A metric for predicting the performance of an application under a growing workload

by E. J. Weyuker
A. Avritzer

A new software metric, designed to predict the likelihood that the system will fail to meet its performance goals when the workload is scaled, is introduced. Known as the PNL (Performance Nonscalability Likelihood) metric, it is applied to a study of a large industrial system, and used to predict at what workloads bottlenecks are likely to appear when the presented workload is significantly increased. This allows for intelligent planning in order to minimize disruption of acceptable performance for customers. The case study also outlines our performance testing approach and presents the major steps required to identify current production usage and to assess the software performance under current and future workloads.

In this paper we introduce a new metric, the Performance Nonscalability Likelihood (PNL) metric, intended to be used to predict whether a software system is likely to encounter performance problems when the workload is significantly increased. Here we use the term *scalability* to refer to a software system's ability to handle such increased workloads. We also include a case study using a newly developed software system for a large customer care service provider. In the case study, we describe our approach to the performance testing of the customer care database system, and also the workload characterization, the test design, the performance measurement results, and our recommendations for performance improvement. We also describe the computation of the PNL metric for this system to show the utility of making predictions of this sort, and to demonstrate how to collect and analyze the necessary data for a system of this magnitude and complexity.

Our personal experience is that most performance testing efforts create the test load without having done a careful workload characterization in the production environment. It is true that collecting the necessary data and doing the analysis are often difficult and costly, but we have found that the use of field-collected data is an extremely valuable asset that has allowed us to accurately predict the behavior of the system under varying workloads, and thereby allowed us to plan for, and prepare for, likely bottlenecks that might have catastrophic consequences were they to occur in the field. Even for new systems, such as the one we describe in our case study, we have found that it is often feasible to collect the data necessary to describe the production workload. This might happen because the input to the system under examination is the output of some already existing software system or device. Therefore, even though the software system being assessed is new, the workload that is being characterized is not. For example, we were involved in doing load testing for a new system that was to be used to issue alarms when certain combinations of events occurred at a hardware switch. Prior to the implementation of this system, the assessment of the switch outputs and the determination of the appropriate actions had been manually accomplished. Nonetheless, the switches

had been issuing messages for many years, and a careful analysis of these messages led to an accurate characterization of the new system's expected workload.

Another reason why the workload of a newly built software system might be available is that its input space is the same as that of another existing system. Thus, for example, a company's customer base is likely to exist for billing purposes, including data describing customers' usage habits. A new software system may then be built to make targeted offerings of new goods or services based on data derived from the existing database information, or a new fraud-detection system might rely on an analysis of this data. In either case, the inputs to the new system would be the same as those for an existing system and therefore could be collected and analyzed.

Still, there are situations under which it is not possible to collect this type of data, in which case the approach we discuss will not be usable in that environment. A new system being deployed in production may not have a target system from which to collect production data. Even when production systems exist, they may be very difficult to instrument for data collection. Other possible reasons are the complexity of the system under test and the lack of personnel trained in performance testing and data collection techniques, or a belief that such efforts are too expensive or are unnecessary. In addition, the same data can be used to prioritize fault removal efforts and also as the basis for determining when to stop testing. A central conclusion of this work is that collecting the types of workload data that was necessary for our approach is often feasible, and when it can be done, it is well worth the necessary resources.

For this project, we used field data collection to characterize current production usage, sometimes known as an *operational distribution* or *operational profile*, and used performance testing to identify the bottleneck resources of interest to the analysis. A thorough description of the collection of operational profile data is available in Reference 1, and its use in test case selection during the load testing phase is described in Reference 2. Specifically, we have used the operational distribution, which is derived at test planning time, as the basis for test case generation, along with Markov chain modeling. Markov chains have also been used to predict software reliability[3] and to capture usage statistics from empirical data.[4]

Selecting performance tests based on the operational distribution guarantees that the testing done in the test lab provides an accurate image of the actual field environment, and we used the operational distribution to determine an efficient order in which to concentrate our fault removal effort. This guaranteed that failures observed during performance testing receive priority according to the expected impact they will have in the production environment. In contrast, when the performance test design has not been based on production data, it is difficult to determine how to best use scarce resources to meet critical project needs.

For the specific system that will be used in our case study, we needed to validate the performance requirements for a system that had been acquired off-the-shelf. This meant that we did not have access to design documents or the code itself and, therefore, we could perform neither design reviews nor code reviews. We were assured that thorough functional testing at the unit, integration, and system test levels had been done by the software vendor, prior to the start of performance testing, but, of course, it was not possible to quantify the comprehensiveness in the way that one can for software that is produced in-house. In our organization, as in many industrial software production sites, the quality of functional testing is typically evaluated using specification and code coverage metrics, while performance testing is usually benchmarked against system-wide quality metrics such as performance and reliability requirements.

There are certainly advantages to doing performance testing throughout the software life cycle, just as there are advantages to doing correctness testing during development. Nonetheless, since the usual procedure in our organization for both systems developed in-house, and those purchased off-the-shelf, is to focus performance testing effort after the system is fully developed and thoroughly tested for correctness, our performance testing approach and the PNL metric are predicated on this assumption.

The paper is organized as follows. The next section, "Predicting scalability," defines our PNL metric, designed to predict whether or not there are likely to be performance problems when the workload is significantly increased. Then, in the section, "Deriving the operational distribution," we describe the way we determined the operational distribution for the customer care service provider that will serve as the system in our case study. In the section "Performance

testing," which follows, we present our approach to performance test design and the main conclusions derived from the performance testing effort for the customer care system. The section "Computing the PNL" demonstrates the computation of the PNL metric for the customer care system, and the last section contains our conclusions.

Predicting scalability

In this section we describe a way to predict the ability of a system to handle significantly increased workloads. This will be known as the *Performance Non-scalability Likelihood* (PNL) metric. In the section "Computing the PNL," we apply it to a production system. Being able to determine whether performance problems are likely to occur when the workload becomes significantly heavier than its current level will allow project personnel to plan for necessary server capacity upgrades in a way that is transparent to users.

The PNL metric is designed to capture the expected performance loss at a given load value. This is done by distinguishing between states for which the behavior is acceptable and those for which it is not. Although we designed this metric in response to the needs of a particular project, namely the one that is included in this paper as our case study, we have defined the metric generally, and will then describe how we particularized it in ways that were most appropriate for this project.

$C(s)$ is a reward function that maps state s to a value within the closed interval [0, 1]. It indicates whether performance degradation has occurred, and may include an indication of the degree of that degradation. The coarsest way of defining $C(s)$ simply distinguishes between acceptable behavior (for which $C(s)$ is set to 0) and unacceptable behavior (for which $C(s)$ is set to 1). For the system used in the case study, this is the most appropriate distinction—the performance is either acceptable or it is not. For other systems, however, this might not be a subtle enough distinction. For example, when modeling the performance of real-time systems, every state that fails to meet the deadline could be considered a failed state and therefore assigned the value 1, while the reward assigned to states that meet the deadline could be computed using a function of the distance from the real-time deadline, which is mapped to the appropriate range. Conversely, when dealing with non-real-time systems, all states that meet the performance objective could be considered good states and for those states $C(s)$ would be set to 0, while the penalty for missing the performance objective could be proportional to the distance from the objective. For a discussion of similar issues in the assessment of software reliability, see Reference 5.

The PNL metric is designed to capture the expected performance loss at a given load value.

Other related work appears in the extensive literature on performability. Two particularly useful sources in this area are References 6 and 7, both of which have very comprehensive bibliographies. *Performability* is defined as the probability that the system under study performs at the required performance level.[6] It is designed to be used when there is a need to evaluate *both* the performance and dependability of computer systems, particularly when the system's performance is degrading because of faults. A stochastic process is defined, which is used to model the system under study. The model is then used to assign values to performability metrics for a certain operating environment. Different types of reward variables are used, depending on the time during which the rewards are accrued.

Therefore, performability modeling could be used to encompass any problem that involves a reward structure, and in that sense is closely related to the PNL metric that we introduce here. Although the most common applications of performability theory are to degradable system components in the presence of failures, in either the hardware or software domain, in this paper we are interested in predicting the likelihood that a software system under study will fail to meet its performance goals when the workload is scaled. Thus, although similar techniques are involved, the work presented here has a fundamentally different perspective. We are interested in predicting the *future* performance behavior of a system that does not fail, given that it is going to have to handle a significantly heavier workload than it currently has to handle. In contrast to this, performability modeling is concerned with assessing the performance of a system that is degrading because of failures.

Table 1 Page request distribution on the servers

Agent-Facing Page Type	Agent-Facing Percentage	Customer-Facing Page Type	Customer-Facing Percentage
Static page	50	Static page	23
Error code page	10	Error code page	23
Search form	7	Search form	30
Search result	8	Search result	16
Other pages	25	Other pages	8

The formal definition of PNL, for program P relative to operational distribution Q is:

$$PNL(P, Q) = \sum Pr(s)C(s) \quad (1)$$

where $Pr(s)$ denotes the steady state probability associated with state s, under the operational distribution Q, and $C(s)$ denotes the degree of performance degradation, with $C(s) = 1$ indicating that the performance is unacceptable, and $C(s) = 0$ denoting that the performance is entirely acceptable. The range of $C(s)$ is the interval [0, 1].

The intuition behind our metric is that PNL includes the probability mass associated with exactly those states whose performance behavior is considered to be unacceptable. Therefore, the PNL metric captures the probability that the system will not meet its performance objective for a given load value.

To compute the PNL metric, the following process should be followed:

- The performance objective is defined in terms of the maximum acceptable response time.
- The arrival process distribution is determined.
- The service time distribution is determined.
- The system under study is modeled to generate a state definition that captures the performance metric of interest.
- The model is solved to generate the state probability distribution.
- The PNL metric is computed from the state probability distribution.

In the section "Computing the PNL," we illustrate the PNL metric computation for a particular system. Although the metric and the process described above to compute its value for a given system are general, the approach used to solve the model and the actual computation of the metric may vary from model to model, and hence from system to system. Specifically, even though we were able to analytically solve the model used in our case study, for some systems, including those that yield more complex models, this may not be possible, and the metric may have to be computed using simulation or numerical methods.

Deriving the operational distribution

In this section we describe a production system, the data that were collected during production, and the method used to determine the operational distribution. A detailed description of this testing approach is included in Reference 2.

The system being tested for our case study is a customer care database with a large customer base. The system has a Web browser front end that initializes with a home page for customer care agents. Pages are served by an off-the-shelf Web server that caches requests. The customer care repository contains information that can be displayed dynamically to customers, as well as to agents. In this way, customers can obtain help using a self-service Web interface, thereby reducing the number of customer care agents needed.

A key part of the application is a search feature that provides the capability for a customer or agent to search the customer care repository based on specified keywords. The cost of customer care provided by agents taking phone calls is very high. Allowing the customer direct access to the application reduces, and, in many cases, eliminates the need for the customer to speak with an agent.

The application runs on four servers, with two servers dedicated to transactions by agents, and the other two supporting customers. The analysis of the types of requests submitted by users revealed distributions for agent-facing and customer-facing servers as shown in Table 1. The four primary types of requests include: static pages, search forms, search results, and error code pages. This last type of page represents application codes that customers are trying to interpret. A primary reason that customers access this software or contact agents is that they have received such a code and are trying to decode its meaning. These error codes should not be confused with browser codes such as 404. Note that the operational distribution data for the case study was derived by collecting field data over a two-and-one-half-month period.

In the workload characterization effort we focused on understanding the traffic scale, the traffic load bal-

ancing between the servers, and the characterization of the low activity periods and high activity periods. We also determined the cache hit ratio for the servers, and analyzed page requests at 15-minute intervals. We note that each Web server has a cache associated with it.

We found that generating static pages usually requires fewer resources than generating search result pages because a search result page generally involves multiple disk requests. For that reason, the requests on the customer-facing servers are often more resource-intensive than agent-facing ones because of the larger fraction of search pages typically requested by customers. Agents were more than twice as likely to request a static page than customers, whereas customers were more than four times as likely to use a search form than agents.

Throughout this paper, we refer to data collected about the Web server cache. The cache hit probability was computed as the average cache hit ratio, weighted by the frequency of each request type. The range of the cache hit probability was stable from day to day, ranging between 80 and 87 percent.

Performance testing

The workload characterization effort described in the previous section was used to design tests that reflect actual production usage. In addition, tests were also designed to cover the key parameters that impact the system capacity of the customer care platform, including single transaction tests, multiple transaction tests to reflect customer and agent workloads, tests using different database sizes, and tests using varying numbers of Web server processes.

In order to avoid overloading our internal intranet, the tests were run on it after hours, in a dedicated performance testing lab. The dedicated test environment was designed to guarantee that there would be no overlap between our tests and other usage. We had exclusive access to the servers and routers between the test drivers and the system under test. The application was installed on dedicated performance testing servers that reproduced the production environment as closely as possible. The primary difference between the production and test environments was that in the test environment, the network delays were local area network delays as opposed to wide area network delays experienced by customers and agents that access our production systems over the Internet.

We started our performance testing effort by running single transaction tests of types static, error code, search form, and search result. We then executed tests using workloads that represented the customer and agent transaction mixes determined in the field and described in the previous section. We concluded by investigating the impact of database size and multiple HTTP (HyperText Transfer Protocol) server processes on the performance. Throughout, we had to determine how to exercise the Web server cache. We controlled the caching explicitly, and each one of the tests in this section indicates whether or not the caching mechanism was turned on. In addition, we also simulated the measured cache hit probability of 85 percent by preparing a set of tests that would drive the process so that it reflected this probability.

The performance testing machine was instrumented with the SUN SE Toolkit.[8] We found that for single transaction tests, with the caching mechanism turned off and a database size of 200 MB, both *error code* and *search result* pages experienced a steep increase in response time far earlier than *search form* and *static page* types. These differences for different request types were not surprising because, as mentioned above, error code and search result pages are more resource-intensive than static pages and search forms.

When we considered the CPU time spent for each transaction type, we also found very noticeable differences in response times for the different types of page requests, when the caching mechanism was turned off. In this case, only the error code pages showed a precipitous rise in cost, brought about by bottlenecked database semaphores. The other three types of page requests showed much more modest increases as the load increased.

Similar tests were performed for single transactions when the caching mechanism was turned on. For these sets of tests, all requests were served from the cache. These tests were designed to distinguish between the cost expended to perform database access and the rendering of pages. The central observation was that, as expected, caching provided a significant speedup over accessing all pages from the database. In particular, we observed that the response times when the cache was turned on showed a 50-fold speedup as compared to when the cache was turned off. In addition, the caching mechanism provided for a significant extension of the workload that could be handled. Even when the request rate was increased

to roughly five times the load used when the caching mechanism was turned off, there was no problem handling that load. Therefore, in this architecture, some scalability could be achieved if we could attain the goal of all requests being served from the cache.

The high variability in system response times is the likely source of any customer perception of poor performance.

This translates to a 100 percent cache hit probability. For our customer care system, for which an 85 percent average cache hit probability was observed, we saw a very high variability in response times. We conclude that the variability in performance of the cached versus noncached transactions is the likely source of any customer perception of poor performance.

Measurements on semaphore spins reported by the SE Toolkit indicate that transactions were consuming CPU time while waiting for service. This conclusion was further substantiated by measurements of CPU usage for transactions. We found that the CPU time consumed by transactions was sensitive to load. This was also observed for the single transaction tests. We observed that the CPU usage per request for the agent workload was significantly higher than the CPU usage per request for the customer workload.

We observed increases in the response time as the database size was increased from 200 MB to 400 MB and then to 600 MB in the customer workload test. In particular, we observed an increase in response time of 25 percent when the size of the database was increased from 200 MB to 400 MB with a fixed request rate. When the size of the database was further increased from 400 MB to 600 MB, the increase in response time was 33 percent at the same request rate.

Our analysis also allowed us to detect an instability point. The decrease in response time as the request rate was increased was caused by a severe slowdown of the system which led to a decrease in the throughput because of a substantial number of aborted transactions. In particular, for a 200MB database, 37 percent of the transactions aborted at a certain load, whereas 85 percent aborted at a somewhat higher load. For a 400MB database, the abort rates were 40 and 49 percent for these loads. When the database size was increased to 600 MB, the rates rose to 80 and 86 percent, respectively.

We used the same databases for the agent tests as we did for the customer tests. Here we observed even larger increases in response times as the size of the database was increased. By far the largest increase in response time occurred when the size of the database was increased from 400 MB to 600 MB, namely 354 percent.

In addition, the steep increase for the 600MB database began earlier for the agent database than it did for the customer databases and the 200MB and 400MB agent databases. These different increases can be attributed to the different amounts of CPU time per request used by the agent and customer workloads when the caching mechanism was turned off. In the agent workload case, an increase in the kernel semaphore holding time, caused by the larger database size, created an unstable queue for semaphores above a certain load. We also saw a severe system slowdown for the agent workload, which led to aborted transactions and reduction of response time above this load.

For the multitransaction tests with the cache turned on, we assumed that the cache hit probability was 100 percent. In practice, the cache hit probability will be significantly less than 100 percent. For this system, we found an average cache hit probability of approximately 85 percent. Therefore, to obtain an accurate estimation of the production working range, a test was designed to simulate a cache hit probability of 85 percent. The load that could be handled was significantly higher than that handled with the cache tuned off, but not as high as when all requests were retrieved from the cache.

The SE Toolkit identified bottleneck states for both the *cache on* and *cache off* test cases. In the *cache off* case, there was kernel contention observed for database semaphores. In the *cache on* case, a CPU run queue and a network bottleneck were revealed above a certain request rate. These bottlenecks were caused by limitations in the networking software, such as the maximum number of connections allowed to be set up. Since the cache on bottlenecks occurred at much higher request rates than we ever expect to see in practice, we expect that they can be safely ignored for the foreseeable future. However, since we are concerned about workload scalability, knowledge

of such potential bottlenecks at greatly increased loads is very valuable information that can be used for proper planning.

To alleviate a mutual exclusion kernel bottleneck, we investigated the performance impact of using multiple HTTP servers, varying the number of HTTP servers from one to six. We used customer workload tests designed to generate a simulated cache hit rate of 85 percent for the single HTTP server case, run on the 200MB database, with multiple HTTP server configurations.

By adding HTTP servers, we found two conflicting situations. On the one hand, by having n-HTTP servers, we created n queues, each of which could be executed independently on an n-CPU server. Since each queue maintains its own lock, lock contention was significantly reduced. This led to an ability to handle a much heavier workload with a minimal increase in response time. On the other hand, however, the cache hit probability was reduced when multiple HTTP servers were run, because the caching mechanism is maintained inside the HTTP server. This effect was seen by an increase in response time as the number of HTTP servers was increased. What we observed was that the cache hit probability dominated the response time at low loads, but the queuing for the database lock caused the curves to cross over as the number of requests per unit time continued to increase. Eventually, the reduced caching probability was the main factor in determining the response time. Therefore, the 2-HTTP server configuration provided the best performance results for both low loads and high loads.

Computing the PNL

In this section, we describe how we computed the PNL metric for the customer care system. We first determined our performance objective, defining T_{max} to be the maximum acceptable response time. This allowed us to precisely characterize poor performance. We defined the arrival process for the customer care system to be Poisson because that is the traditional way of modeling a system with a large population of users, and we have found in the past that we have gotten meaningful results when we assumed this type of distribution for similar systems.

To define the service time distribution, we noted that for this dedicated database all access is serialized. Furthermore, users experience a degradation in performance only if an arriving transaction at the database finds at least $N = \lceil T_{max}/x \rceil$ waiting transactions, where x is the constant service time provided at the Web server when the caching mechanism is turned off. This statement is true because requests served from the cache always meet the performance objective.

Since our performance tests indicate that the database service time distribution can be accurately approximated by a deterministic process, we modeled our customer care service provider database as an M/D/1 queue. This means that we assumed Poisson arrivals and deterministic service times. Because the measured cache hit probability was 85 percent, only 15 percent of the total requests were served by the M/D/1 queue. Therefore the state that will be used to compute the PNL metric is:

S = number of customers found in service at the M/D/1 queue

In general, the model we obtain should be solved using either a simulation tool or analytical methods. For this system we used M/D/1 queue tables to obtain the model solution. To compute the PNL metric, we need to determine the probability that users would experience poor performance, i.e., have a response time exceeding T_{max}.

Finally, the PNL metric was computed by weighting the probability of finding $N = \lceil T_{max}/x \rceil$ customers in service at the M/D/1 queue, by the probability that the request was not found in the cache.

The probability of finding N or more customers in service on an M/D/1 queue using a first-come, first-served queuing discipline was tabulated in Reference 9 and is shown in Table 2, for N equal to 1, 2, 3, and 5. Plots of the PNL value for the system with cache hit probabilities of 85 and 95 percent are shown in Figures 1 and 2. These plots were used to assess the probability that the users would experience poor performance as the workload scales.

Figures 1 and 2 show how queuing at the Web server impacts the value of the PNL metric, for a given cache hit probability. For example, if the cache hit probability is 85 percent, we would use Figure 1 and draw a line representing the maximum acceptable PNL parallel to the "offered load" axis. We would then compute $N = \lceil T_{max}/x \rceil$. If the PNL curve for the computed N lies within the rectangle defined by the axes, the maximum acceptable PNL value, and the max-

Figure 1 PNL for cache hit probability 85 percent

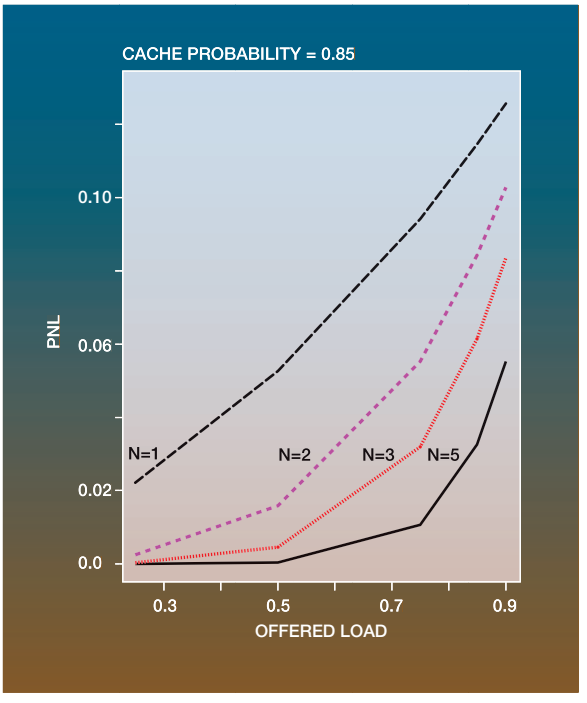


Table 2 M/D/1 system, probability of a transaction waiting longer than N service times

Offered Traffic	$N = 5$	$N = 3$	$N = 2$	$N = 1$
0.25	0.000016	0.001704	0.01685	0.147924
0.50	0.002476	0.030507	0.106083	0.351283
0.75	0.070931	0.213147	0.368701	0.627677
0.85	0.217355	0.409456	0.561451	0.763608
0.90	0.368163	0.557102	0.684924	0.837845

imum expected load, then the system would be scalable. Otherwise, performance problems can be expected when the indicated loads are reached. This would imply that additional capacity would have to be added before those load levels are reached in order to maintain acceptable performance.

Applying this approach to the customer care system, we selected the 85 percent cache hit probability shown in Figure 1, because that was consistent with the measured cache hit probability observed in the field. Our goal was to meet the PNL objective of no more than 0.02 for loads that we determined to be as high as we might encounter in the foreseeable future. In this case we selected a load of 0.5 units. We see that the system will meet the requirement for $N = 5$, $N = 3$, and $N = 2$, for loads in this range. If the load was increased to 0.6 units, then there would likely be problems for $N = 1$ and $N = 2$, but not for $N = 3$ or $N = 5$. In contrast, if the workload were to be increased to 0.8 units, then we see from Figure 1 that there would likely be problems for $N = 1$, $N = 2$, and $N = 3$.

For comparison, we again consider loads of 0.5, 0.6, and 0.8 units with a maximum PNL of 0.02, this time with a cache hit probability of 95 percent. The relevant information is shown in Figure 2. We see that with the higher cache hit probability and a workload that does not exceed 0.5 units, any of the considered values of N would be unlikely to encounter problems. When the workload was increased to 0.6 units, the system would likely meet the requirements for all values of N except $N = 1$. For a load of 0.8 units, there would likely be problems when $N = 1$ and $N = 2$.

Conclusions

In this paper we presented a new metric, PNL, used to predict whether a software system is likely to provide satisfactory performance under heavier workloads. A case study involving the performance testing for a large industrial production system is included, and the computation of the PNL metric for this system is described in detail.

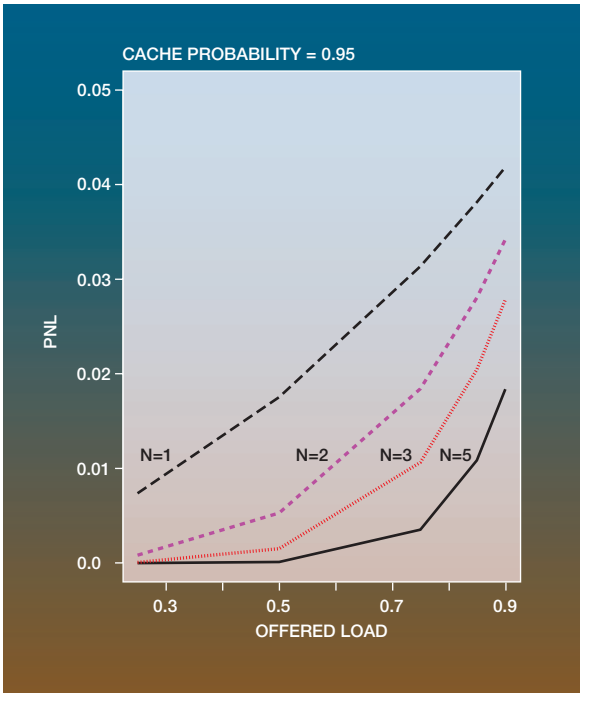
Our performance testing method relied on production workload characterization. First, we collected measurements from the production environment in order to characterize the system workload and to capture the system performance. We classified page requests by type and estimated the cache hit ratio for each request type. Next, we put in place a detailed performance testing plan to obtain response time curves and CPU costs per transaction for each request type, under various load levels. A primary conclusion of our analysis was that the main performance discriminators for this system were the cache hit probability and the serialized access to the database server. We determined that an observed performance bottleneck could be relieved by increasing the number of Web servers and by reducing the time required to obtain pages from the database. This led to a series of recommendations for the project that were likely to alleviate the bottleneck by allowing parallel access to databases.

We then applied the PNL metric using data collected from the customer care service provider. The results were used to communicate to management the likelihood of problems caused by future workload growth.

As expected, the cost of deriving the operational distribution was modest, whereas the payoff was very high. In particular, it would have been likely that the system would not have been able to handle the required workload once its predicted customer base was achieved. Generally, we recommend that projects collect daily reports on traffic and performance measurements, and benchmark performance during lab testing for new releases and proposed system changes. This will be useful for developing an operational distribution for performance testing.

We expect that our PNL metric will be applicable to a wide range of systems including backend systems that support a large number of TCP/IP (Transition Control Protocol/Internet Protocol) connections, HTTP and HTTPS (HTTP over Secure Sockets Layer) servers, and database engines. Of course, more complicated models may be required, depending on the resources being modeled and the protocols used for access. A key step to enable applicability is the bottleneck identification process, which is performed to identify the key resources that constrain scalability. This allows us to proceed to the modeling step required to compute the PNL metric. It is our experience that mature software development organizations that develop large industrial software invest time and resources to understand how their software will be used and to locate the performance bottlenecks. Therefore, we believe that our PNL metric will be widely applicable to large software systems that have stringent scalability requirements.

Figure 2 PNL for cache hit probability 95 percent



Cited references

1. J. D. Musa, "Operational Profiles in Software Reliability Engineering," *IEEE Software* **10,** No. 2, 14–32 (1993).
2. A. Avritzer and E. J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Transactions on Software Engineering* **21,** No. 9, 705–716 (1995).
3. S. S. Gokhale and K. S. Trivedi, "Structure-Based Software Reliability Prediction," *Proceedings of Advanced Computing (ADCOMP)'97*, Chennai, India (1997) pp. 1–7.
4. J. A. Whittaker and M. G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering* **SE-20,** No. 10, 812–824 (1994).
5. S. N. Weiss and E. Weyuker, "An Extended Domain-Based Model of Software Reliability," *IEEE Transactions on Software Engineering* **SE-14,** No. 10, 1512–1524 (1988).
6. J. F. Meyer, "Performability Evaluation: Where It Is and What Lies Ahead," *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, Erlangen, Germany IEEE, New York (1995) pp. 334–343.
7. *Performability Modeling: Techniques and Tools*, B. R. Haverkort, R. Marie, G. Rubino, and K. S. Trivedi, Editors, John Wiley & Sons, Inc., New York (2001).
8. A. Cockcroft and R. Pettit, *Sun Performance and Tuning, Second Edition*, Sun Microsystems (1998).
9. P. Kuhn, *Tables on Delay Systems*, Institute of Systems and Data Technics, University of Stuttgart, Germany (1976).

Accepted for publication October 16, 2001.

Elaine Weyuker *AT&T Research Laboratories, 180 Park Avenue, Florham Park, New Jersey 07932 (electronic mail: weyuker@research.att.com).* Dr. Weyuker received a Ph.D. in computer science from Rutgers University, and an M.S.E. in electrical engineering from the University of Pennsylvania. She is currently a technology leader and an AT&T Fellow at AT&T in Florham Park, NJ. Before joining AT&T in 1993, she was a professor of computer science at the Courant Institute of Mathematical Sciences of New York University, where she had been on the faculty since 1977. Prior to that she was a faculty member at the City University of New York, a systems engineer at IBM, and a programmer at Texaco, Inc. Dr. Weyuker is a Fellow of the ACM and a senior member of the IEEE. She was awarded the YWCA Woman Achiever award in 2001. She is a member of the Board of Directors of the Computing Research Association, and on the Technical Advisory Board of Cigital, Inc.

She is a member of the editorial boards of *ACM Transactions on Software Engineering and Methodology* and the *Empirical Software Engineering Journal*, and she is an advisory editor of the *Journal of Systems and Software*. She has been the secretary/treasurer of ACM SIGSOFT and an ACM national lecturer. Her research interests are in software engineering—particularly software testing and reliability—and software metrics. She has published more than 100 refereed papers in journals and conference proceedings in those areas. She is also interested in the theory of computation and is the author (with Martin Davis and Ron Sigal) of *Computability, Complexity, and Languages*, published by Academic Press.

Alberto Avritzer *AT&T Laboratories NDPA, 200 Laurel Avenue, Middletown, New Jersey 07748 (electronic mail: avritzer@att.com).*
Dr. Avritzer received a Ph.D. in computer science from the University of California, Los Angeles, an M.Sc. in computer science from the Federal University of Minas Gerais, Brazil, and a B.Sc. in computer engineering from the Technion, Israel Institute of Technology. He is currently a technology consultant at AT&T Laboratories in Middletown, NJ. Dr. Avritzer spent the summer of 1987 at the IBM Research Center in Yorktown Heights, NY. His research interests are in software engineering, particularly in software testing and reliability, real-time systems, and performance modeling, and he has published several papers in those areas. Dr. Avritzer is a member of ACM SIGSOFT, and IEEE.