expanded system to note improvement (or lack thereof).

This is nothing more than a structured scientific method. Some of these steps are intuitive, and some require detailed explanation. The main difference between this and trial by error is that we are better informed, and the path to our goal is more strictly guided.

The reasons we present this methodology are threefold. First, although there is a trend toward open-source in IT, there remain a large number of inscrutable closed-source software systems. Detailed modeling (and understanding) of the interaction between such systems is impractical for system administrators. Second, SAs should not need to be software engineers or computer scientists in order to squeeze optimal performance out of their infrastructure. They require tested, transparent, and relatively intuitive techniques for achieving performance gains without greatly sacrificing other goals. Last, the rapid adoption of virtualization will significantly increase the complexity and opacity of performance problems. External modeling will become a necessity rather than an alternative.

In the following, we mix intuition and previously developed rigorous mathematics to inform our approach. It turns out that most of an expert practitioner's intuition about performance tuning has its basis in rigorous mathematics. The flip side of this, though, is that other results arise from the same mathematical basis that are not intuitive to system administrators. So, many of our claims will seem like common sense at the outset, but by the end of this paper, rather useful and counter-intuitive claims will arise.

Troubleshooting Performance

A common question asked by users and IT staff alike is "exactly what is making this service run so slowly?" Classical performance analysis [16, 17] relies on the ability to describe a system under study precisely and in detail. In many environments, this can take more time than it is worth, and obtaining sufficient precision in the description may well be wasteful[1] or impossible (because of technical limitations, closed source, or intellectual property concerns).

If we instead look at the system as a collection of interconnected components, whose internal function remains unknown but whose interdependence is understood, we can make inferences about performance without referring to internals. Since the true service model of some components may well be unknowable, this approach sidesteps knowledge of source code and specific service configurations. Thus, it is equally applicable to open source systems as to, e.g., proprietary SANs or turnkey solutions.

Factoring a Service Into Components

The first step in our methodology is to create a high-level model of a service's dependencies about which we can reason. There are two useful ways to think about relationships between a set of interdependent components: as a set of information flows and as a set of dynamic dependencies. The flow model is more familiar; interdependence arises from request/response behavior. Sometimes, however, requests and responses cannot be easily described, and one must instead assert a dependency without an explicit (and measurable) request/response model.

Modeling Information Flows

Traditional performance analysis describes systems via request flow models. For example, how does a web request get served? First, it arrives at the web server, which makes a request for a file, which is serviced by a file server or filesystem, which returns the file, which is processed in whatever manner the web server wishes, and the results are returned to the user. Thus the model of information flow looks like that in Figure 1.

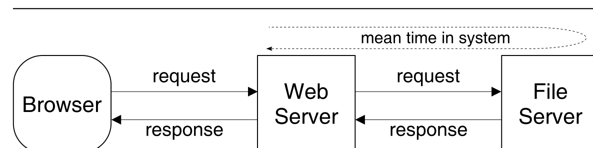

Figure 1: A model of information flow between a web server and its associated file server.

A few basic concepts of this flow model will prove useful. The *mean time in system* for a request is the average amount of time between when a request arrives and the response is sent. What we normally refer to as *response time* is mean time in system, plus network overhead in sending both request and response. In the above diagram, there are two environments in which mean time in system makes sense: the web server and the file server.

It is often useful to express times as their reciprocals, which are rates. The reciprocal of mean time in

[1] We might even say that detailed classical analysis of a performance problem is something like "fiddling while Rome burns." When there is a performance problem, the city is burning down around us and we need to take action. Redrawing a city plan is usually not the first step!

system (in seconds) is *service rate* (requests serviced per second); this is often notated as the symbol μ. The *arrival rate* λ for a component is the rate at which requests arrive to be serviced. Both μ and λ can change over time.

It is common sense that for a system to be in a *steady state*, the arrival rate must be less than the service rate; otherwise requests arrive faster than they can be processed, and delays increase without bound. The concept of steady state is often notated as $\lambda/\mu < 1$.

Steady-State Behavior

The average behavior of a system has little meaning unless the system is in some kind of steady state. The actual meaning of *steady state* is somewhat subtle. We are interested in response time for various requests. Steady state does not mean that response times are identical for each request, but that their statistical distribution does not vary over time. A system is in steady state between time X and time Y if the population of possible response times does not change during that period.

To understand this concept, think of the system being tested as analogous to an urn of marbles. Each marble is labeled with a possible response time. When one makes a request of the system, one selects a marble and interprets its label as a response time for the request. After picking many marbles, the result is a statistical distribution of response times (e.g., Figure 2). More common response times correspond to lots of marbles with the exact same label.

The exact definition of steady state for our systems is that the distribution of possible response times does not change, i.e., there is only one urn of marbles from which one selects response times. No matter when one samples response time, one is getting marbles from the same urn which in turn means that the frequencies of each response time do not vary.

Synthesizing Steady-State Behavior

A real system is seldom in a steady-state condition. Luckily there are ways of obtaining steady-state data for a system in flux by normalizing performance data.

Suppose, for example, that the load on a system changes with the time of day. This is often referred to as a *sinusoidal arrival rate*. A very common technique for dealing with time-varying behavior is to note that sinusoidal arrival rates (e.g., correlated with business hours) can be viewed as steady state if one looks at them for long enough in time. An hour's results may always vary, but when looking at several days worth of data, starting at the same time each day for beginning and end, frequency diagrams look the same regardless of the day upon which one starts.

Likewise, we can also normalize our data into a steady state by sampling behavior that just happens to match in state. Looking at the same hour of the day for several days yields steady-state data. We used this method previously in [8] to model costs of trouble ticket response, by treating each hour of the day as a different sample and combining data for different days (excluding holidays).

Measuring Response Time

There are several ways in which to measure service time. Ideally, it is measured from the point at which a request arrives to the point at which a response is put on the wire. Calculating true service times may or may not be technically feasible, depending on the capabilities of the operating system and/or

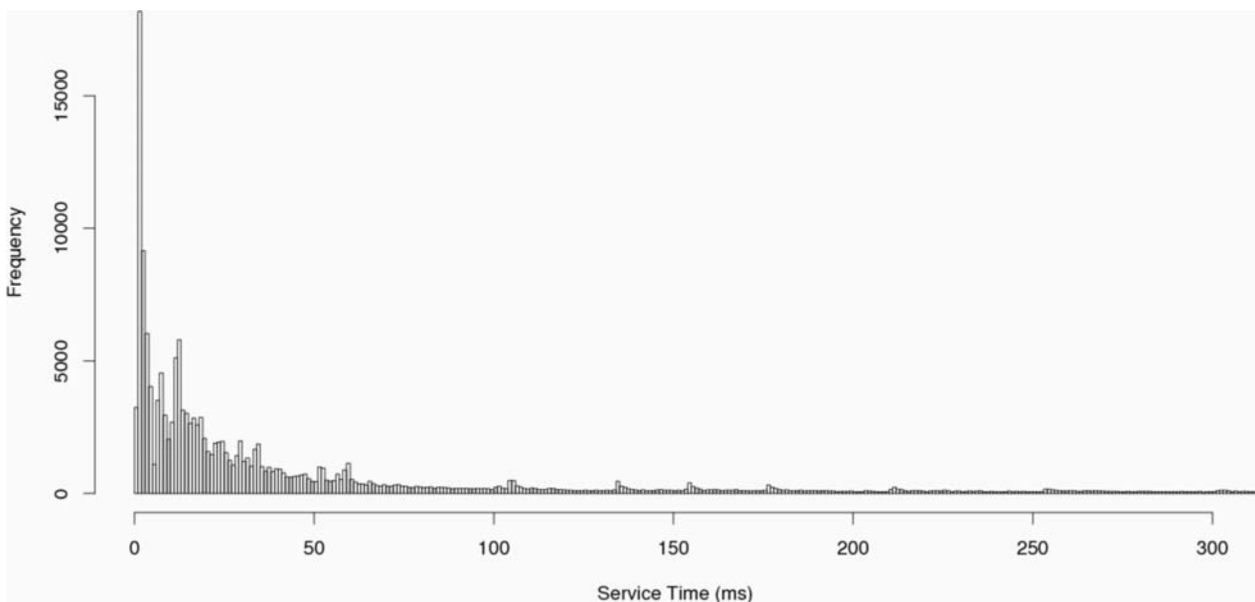

Figure 2: Histogram of response times for multi-class requests over a six hour period. The X axis corresponds to response time while the Y axis represents frequency of that time.

service software. In this work, technical limitations of Apache prevent us from using this measure. Even measurements of socket durations from within the kernel are asymmetric in the sense that while the start time for service is very close to correct, the end time includes the time the response spends in transit. In practice, we can approximate the true service time by sending requests over a high-speed LAN and measuring round-trip times from the source.

To measure response times, we use a benchmarking tool called JMeter [2]. The tool allows us to programatically construct various numbers and types of HTTP requests for static or dynamic content. These requests can then be generated from multiple threads on one or more clients and sent to one or more servers. Additionally, JMeter affords us very fine control over when and for how long tests and sub-tests are run, and the kind of results they collect.

So far, we have run a multitude of simple tests that repeatedly ask a single Apache server for one or more of several *classes* (i.e., file size) of static content. Thus, the components of the service all reside on the same machine. There is no reason, however, that the black box cannot be expanded to include multi-system services or entire networks.

One very useful way to visualize performance of a complex system is via histograms, with response time on the X axis and frequency of that response time on the Y axis. Figure 2 shows a histogram for one 6-hour test. Here, the JMeter client repeatedly asks for a random file from a set of 40 classes. Requests are sent from the client at an average rate of 16.6 per second (Poisson delay of 60 ms). The distribution of service times seems to be exponential, but it is dangerous to conclude this just from eyeballing the graph, especially in the presence of the self-similar "spikes."

We cannot say much about the server just by collecting response time statistics. The server's performance depends in large part on which resources it is being asked to utilize. It is impossible to get a picture of every possible combination of requests. To approach the problem in this manner would limit us to making broad statements about our system only as it is currently configured and under arbitrarily determined categories of load.

We can gain more useful insight by normalizing our performance data. Suppose that we have a very simple probe (akin to a heartbeat monitor, but yielding more information) that continually sends requests to the server at a rate that is intended not to affect performance. At regular intervals, we create histograms of the probe results and observe how they change, e.g., calculate their means, medians, standard deviations, etc. If the system is in a steady state, we can expect that the response time distribution will not change much. As saturation approaches, certain probe statistics will change.

Figure 3 shows response time distributions from probes under increasing background loads. In all cases, not only do statistics such as the mean change, but the probability mass changes shape also. This indicates that the underlying distribution is changing, making it difficult to compare against any theoretically ideal baseline. If we determine that the ideal

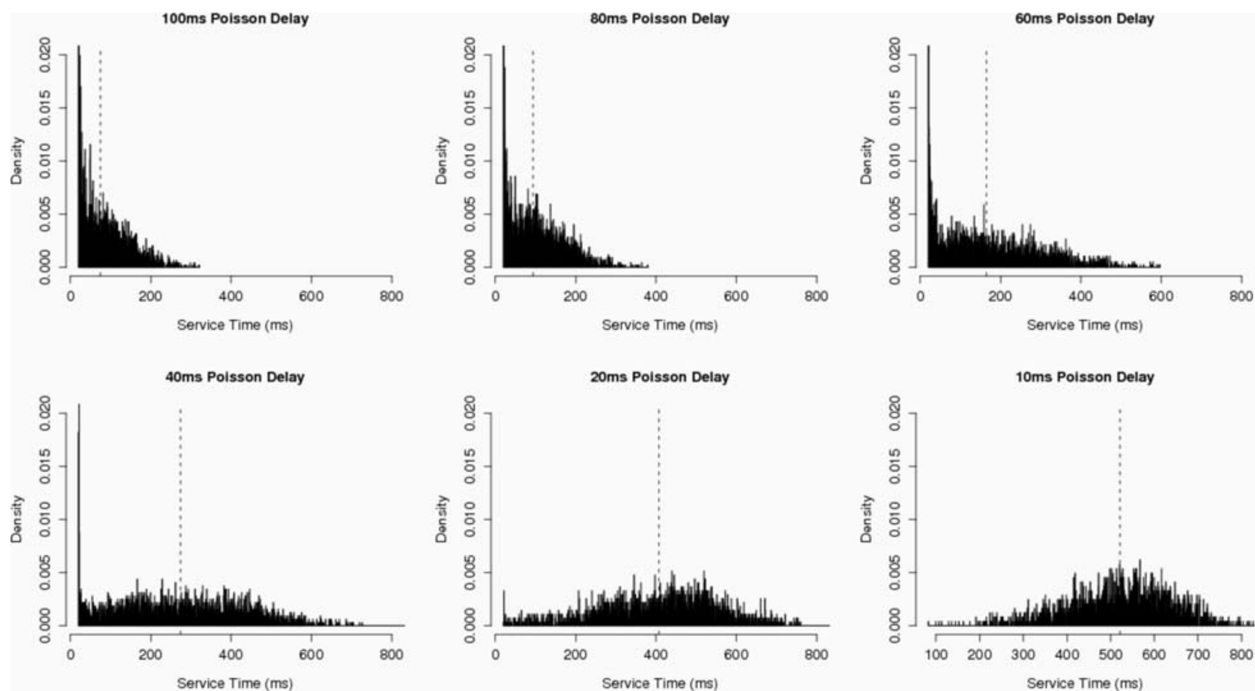

Figure 3: Decreasing the delay between requests (increasing load) has a noticeable effect on the probe's response time distribution. The mean is indicated by the dashed line.

distribution is exponential, what does an observed bimodal or normal distribution tell us about resource utilization? Any goodness-of-fit test for the exponential – which we might use to measure the distance from expected – will fail miserably. This fact motivates our use of non-parametric statistical methods (presented in a later section), which make no assumptions about underlying distributions.

Modeling Dependencies

The nature of information flow within a system determines one level of performance dependencies between subsystems. Our next step is to codify dependencies based upon contention for resources. These dependencies may be more difficult to directly observe; individual requests and responses may be hidden from the analyst. Examples of contention dependencies include the use of memory, CPU cycles, or a network interface among several processes. For these dependencies, there is no (reasonably) measurable notion of a request or response; the system simply slows down (sometimes mysteriously) when there is not enough resource to go around. The reason why something like memory contention is not measurable in terms of a request-response model, is that the act of measurement would slow the system down too much to be practical. The requests and responses do exist (as calls to, e.g., `sbrk`), but we must theorize their presence without being able to observe them directly.

One way we can begin to model these dependencies is via a *dependency graph*. A dependency graph for a system consists of nodes and edges, where a node represents a resource and a directed edge represents a hypothetical dependency between two resources. $A \rightarrow B$ means that A depends upon something that B has.

For example, suppose that the web server and the file server both run on the same host. Then the information flow is just one kind of dependency, and there are three other bounded resources (CPU, disk I/O, and memory) that the servers share. To depict this, we can draw arrows from services to dependencies, as in Figure 4. In this case $A \rightarrow B$ means that A depends upon B in some way. Two arrows to the same place indicate possible contention. One value of such a model is that it can describe, at a very high level, the potential performance bottlenecks of a system, in a manner that is invariant of how the system is actually implemented.

The main purpose of a dependency model is to list – in a compact form – the resource limitations that can affect performance. A useful model describes elements that are subject to change, and ignores elements that one cannot change. For example, if one can expand the memory in a system, but cannot change the processor, modeling in-processor cache size is not particularly useful. In practice, this tends to keep dependency models to a manageable size.

The dependency model corresponds to a flow model, of course, but the details of that underlying flow model may not be useful. For example, disk I/O indeed consists of requests and responses but the overall response time of the system is only vaguely related to disk response; there are many other factors that influence performance. Likewise, use of memory corresponds to requests for more memory and responses that grant access.

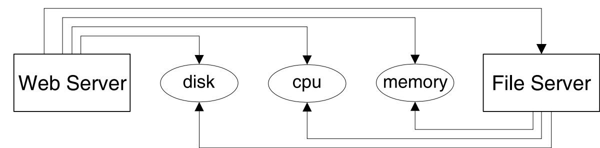

Figure 4: A model of resource conflicts, where two arrows to the same place represent a conflict.

Näive Physics

Dependency models mostly help us reason about performance. The most immediate and easy thing one can do with a dependency model is to utilize what might be most aptly called *näive physics* [20]. The term first arose in the context of artificial intelligence, as a machine model of human intuition in understanding physical processes. A so-called näive model concerns order but not quantity. We know, e.g., given the above model, that if we decrease the memory size, both web server and file server are potentially adversely affected. If we increase memory size, they both potentially benefit. But more important, we also know that if we increase memory size and performance does not improve, then there is some other factor that affects performance that we have not changed. Worse yet, if we change nothing at all and performance worsens or improves, then our model itself is incomplete and/or invalid.

Näive physics are our first defense against formulating an invalid model of a system. When load increases, service time should increase accordingly, and certainly should not decrease. A good understanding of what a decrease or increase should do is enough to allow us to check many simple dependency models.

Validity

Obviously, a dependency graph that omits some important element is not particularly useful. It is best to consider a dependency graph as a hypothesis, rather than a fact. Hypotheses are subject to validation. In this case, the hypothesis can be considered valid (or complete) if changes in resources available within the graph cause reasonable changes in performance at the edges of the system, and invalid (or incomplete) otherwise.

For example, consider a web server. We might start with the initial hypothesis that the web server's performance depends only upon a file server, as above. This means that as file server load goes up for whatever reason, we would expect from näive physics that the web server's performance would decrease or –

at the least – remain constant. Likewise, we would expect that if the file server remains relatively idle, web service should not vary in performance.

Testing this hypothesis exposes a dichotomy between validating and invalidating a hypothesis that will affect all rigorous thinking about our problem. Our hypothesis tests true as long as web server performance *only decreases* when the file server is loaded, and tests false if there is any situation in which, given the same input request rate as before, the file server is not busy and the web server slows down anyway. A point to bear in mind is that hypothesis testing can only ever invalidate a hypothesis; good data simply shows us that the hypothesis is reasonable so far, but cannot prove the hypothesis to be valid.

One potential problem in our model above is that web servers do not necessarily depend just upon file servers. Changing a single line of code in its configuration makes an Apache web server depend additionally (via flow) upon a DNS server to look up all request addresses on the fly. In the context of our model, such a web server must have a *hidden dependency* upon the DNS server. If the DNS server is now explicitly included in our model, then a slowdown in that component offers alternative evidence for why the web server is slow.

Criticality

With a reasonable model in hand, an obvious question is ''which elements of a dependency graph should be improved to make the service respond faster?''

A *critical resource* is one that – if increased or decreased – will cause changes in overall performance. For example, available CPU time is usually critical to response time. If there is more than sufficient memory, then changing memory size will not be critical to performance. If our web server is executing many scripts, CPU time may be the most critical resource, while if it is performing a lot of disk I/O, disk read and write speed may be more critical.

Micro Resource Saturation Tests

Testing criticality of a resource in a complex system is difficult. We borrow an idea from [5], used previously to trace packets statistically in a complex network. A resource is critical to the extent that changing its availability or speed changes overall performance. If we can perturb the system enough to change the availability of one resource, and the whole system's performance responds, then the resource is critical. The extent to which a resource is critical is – in turn – determined by how much of an effect a small change has upon overall performance.

We can analyze the effects of resource perturbation via use of a Micro Resource Saturation Test (mRST). A mRST is a programmatic action that – for a little while – decreases the amount of resources available. We cannot change the amount of memory a system has on the fly,[2] but we can – for a short time – consume available memory with another process. We cannot change network bandwidth on the fly but we can – for a short time – disrupt it a bit by doing something else with the network. If response time consistently changes for the service itself in response to our disruptions, then we can conclude that we are changing the amount of a critical resource, and the extent of the change indicates how critical. Using a simple probe, we can sometimes determine which resources should be increased and which are of no importance.

The reader might ask at this point why we do not simply check the available amount of the resources in which we are interested. The reason is that the kernel's idea of availability may well be biased by what one or more processes are doing. For example, in our first case study above, the web server immediately allocated all available memory, so that the memory always looked full. The reality was that this memory was a cache that was only full for part of the time.

Quantifying Criticality

The purpose of a mRST is to perturb a resource r by an amount Δr and observe the difference in performance Δp. Thus the criticality of a resource r under specific load conditions is a derivative of the form:

$$C_r = \frac{dp}{dr} = \lim_{\Delta r \to 0} \frac{\Delta p}{\Delta r}$$

where p represents service performance, Δp represents the change in performance, r represents the amount available of a resource, and Δr represents the change in a resource. This can be estimated as

$$C_r \approx \frac{\Delta p}{\Delta r}$$

for a specific Δr we arrange, and a specific Δp that we measure.

Note that for a multiple-resource system with a set of resources r_i, criticality of the various resources can be expressed as a gradient

$$\nabla p = \Sigma_i \frac{\partial p}{\partial r_i} \approx \Sigma_i \frac{\Delta p}{\Delta r_i}$$

This suggests that one way to tune systems is via gradient ascent, in which one increases a resource in the direction of maximum performance improvement.

Resource criticality is a concept that is only reasonably measured relative to some steady state of a system (in which input and output are in balance). Obviously, resources are only critical when there is work to be done, and the above gradient only makes sense when that work to be done is a constant.

Cost Versus Value

Remember that the objective here is to get the best performance for the least cost. Thus criticality is best defined in terms of the cost of a resource, rather than its value. One can think of the units for r in $\partial p / \partial r$

[2] The advent of virtualization does make this possible, but it is not likely to be practical in a production environment.

as a cost increment rather than a size increment, so that the units of the derivative are *rate*/*cost* [8]. Often, this is a finite difference because cost is a discrete function, e.g., memory can only be expanded in 128 MB chunks.

Interpreting mRST Results

Critical resource tests can be interpreted either via näive physics or statistics. From a näive physics point of view, resources are either non-critical ($\partial p/\partial r \approx 0$) or critical ($\partial p/\partial r > 0$), where p is the average response rate (1 divided by average response time). Increases in p represent increased performance and increases in r represent increased capacity.

The näive view may suffice for most practice, but we would be served better by a more rigorous approach. From a statistical point of view, some resources can be characterized as more critical than others, but the definition of criticality is rather tricky

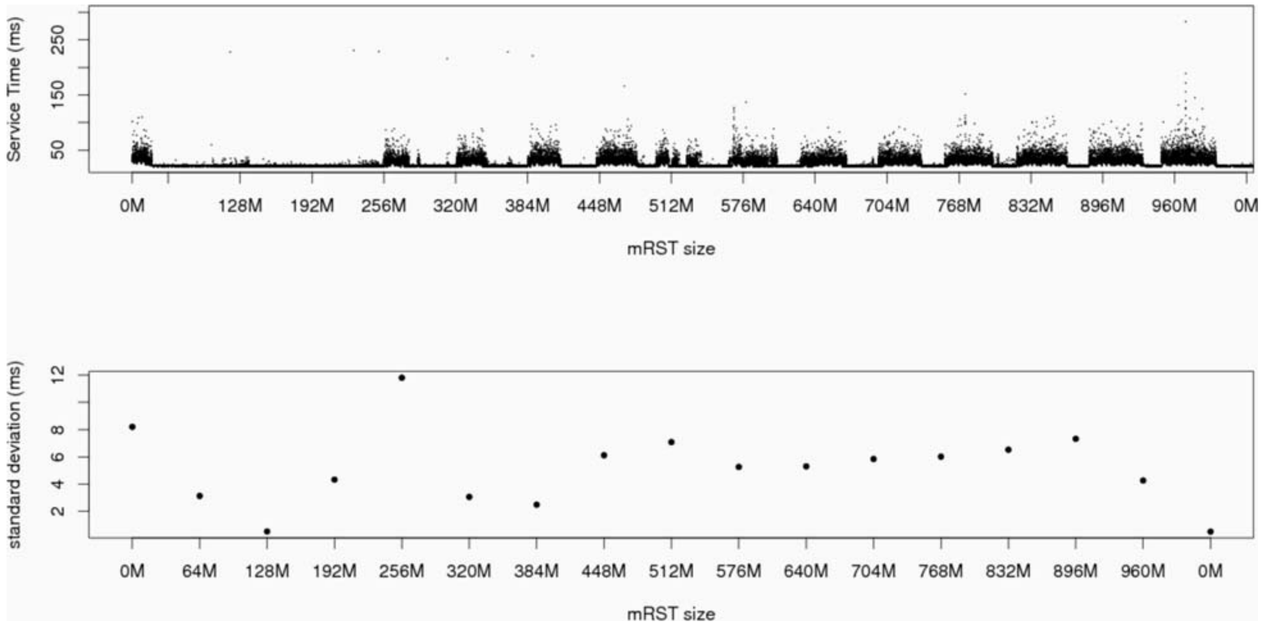

Figure 5: (top) The raw response times of an Apache web server during and between increasing memory mRSTs. The initial perturbation at 0M is due to the loading of the operating system's page-cache. (bot.) The variability (standard deviation) of response times does not change much from test to test.

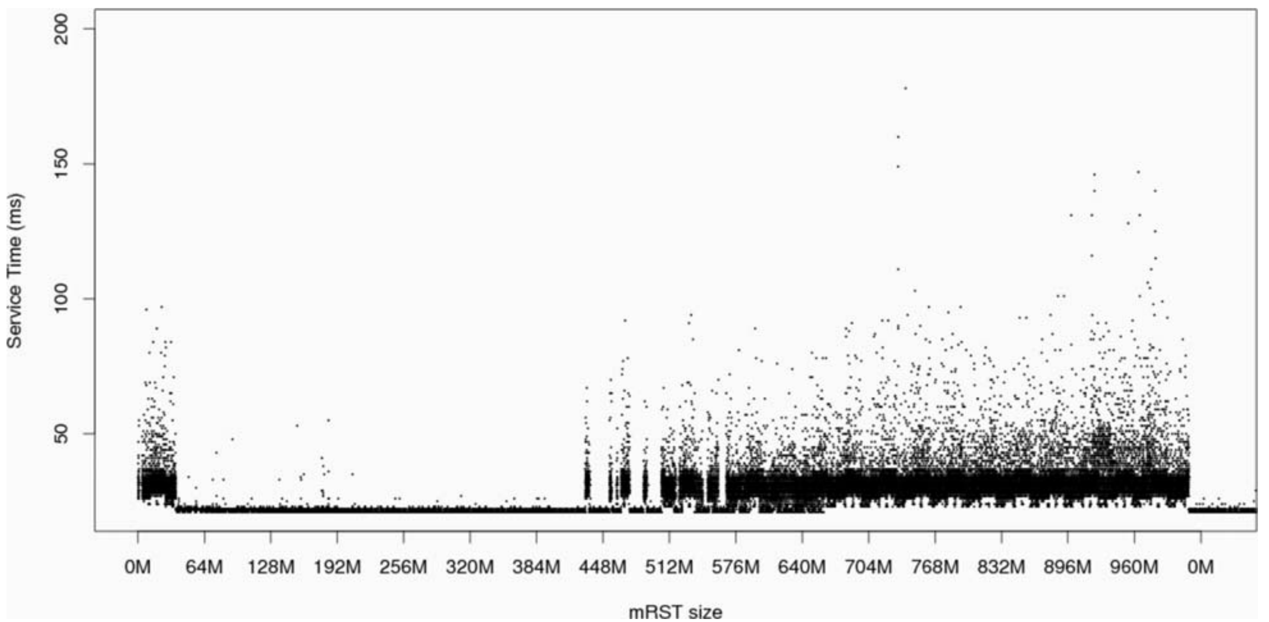

Figure 6: Adding to memory usage with a series of increasing mRST loads shows a clear plateau where memory becomes critical.

and the actual inferences somewhat more subtle than when utilizing näive physics.

Example: Perturbing Memory Use

We can see the effects of perturbation through a simple example. We synthesize steady-state behavior by creating a constant load of probe requests on a server. Figure 5 shows the raw response times of the server when hit with a sequence of memory perturbations. Our test run continually requests a sequence of different files totaling 512 MB in size. The server is equipped with 1 GB of RAM and no swap space. The initial variations in response time (at 0 MB) are a result of the OS filling the page-cache from disk. Each mRST works by obtaining a specified amount of memory and freeing that memory after ten minutes. The lulls (reductions) in response time between perturbations are an indication that the server has returned to an unstressed state and is serving the requested files from cache. There is a relative lack of variation in response times after the page-cache has loaded and up until the 256 MB mRST. This indicates a critical point in memory usage. Also, while there is no specific trend in the standard deviation of the response time (aside from an anomaly at 320 MB), the settling time back to undisturbed performance increases with the size of the mRST. This can be observed in the increasing width of the perturbations. Alternatively, one can observe the same critical point for memory without waiting for the system to reset (Figure 6).

The other case – in which the resource under test is not critical – is shown in Figure 7. In this case, 10 minutes into the test, 512 processes are spawned to compete for CPU cycles for the next 10 minutes. This does not affect response time noticeably, so we conclude that CPU cycles are not critical. Note that this test is much more convincing than load average, because the interpretation of load average changes depending on the OS platform and the architecture of a given machine. In this case, the load average becomes very large, but the system still responds adequately. One must take care, however, when systems are near saturation. Figure 8 shows a series of increasing mRSTs after a system is already saturated. The perturbations in this case do not have a significant effect upon behavior, as shown by the histograms found in Figure 9, which are virtually identical.

Reasoning About Criticality

Reasoning about what is critical is sometimes difficult and counter-intuitive. The system under test is in a dynamic state. It is important to distinguish between what one can learn about this state, and what aspects of performance are unknowable because there is no mechanism for observing them, even indirectly.

For example, the question often arises as to what to change first in a complex system, in order to achieve the most improvement. Testing via mRST does not tell us how much improvement is possible, but does tell us the rate of change of improvement around current resource bounds. If there is no change, a resource is not critical and need not be enlarged and/or improved. After an adjustment, another set of mRSTs is required to check criticality of the new configuration.

For example, suppose that we manage to take up 10% of 1 GB of memory for a short time and overall performance of the system under test decreases by 5%. This does not indicate exactly what will happen if we expand the memory to 2 GB, but instead, bounds any potential improvement in performance. If memory is the only factor, we could perhaps expect a performance

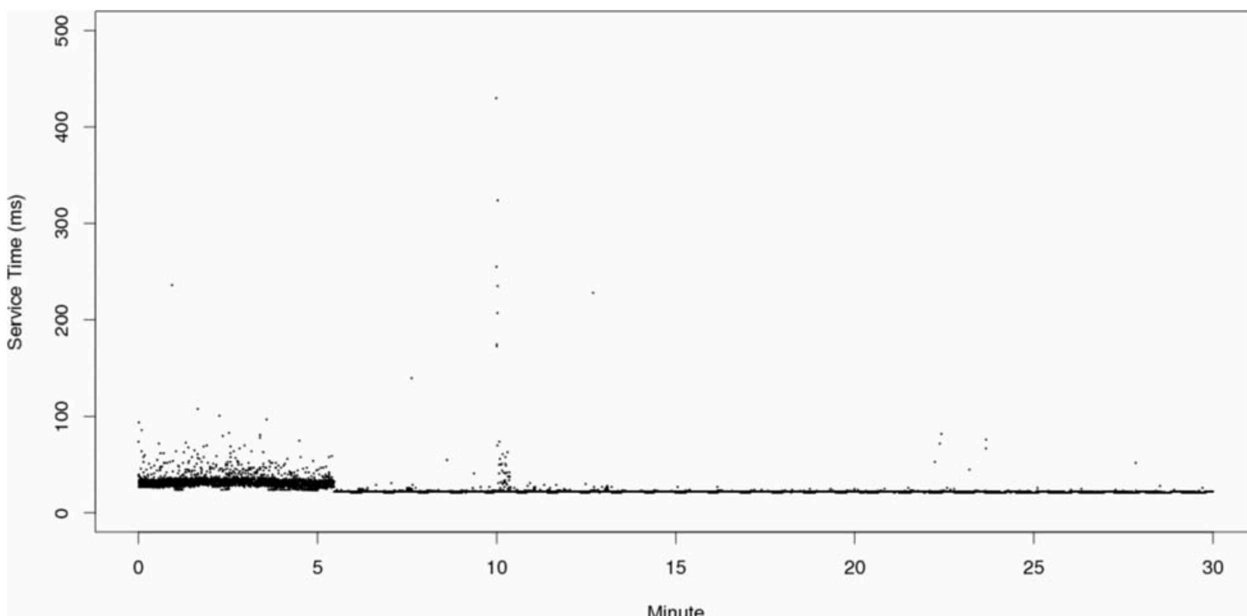

Figure 7: A CPU-time mRST conducted 10 minutes into a test shows almost no criticality of CPU resources.

improvement of 50% (presuming 5% per 100 MB) but this is unlikely. It is more likely that memory will cease being a constraint and another resource (e.g., CPU) will become most critical long before that happens, and that the overall benefit from memory increases will plateau.

To better understand this concept, note that the measurements we can make via mRST are only *point rates* and not global rates. We can measure what will happen to performance if we make a small change in a resource, but this estimation only applies to changes near the estimation. In other words, if we manage to temporarily consume 1 MB of 1024 MB of memory, and performance drops by, e.g., 5%, we can only conclude that adding 1 MB to an existing 1023 MB might improve performance by 5%, and not that adding 1 MB to 1024 MB might improve it by another 5%. We

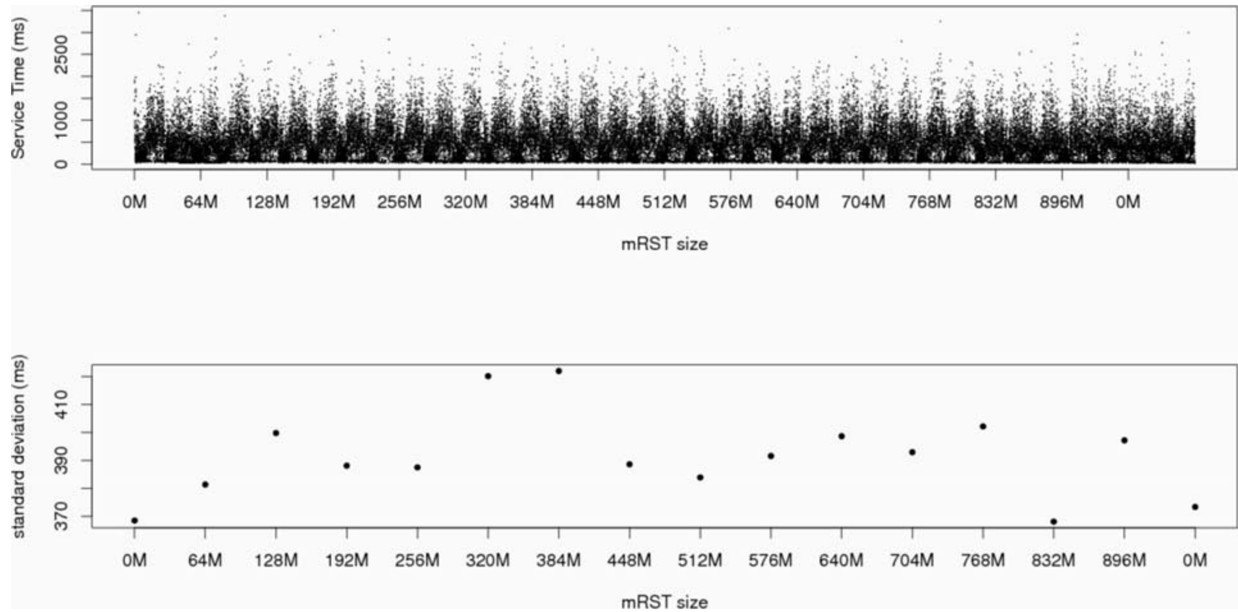

Figure 8: (top) Increasing sequence of mRSTs after saturation with no lulls between increases shows no significant change in behavior. (bot.) Variability in response times during each mRST is more significant than in the case with resting time.

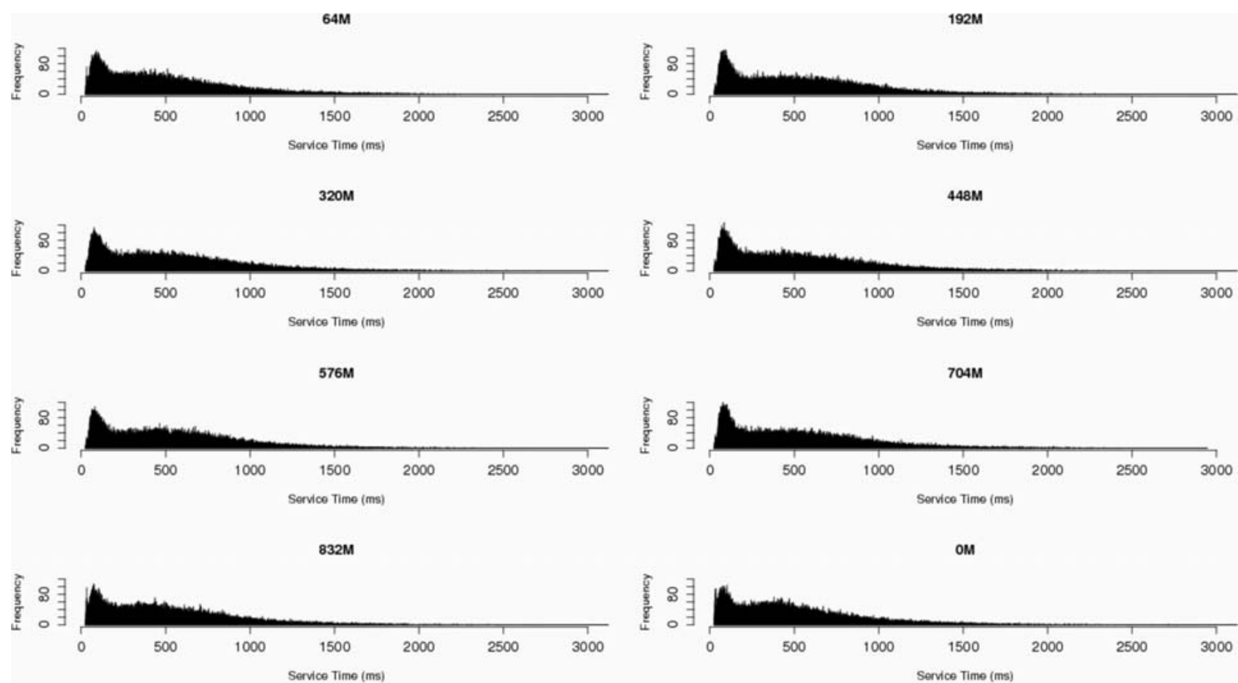

Figure 9: Histograms of response time for progressively larger memory mRST of an already saturated system shows that there is little change in distributions as resource availability decreases.

can be reasonably sure that adding 1 MB to 1024 MB will improve things a little, but we have no information (from the test in hand) as to when we will have added enough memory that the addition of more will no longer be critical.

Statistical Inference

So far, we have discussed a methodology based upon a mix of direct measurement and intuitive reasoning. While this is certainly better than trial by error, there are also powerful mathematical tools we can bring to bear to make this process more rigorous and accurate.

The long-term goal of this work is to develop a cogent statistical inference methodology that allows one to infer – with reasonable certainty – whether system attributes are critical or not. We would like, e.g., to be able to say that ''with 95% certainty, resource X is critical.'' An ideal technique would also measure the extent to which X is critical.

The key question when we observe a change in a system is whether there is a real change. Suppose that we observe an increase in mean response time between two experimental conditions. It could be that this is a true underlying change, but it is also possible that the difference in measurements is due to random factors other than resource criticality.

Returning to the urn analogy, if we measure response time under two conditions, then the basic question is whether there is one urn or two. If the conditions are different, then this is analogous to two different urns from which marbles are being selected. If they are the same, this is analogous to drawing both sets of marbles from the same urn. We need to be able to determine, to some acceptable degree of certainty, whether there are really two different urns.

The key to answering this question is to consider chance. At any time, any marble in the urn could be selected. Thus it is possible, though not very likely, that one could select the same marble over and over again, ignoring the others. This would lead to a measured response time distribution with a different mean, even though we are choosing from the same urn as before.

There are many tests that statisticians use to compensate for randomness in data. Each test utilizes some model of how randomness can arise in the data, and allows one to compute the probability that two sample distributions differ by chance even though the distributions are collected under the same conditions. This probability allows one to judge whether an observed difference is likely to be large enough to matter.

Our tools for this task are non-parametric statistical models and tests [21] that do not assume any particular distribution of data. As illustrated in the figures above, the data we observe has no clear relationship to well-known statistical distributions. Moreover, the true population distribution likely changes under different circumstances. This is important because if we were to use traditional parametric tests such as, e.g., Pearson's chi-square, on data that are not normally distributed, the results would be unreliable or inconsistent at best.[3] The goal of the following section is to compute the probability that there is one urn rather than two. If this probability is high, one cannot confidently claim that observed differences are meaningful; in statistical terms, the differences would not be *significant*. Fortunately for us, this problem arises frequently in the social sciences and we can borrow tools from statistical analysis to help. In order to do this, however, we must carefully analyze our experiments and data, and select tools that apply to the task.

Non-parametric Statistics

One key to applying a statistical technique is to first consider the assumptions required and ensure that it is safe to make those assumptions about one's data.

Statistical techniques that show promise for determining criticality of a resource include the Mann-Whitney-Wilcoxin rank-sum test (U test) [14], and the Kolmogorov-Smirnov test (K-S test) [7]. These tests are robust in the sense that they still perform well when certain distributional assumptions are violated. They are also generally less powerful than their parametric counterparts, which means that they are more likely to generate false negatives [19]. There are ways of improving this, but that discussion is beyond the scope of this paper.

Both of these tests take as input two sample distributions of data, and test whether differences between the samples could have arisen by chance. The output of the test is a *p-value* that describes how likely it is that observed differences are due to chance alone. A low p-value (e.g., $p \leq 0.05$) is evidence that the populations from which the samples are drawn differ, but a high p-value does not give evidence that the distributions are the same. This counter-intuitive result partly arises from the fact that we are sampling the populations; the fact that a particular sample of a population conforms does not mean that the entire population conforms, but if two samples differ sufficiently, this is evidence that the populations differ as well.

To apply these tests meaningfully, however, one must satisfy their requirements. Both the U test and K-S test require that individual samples (e.g., the response times for single requests) are independent. Two samples are independent if the response time for one cannot affect the other's response time, and vice versa. Unfortunately for us, independence does not arise naturally and must be synthesized. Two requests that occur close together in time are not independent, in the sense that they may compete for scarce resources

[3] If the population can be made to approximate a normal distribution by, e.g., collecting large enough samples, then the test would work correctly. However, to make our methodology as efficient as possible, one of our goals is to minimize sample sizes while still obtaining valid results.

and affect each other. But, if one considers two requests that are far enough apart in time, say, 10 seconds, there is no way in which these can compete, so these can be considered independent.

We can use the non-parametric tests in our performance analyses as follows. First we collect data for a steady state of a system under test. We then perturb the system by denying access to some resource and collect data during the denial. These two data sets are both discrete distributions of service response times. We apply a test to these data. If the p-value for the test is $p \le 0.05$, we can conclude that the two distributions are different, while if $p > 0.05$, results are inconclusive and the difference could be due to randomness alone.[4] Here great care must be taken to avoid erroneous conclusions. Suppose we run the K-S test on two sets of measurements known to be from the same population and they test as significantly different. We know that the K-S test requires that the system be in a steady state, and that the measurements are independent of one another. Thus if the test returns a result we know to be false, then something must be wrong with the way we are applying it: either the system is not in a steady state, or measurements are not independent.

We can exploit this reasoning to test independence of measurements. If we apply the K-S test to two samples from one population, and we know the system is in a steady state, and the test finds significant differences, then we can conclude that either measurements are not independent or the sample size is too small, because all other assumptions of the test are satisfied and the test should not identify significant differences.

Thus we adopt a methodology of grounding our assumptions by synthesizing an environment in which our tests do not flag samples of the baseline steady state as significantly different. To do this, we:

a) probe the system regularly at intervals long enough to assure independence of measurements.
b) validate our probes by assuring that two sets of measurements for the same system state could occur by random chance.
c) create another system state via mRST.
d) check whether that state yields a significantly different population of probe measurements.

The point of this technique is that a system is defined as having a steady state if any deviations in measurements could occur by random chance, so that comparisons between that state and any other then make sense. Otherwise, we cannot be entirely sure that we are applying the test correctly.

Related Work

Systems performance analysis has been conducted for many years and in many contexts, such as hardware design, operating systems, storage, and networking. Many books have been written that focus on the practical sides of analysis such as experimental design, measurement, and simulation [10, 13, 16, 17]. These present comprehensive, general-purpose methods for goals such as capacity planning, benchmarking, and performance remediation. The scope of our research is more narrow; it is less about traditional performance analysis, and more about discovering performance dependencies. It is targeted to a system administration community that is frequently asked to perform small miracles with a deadline of yesterday and at the lowest possible cost.

Several past works have attempted to describe what is normal "behavior" or performance for a system, either for specific classes of load or in a more general fashion [4, 6, 12]. Detecting behavioral abnormalities is also frequently studied in a security context [9] and for fault diagnosis [11, 15]. Since our own work relies on perceived or statistical changes in system response, it is to a significant extent, compatible with many such approaches. In fact, time series analysis techniques such as those found in [4] could be used inside our methodology, but we can make no claims about their efficiency in discovering critical resources.

Finally, prior work that uses black-box testing to uncover performance problems [1, 3, 18] is similar to our own in philosophy, but not in purpose. These research projects attempt to locate nodes that choke performance in a distributed system by analyzing causal path patterns.[5] Our methodology can be adapted to function on larger black boxes such as a distributed system simply by scaling, i.e., we can factor a distributed system into distinct communicating subsystems, e.g., nodes or groups of nodes, and/or resources, e.g., individual services. Once we have located problem nodes, we can apply the process again to the resources on those particular nodes.

Future Work

This paper is just a start on a rather new idea. We plan on eventually embodying the thinking presented here into a toolkit that can aid others in doing performance analysis of this kind. This requires, however, much more groundwork in the mathematics of statistical inference, so that the toolset can enable appropriate statistical reasoning with little or no error or misinterpretation. Hand-in-hand development of tests and statistical reasoning will be paramount.

Conclusions

Resource dependency analysis and performance tuning of complex systems remains an art. In this paper, we have made a small start at turning it into a science. Our initial inspiration was that classical methods of performance analysis that require a complete factoring of a system into understandable and modeled

[4] A 0.05 significance level is typically used when testing hypotheses. It represents the probability of finding a difference where none exists.

[5] And in the case of [3], also utilizing some whitebox information.

components are not particularly useful in a complex environment. Instead, we rely upon a partial factoring into a high-level entity-relationship model that reveals the bonds between services and resources. We can quantify those relationships via näive physics or via statistical inference. This gives us a relatively quick way to flag resources to change, and to check whether our intuitive ideas of what to change are valid.

Acknowledgements

We are grateful to the students of *Comp193SS: Service Science*: Hamid Palo, Ashish Datta, Matt Daum, and John Hugg, for sticking with a hard problem and coming to some surprising results without knowing just how useful their thinking would eventually turn out to be.

Author Biographies

Marc Chiarini is completing his Ph.D. at Tufts University. He previously received a B.S. and M.S. from Tufts in 1993 and an M.S. from Virginia Tech in 1998. His research interests lie in system configuration management, system behavior and performance, machine learning, and autonomic computing. Before studying at Tufts, he worked as a UNIX system administrator and toolsmith for a wide variety of companies in the San Francisco Bay area. All of his computer prowess derives from the TRS-80 COCO Introduction to BASIC manuals that he voraciously consumed at the age of nine. Marc will become a first-time father in the spring of 2009. He can be reached via electronic mail at marc.chiarini@tufts.edu .

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M.I.T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Computer Science at Tufts. Prof. Couch is the author of several software systems for visualization and system administration, including Seecube (1987), Seeplex (1990), Slink (1996), Distr (1997), and Babble (2000). He can be reached by surface mail at the Department of Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@cs.tufts.edu .

Bibliography

[1] Aguilera, Marcos K., Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen, ''Performance Debugging for Distributed Systems of Black Boxes,'' *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pp. 74-89, New York, NY, 2003.

[2] *Apache JMeter home page*, http://jakarta.apache.org/jmeter/ .

[3] Barham, Paul, Austin Donnelly, Rebecca Isaacs, and Richard Mortier, ''Using Magpie for Request Extraction and Workload Modelling,'' *OSDI*, pp. 259-272, 2004.

[4] Brutlag, Jake D., ''Aberrant Behavior Detection in Time Series for Network Monitoring,'' *LISA '00: Proceedings of the 14th USENIX Conference on System Administration*, pp. 139-146, Berkeley, CA, USA, 2000.

[5] Burch, Hal and Bill Cheswick, ''Tracing Anonymous Packets to their Approximate Source,'' *LISA '00: Proceedings of the 14th USENIX Conference on System Administration*, pp. 319-327, Berkeley, CA, USA, 2000.

[6] Burgess, Mark, Hårek Haugerud, Sigmund Straumsnes, and Trond Reitan, ''Measuring System Normality,'' *ACM Transactions on Computer Systems*, Vol. 20:, pp. 125-160, 2002.

[7] Conover, W. J., *Practical Nonparametric Statistics*, pp. 428-441, John Wiley & Sons, December, 1998.

[8] Couch, Alva, Ning Wu, and Hengky Susanto, ''Toward a Cost Model of System Administration,'' *LISA '05: Proceedings of the 19th USENIX Conference on System Administration*, pp. 125-141, Berkeley, CA, USA, 2005.

[9] Denning, Dorothy E., ''An Intrusion-Detection Model,'' *IEEE Transactions on Software Engineering*, Vol. 13, pp. 222-232, 1987.

[10] Gunther, Neil J., *The Practical Performance Analyst: Performance-by-Design Techniques for Distributed Systems*, McGraw-Hill, Inc., New York, NY, USA, 1997.

[11] Hood, Cynthia S. and Chuanyi Ji, ''Proactive Network Fault Detection,'' *INFOCOM*, p. 1147, IEEE Computer Society, 1997.

[12] Hoogenboom, P. and Jay Lepreau, ''Computer System Performance Problem Detection Using Time Series Models,'' *USENIX Summer 1993 Technical Conference*, pp. 15-32, Cincinnati, OH, 1993.

[13] Jain, Raj, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley-Interscience, New York, NY, USA, 1991.

[14] Mann, H. B. and D. R. Whitney, ''On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other,'' *Annals of Mathematical Statistics*, Vol. 18, pp. 50-60, 1947.

[15] Maxion, R. A. and F. E. Feather, ''A Case Study of Ethernet Anomalies in a Distributed Computing Environment,'' *IEEE Transactions on Reliability*, Num. 39, pp. 433-443, 1990.

[16] Menascé, Daniel A. and Virgilio A. F. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*, Prentice Hall, Englewood Cliffs, NJ, USA, second edition, 2001.

[17] Menascé, Daniel A., Lawrence W. Dowdy, and Virgilio A. F. Almeida, *Performance by Design: Computer Capacity Planning By Example*, Prentice Hall PTR, 2004.

[18] Reynolds, Patrick, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat, ''Wap5: Black-Box Performance Debugging for Wide-Area Systems,'' *WWW '06: Proceedings of the 15th International Conference on World Wide Web*, pp. 347-356, New York, NY, USA, 2006.

[19] Siegel, Sidney and N. John Castellan, *Non-Parametric Statistics for the Behavioral Sciences*, McGraw-Hill, NY, NY, USA, second edition, 1988.

[20] Smith, Barry and Roberto Casati, ''Näive Physics: An Essay in Ontology,'' *Philosophical Psychology*, Vol. 7, Num. 2, pp. 225-244, 1994.

[21] Wasserman, Larry, *All of Nonparametric Statistics (Springer Texts in Statistics)*, Springer, May, 2007.