Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study

Elaine J. Weyuker, *Senior Member*, *IEEE*, and Filippos I. Vokolos

Abstract—An approach to software performance testing is discussed. A case study describing the experience of using this approach for testing the performance of a system used as a gateway in a large industrial client/server transaction processing application is presented.

Index Terms—Software performance testing, program testing, software testing, performance testing.

1 Introduction

Although there has been very little work published describing approaches to software performance testing, it is nonetheless an extremely significant issue for many large industrial projects. Often, the primary problems that projects report after field release are not system crashes or incorrect system responses, but rather system performance degradation or problems handling required system throughput. When queried, it is not uncommon to learn that, although the software system has gone through extensive functionality testing, it was never really tested to assess its expected performance.

Recently, we have had experience examining a substantial amount of data collected during architecture reviews. Our goal was to develop metrics to predict the likely success or failure of a project [4]. At AT&T, architecture reviews are conducted very early in the software development lifecycle, after the system requirements are complete, but before low-level design has been done. The goal of an architecture review is to assure that the architecture is complete and to help identify likely problems that can be easily addressed at this stage, but would be much more expensive to correct once implementation is in progress. During an architecture review, problems are identified that are considered likely to negatively impact the software system that is ultimately produced and categorized both according to the cause of the problem and the severity.

In this process, we have found that performance issues account for one of the three major fault categories. Performance problems identified at this stage might include such things as the lack of performance estimates, the failure to have proposed plans for data collection, or the lack of a performance budget. Our experience shows that a lack of sufficient planning for performance issues is often a precursor of performance problems once the software is released to the field.

During architecture reviews, in addition to categorizing problems by type, they are also assigned a severity class. The most severe type of problem is said to be project-affecting. These are problems that need to be addressed immediately in order to prevent the project from being put in jeopardy. There are three other severity classes, which we call critical, major, and minor. As the severity of a problem decreases, the timeframe in which it must be addressed gets longer. A critical problem must be solved before the architecture can be validated. Major issues can have an impact on user satisfaction. Minor issues are those that might affect later releases. While examining data collected during more than 30 architecture reviews, for systems containing many millions of lines of high-level code, we found evidence of a Pareto-type distribution.[1]

This type of very uneven distribution of resource usage has been observed for many different computer-related properties. For example, it is often observed that 20 percent of the system features account for 80 percent of the system usage. For the study that we recently performed to predict project risk based on architecture review findings, we found that, for project-affecting issues, there was a 70-30 rule evident. In particular, each of the projects was independently assessed as being at high risk, moderate risk, or low risk of failure. Seventy percent of the projects were considered to be in excellent shape and therefore at low risk of failure, while 30 percent of the projects were considered to be at least somewhat problematic and classified as having either moderate or high risk of failure. When we looked at the distribution of project-affecting issues, we found that 70 percent of the most severe class of

- *E.J. Weyuker is with AT&T Labs-Research, Room E237, 180 Park Ave., Florham Park, NJ 07932. E-mail: weyuker@research.att.com.*
- *F.I. Vokolos is with Lucent Technologies, Room 2K-041, 480 Red Hill Rd., Middletown, NJ 07748. E-mail: filip@lucent.com.*

Manuscript received 16 Apr. 1999; revised 5 Oct. 1999; accepted 15 Mar. 2000.
Recommended for acceptance by A. Cheng, P. Clements, and M. Woodside.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111493.

1. The Pareto principle was named for an Italian economist, Vilfredo Pareto, who observed in the late 19th century that, in many different countries which he studied, there was a very uneven distribution of wealth. In particular, he found that it was common for 80 percent of the wealth to be owned by 20 percent of the population.

problems resided in this weakest 30 percent of the projects. These assessments were made at the time of the architecture reviews.

When we focused our attention only on performance-related project-affecting issues, however, we found that the distribution of these types of problems was even more skewed than for all classes of project-affecting problems. Instead of seeing a 70-30 split, we found that 93 percent of the performance-related project-affecting issues were concentrated in the 30 percent of systems that were deemed to be most problematic. This confirms our experience that projects that are in good shape are very likely to recognize that addressing performance issues are a necessary part of the architecture phase of development, while those projects that are in the most trouble almost never have what they often perceive to be the "luxury" of considering performance issues. But, as we mentioned above, performance problems account for many of the showstopping faults encountered once the projects are released to the field and, so, developing comprehensive performance testing strategies is essential.

The issues that have to be addressed when doing performance testing differ in a number of ways from the issues that must be addressed when doing typical functionality testing. In this paper, we will examine performance testing issues and describe an approach to address some of these issues for certain classes of software systems. This work is based on experience gathered while testing the performance of different industrial software systems. We will clarify the objectives of performance testing and describe a technique for creating appropriate workloads needed for the testing. We will look at the role of requirements and specifications in successful performance testing. We will also present a case study describing the results of some recent work that we have done to test the performance of an existing gateway system that serves as the middle layer of a three-tiered client/server transaction processing application. The approach used drew on an earlier technique that we developed for performance testing, as described in [2], but dealt with the fact that, unlike the earlier project, this project did not have detailed historical usage data available.

2 Previous Work

There has been very little research published in the area of software performance testing. A search of the published literature using the on-line version of Science Citation Index, for example, returned only two papers: an earlier version of this paper [11] and one by Avritzer and Weyuker [2]. Similarly, using the ACM on-line search engine to search all available ACM journals and conference proceedings uncovered only [11], while the search facility for both the IEEE and IEE journals and conference papers uncovered no papers on the subject.

There *is* substantial literature describing work on software performance modeling [6], [8], [10] and software (functionality) testing [5], [7], but, although related, this work does not directly consider how to select or generate test suites for testing the performance of software, which is the primary goal of this paper. A search of the internet for "software performance testing" uncovered relatively few businesses that offered some sort of tool to aid people doing software performance testing or workshops or seminars that included a section on the subject, but the search uncovered *no* research papers. It is interesting to note that one of the websites, [1], had the following statement on their software performance testing webpage: "Performance testing software is not something many companies do much of, unfortunately." Thus, it appears that there is neither an existing body of research literature nor an extensive collection of practical or experiential information available to help testers faced with the prospect of doing software performance testing.

One of the few research papers on the topic [2] does present a software performance testing approach. The goal in that work was to compare the performance of an existing production platform and a proposed replacement platform to assess whether the new platform would be adequate to handle the required workload. The traditional approach to such a comparison is to develop software for the proposed platform, build the new architecture, and collect performance measurements on both the existing system in production and the new system in the development environment.

In contrast, Avritzer and Weyuker [2] introduced a new way to design an application-independent workload for doing such a performance evaluation. It was determined that, for the given project, the quantity of software and system availability requirements made it impossible to port the system to the new platform in order to do the performance testing. Therefore, a novel approach was designed in which no software was ported at all. Instead, a synthetic workload was devised by tuning commercially available benchmarks to approximate resource usage of the actual system.

Motivating this work was the insight that, given that the project had systematically collected detailed system usage data on the old platform, this information represented a major asset that could be exploited for doing performance testing on the new platform. In particular, it was not necessary to port the software and run the actual system on the new platform, only to run software that behaved the way the system did from the point of view of resource usage. It was recognized that it was totally irrelevant what the software was actually doing, provided that it used resources in ways similar to the way the software to be ported used them. The ultimate result of doing performance testing of the software using this approach was a decision not to port the system to the proposed platform because the testing indicated that, although the new platform would be adequate to handle average workloads, it would not be able to handle peak loads that the system was likely to encounter.

This technique proved to be a very efficient way to arrive at this decision and saved the project significant resources, both in terms of personnel costs for porting the software that would ultimately have had to be back-ported to the existing platform, plus an enormous savings made by not purchasing what this testing approach determined would have been inappropriate (and very expensive) hardware for the system.

In the next sections, we will discuss a related approach to software performance testing that we have found useful when testing the performance of an existing system to be redeployed on a new platform. In this case, the commitment to purchase the new equipment had already been made and work was underway toward rewriting and porting the software to the new platform. Nevertheless, even though we could not affect the decision as to what hardware to purchase, our goal in doing this performance testing was to determine whether or not there were likely to be performance problems encountered on the new platform and to prepare for them as best we could in order to minimize service disruptions. We next outline performance testing objectives. In Section 6, we outline our approach to testing and present a case study describing our experience doing performance testing for this system.

3 Performance Testing Objectives

In this paper, when we speak of software performance testing, we will mean all of the activities involved in the evaluation of how the system can be expected to perform in the field. This is considered from a user's perspective and is typically assessed in terms of throughput, stimulus-response time, or some combination of the two.

An important issue to consider when doing performance testing is scalability: the ability of the system to handle significantly heavier workloads than are currently required. This necessity might be due to such things as an increase in the customer base or an increase in the system's functionality. Either of these two changes would typically cause the system to have to be able to provide a significantly increased throughput. If there has not been appropriate testing to assure that the system can be scaled, unacceptable levels of denial of service or unacceptable response times might occur as workloads increase. This is likely to have a very negative impact on customer satisfaction and therefore retention.

With these issues in mind, there are a number of different goals one could have when designing a research project related to performance testing. These include:

1. the design of test case selection or generation strategies specifically intended to test for performance criteria rather than functional correctness criteria,
2. the definition of metrics to assess the comprehensiveness of a performance test case selection algorithm relative to a given program,
3. the definition of metrics to compare the effectiveness of different performance testing strategies relative to a given program,
4. the definition of relations to compare the relative effectiveness of different performance testing strategies in general; this requires that we be able to say in some concrete way what it means for performance testing strategy X to be better than testing strategy Y,
5. the comparison of different hardware platforms or architectures for a given application.

In this paper, we focus on the first of these goals. We note here that, since there is almost a complete absence of any published research on software performance testing, there is a critical need for work in all of these areas.

There are also a number of things that need to be measured when evaluating a particular software system's performance. Included among these are resource usage, throughput, stimulus-response time, and queue lengths detailing the average or maximum number of tasks waiting to be serviced by selected resources. Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage. Other resources that might be of importance for specific projects include switch usage (for telecommunications projects) or database access rates.

Note that it is sometimes impossible to satisfy all requests for resources simultaneously. If the total requirements for disk space by different processes exceeds the space available, for example, this problem should be identified as early as possible in the requirements phase. In some cases, the system architecture will have to be redone. This can be a very expensive process with severe consequences. In other cases, processes will simply have to cut back on their resource demands, perhaps by developing better algorithms or by reducing functionality. Another alternative might be to order additional hardware to accommodate demands. In any case, the earlier there is an awareness of potential performance problems, the more likely it is that an acceptable solution can be developed and the more economically any necessary rearchitecture work can be done. Although performance planning and testing may be expensive and time consuming, we believe that it is nonetheless likely to be cost-effective for the reasons outlined above.

4 Creating Workloads for Performance Testing

One traditional goal of performance testing is to allow an organization to compare competing hardware platforms or software products under similar conditions that are considered to be of particular interest to the project. In that way, the project is able to select the best product for their needs as determined by a set of performance requirements. The usual way of doing this type of performance testing is to develop a benchmark (a workload or workloads that are representative of how the system will be used in the field) and then run the benchmarks on the systems, comparing the results. It is tacitly assumed that the system's behavior on the benchmarks will be indicative of how the system will behave when deployed because the benchmarks were specifically designed to be representative of field usage.

That is *not* the goal of the work described in this paper. We are not comparing potential platforms or software products. Firm decisions have been made about the hardware being purchased and the software has been designed and written explicitly for this new hardware platform. The goal of the work described here is to determine, as early as possible, whether there are likely to be performance problems once the hardware is delivered and the software is installed and running with the real customer base. Just as the software's functionality has to be

tested, so too does the software's performance since, from the customer's perspective, unacceptable response times, or insufficient resources can be just as problematic as incorrect or missing functionality.

Of course, the notion of a representative workload is itself problematic for several reasons. The first problem is the issue of where that data is to come from. How will we know what a representative workload actually is? This is often one of the easier issues for us to resolve. In our (telecommunications) environment, many projects routinely monitor system traffic, thereby providing the tester with the resources to develop a so-called operational profile that describes how the system has historically been used in the field and, therefore, is likely to be used in the future. An *operational profile* is a probability distribution describing the frequency with which selected important operations are exercised. This typically requires that domain experts make decisions regarding which operations are actually central for a given application. It also requires that someone make decisions about the window of observation from which the data will be drawn.

If no earlier version of this system is available, or historical usage data has not been collected for the system, there is frequently a similar system that could provide guidance in the design of workloads that can be used for performance testing. Alternatively, if there is a system whose output becomes the input to the system under consideration and this older system has been monitored, the required data may thereby be available to help in the design of an appropriate workload for performance testing. In the work we describe in this paper, detailed operational profile data was *not* available. That was one of the major challenges that we faced when doing this work.

A second issue that must be considered is whether the performance test cases should reflect an average workload or a very heavy or stress load. In either case, it is again necessary to consider the window of observation from which the average or stress loads will be taken. Will we look at the average traffic during a one hour window, during a 24-hour period, or during some larger period? Will we distinguish between times of light load and heavy load if this is predictable? Will we look at the heaviest load that occurred during a given 24-hour period or the heaviest load occurring during the month or during a year? These would almost surely not all be the same.

For many telecommunications systems, it is usual to consider the period of repeatability to be one week. Thus, except for special occasions, such as holidays or natural (or unnatural) disasters, the system usage on a given day of the week will look almost identical to the same day of the week during the following week. That is, the peaks and the valleys will generally occur at the same times from week to week. In [3], system traffic data was provided for a 24-hour weekday for five successive weeks. Examination of those graphs is consistent with this observation. The five graphs are very similar, having the same shape and same busy-hour each week.

Once the answers to these questions have been determined, the project must then determine whether or not the system for which performance testing is to be done will typically run in isolation on the hardware platform. In our environment, it is common for a project to be the sole application running in the environment. Therefore, provided that a representative workload has been selected, the observed throughput or response time is an accurate reflection of the behavior that the user can expect to see. However, if the software system is likely to be running with other unrelated systems on a common platform, it is necessary to assure that, during performance testing, the workload is run along with a representative background workload. Another way of thinking of this is to say that a performance test suite should include test cases that are a representative portrayal of the workload of the system under test, as well as a representative portrayal of the workload that is likely to be resident with the system when it is operational. The more complex the mix of jobs typically running on a system, the more difficult it is to accurately measure the expected performance. In the work described in this paper, we did not have to be concerned about this issue since the system would be running on a dedicated platform.

Still another issue to be considered is whether an average event arrival rate will be used or a more accurate probability distribution describing the variation in arrival rate times will be used. Although this probability distribution might have the same average value, it would likely represent a significantly more accurate picture of the system's behavior. However, it will be much more costly to determine and the mathematics involved in dealing with it might be a great deal more complex.

5 The Role of Requirements and Specifications in Performance Testing

In order to do software performance testing in a meaningful way, it is necessary to have performance requirements, provided in a concrete, verifiable manner. This should be explicitly included in a requirements or specification document and might be provided in terms of throughput or stimulus-response time and might also include system availability requirements. Frequently, unfortunately, no such requirements are provided, which means that there is no precise way of determining whether or not the performance is acceptable. This is one type of problem that we try to forestall by including performance issues in an architecture review and not permitting a project to go forward with development until the performance requirements have been made explicit.

Another crucial issue when doing performance testing is making sure that if there are stated performance-related requirements, they can actually be checked to establish whether or not they have been fulfilled. Just as it is not very useful to select inputs for which it is not possible to determine whether or not the resulting output is correct when doing correctness testing for a software system, when doing performance testing it is just as important to write requirements that can be validated. It is easy enough to write a performance requirement for a compiler such as: It must be possible to compile any module in less than one second. Although it might be possible to show that this

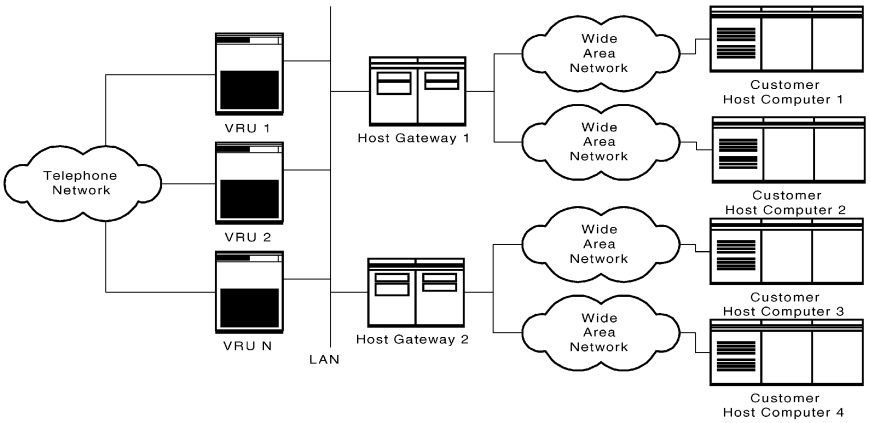

Fig. 1. High level system architecture.

requirement is not satisfied by identifying a module that takes longer than one second to compile, the fact that the compiler has been tested on many modules, all of which compiled correctly in less than one second, does not guarantee that the requirement has been satisfied. Thus, even plausible sounding performance requirements might be unverifiable.

A more satisfactory type of performance requirement would state that the system should have a CPU utilization rate that does not exceed 50 percent when the system is run with an average workload. Assuming that test cases have been created that accurately reflect the average workload, it is possible to test whether or not this requirement has been satisfied. Another example of a verifiable performance requirement might be that the CPU utilization rate must not exceed 90 percent even when the system is being run with a stress load equal to the heaviest load ever encountered during any monitored period. This would require that the appropriate stress workload be determined and run and a check be made of the CPU utilization rate under that workload. Similarly, it is possible to test for things like maximum queue length or total throughput required under specified loads, or the time required for disk input/output when the disk has certain characteristics.

Since performance requirements must be included for average system loads and peak loads, it is important to specify those loads as early as possible, preferably in the requirements document. As discussed above, detailed operational profile data is sometimes available, given that system traffic data has been collected and that analysis of the recorded data has been performed. This is frequently a significant task, requiring a great deal of resources, that needs to be planned for and budgeted for, but one that is extremely important both for functionality testing and performance testing.

For the system used in the case study described in the next section, detailed traffic data was not available. This meant that estimates had to be made by the most knowledgeable team members. If a business case has been made for a system, or even for the addition of new features, then that might include the necessary estimates. Of course, as additional information is acquired, a refinement of these estimates should be made.

6 A Case Study

In this section, we describe how we modified the performance testing approach described in [2] to test a gateway system that is the middle layer of a 3-tiered client/server transaction processing application, even though we did not have precise historical usage data.

6.1 System Description Information

The system accepts input from a caller and returns information that resides on a host server. The input may be in the form of dual tone multifrequency signals (touch tones) or limited vocabulary speech. Output is in the form of prerecorded digitized or computer-generated speech.

The client, or first tier of this architecture, consists of Voice Response Units (VRUs). These are computer systems that interface with the caller. The server, or third tier, consists of mainframe host computers that contain the information requested by end-users. The middle layer or second tier consists of the gateways: computer systems that allow the clients to interface with the servers. Servers generally reside at remote locations. From an architectural standpoint, the purpose of the gateway is to allow concentration of network connections and to off-load some of the processing that would otherwise have to be performed by each VRU.

Typically, a gateway will provide service to many different VRUs, while each VRU is able to access the services provided by many different gateways. These gateways and the VRUs share a LAN to permit this communication. Similarly, each gateway machine can connect to multiple remote servers. In order to allow this communication with a variety of host servers, each gateway is designed to support three different network protocols: SNA/3270, TN3270, and TCP/IP. The system architecture is shown in Fig. 1. We produced this high-level representation of the system architecture based on discussions with a senior person who was very familiar with the project. At this point, we made the decision that, from the point of view of software performance testing, this representation of the system architecture provided sufficient levels of detail.

An Application Script running on the VRU interacts with the end-user. When the application needs to perform a host transaction, an interprocess communications (IPC)

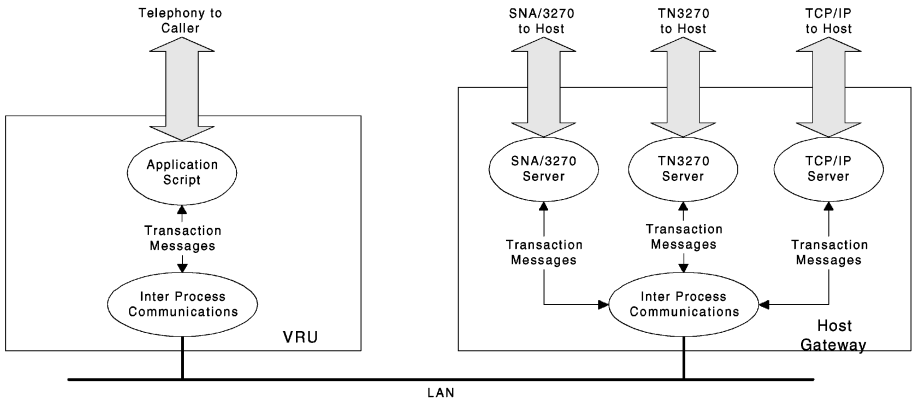

Fig. 2. High level software architecture.

message is sent to the appropriate server process running on the gateway. For our purposes, a transaction is defined as a request sent to the host followed by the reply received from the host. The server process on the gateway formats the request according to the host requirements and sends it to the host. When a reply is received, the process on the gateway parses the returned host data, formats an IPC message, and sends the reply to the application on the VRU. The high level software architecture is shown in Fig. 2. We also prepared this architecture diagram based on discussions with members of the project team. Since we were primarily interested in determining system input parameters that could affect the performance, we viewed the system at a relatively coarse level of granularity, identifying both system-level parameters and interprocess communications.

The existing gateway systems are based on PC hardware (Intel Pentium processors, ISA bus) running the Unix operating system. In order to improve the performance and reliability and to reduce maintenance costs, it was decided to upgrade the hardware platform with mid-range systems that use multiple PA RISC processors, have large amounts of RAM, and are capable of supporting many physical I/O interfaces.

6.2 Performance Testing Goals

Although there has been a software system running on the old platform that has been performing most of the tasks to be performed by the new software system on the new platform, detailed performance data had not been collected. Therefore, we were forced to use the only available information regarding the new system's performance requirements, which came from an analytical study done by the project team after the new platform was purchased, but before it was available.

This study estimated the expected number of transactions that the new platform should be able to process when it was connected to host systems using the SNA/3270 and TN3270 protocols. It determined that 56 links, each capable of handling 56Kbps, were needed and that the system would have to be able to function with 80 percent of the links active, each handling transactions averaging 1,620 bytes. Although the vendor had given assurances that the new platform would be able to handle the required workload, they did not provide adequate data to substantiate these claims. What they did supply was performance data from the use of their hardware as database servers. However, a gateway system provides functionality that is qualitatively different than the functionality provided by a database server and, therefore, we felt that the performance data supplied by the vendor would not be indicative of the behavior that we could expect in our target environment.

The project team therefore concluded that the new platform contained new software components whose performance behavior was largely unknown and that the performance data supplied by the vendor was not directly applicable to a system used as a gateway. For these reasons, they decided to test the performance of the gateway prior to deployment using a configuration similar to the one that would be used in production.

An important first step was to identify the objectives of the performance testing activity. Initially, we thought that the objective should be to validate the requirements set by the analytical study, that is, to directly address the question of whether or not the gateway could effectively handle the expected number of transactions per second. After careful consideration, however, we decided that performance testing should help us answer the following more far-reaching questions relating to such issues as scalability, cost, and bottlenecks:

1. Does the new gateway have enough resources to support increased end-user volume within prespecified budgets of resource consumption and latency?
2. At what volume of end-users do these resources become inadequate?
3. What components of the hardware and software architecture of the gateway limit performance? Where are the bottlenecks?

6.3 Designing the Performance Test Cases without Detailed Historical Data

As we mentioned earlier, the project team decided that it would be best to do performance testing on a configuration as close as possible to the one to be used in production. To accomplish this, we had to borrow test lab facilities from the vendor since our own hardware was not yet available. Although this involved considerable expense both for the use of the facilities, as well as costs for our testing personnel

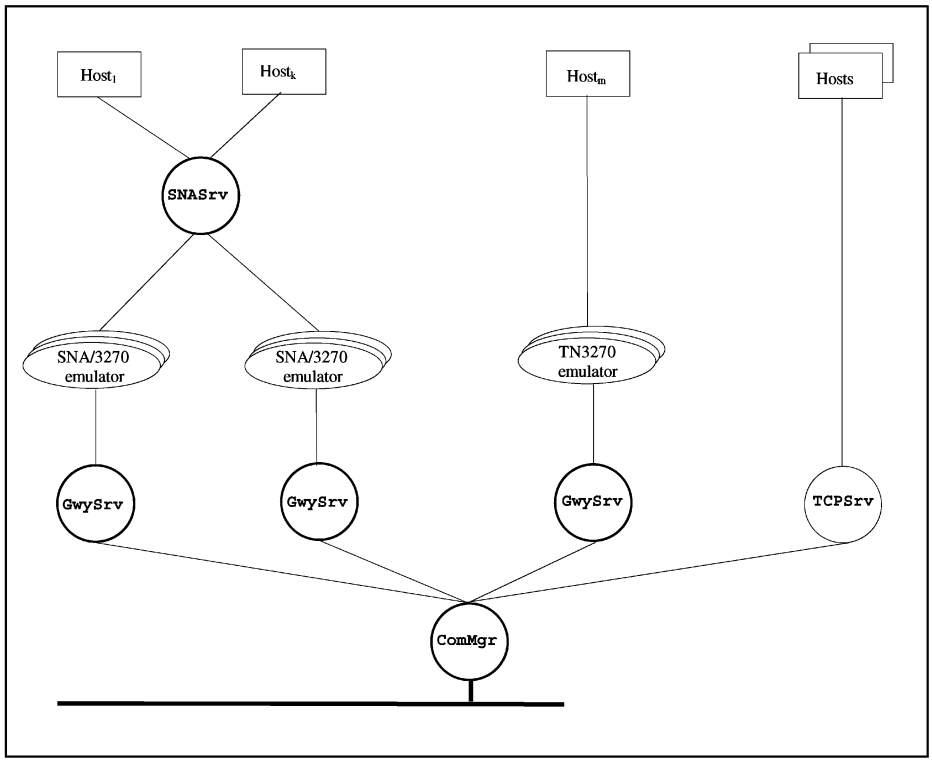

Fig. 3. Gateway software architecture.

for travel and living expenses, we knew it would be too risky to wait until our own platform was available. The limited availability of this lab and the need to minimize costs meant that we had to preplan all test activities with extreme care. We feared that there might be catastrophic performance problems when the new platform was deployed if we did not do comprehensive performance testing prior to its arrival and deployment.

Access to the vendor test lab facilities was strictly limited to one week. As a result, test cases had to be carefully planned prior to going to use the lab since we knew that there would only be enough time to run and analyze the results from a limited number of test cases while there and that experimentation during this testing phase would be impossible because of the time constraints. For this reason, we had to develop concrete test plans and test suites prior to visiting the vendor's facility.

It is essential to recognize that designing test suites for performance testing is fundamentally different from designing test suites for functionality testing. When doing functionality testing, the actual values of parameter settings is of utmost importance. For performance testing, in contrast, the primary concerns are the workloads and frequencies of inputs, with the particular values of inputs frequently being of little importance.

It was decided that our performance testing had to address the questions listed in the last section for both average and peak workloads. This meant that we had to determine what these loads were likely to be. Experience told us that a usage scenario should be defined in terms of those parameters that would most significantly affect the overall performance of the system. Therefore, our immediate priority was to identify such parameters. To do this, we first looked at the software architecture of the gateway and its relationships to both the VRUs and to the customer host systems. We felt that the software architecture of a system could provide important insights that could be useful when designing our performance test suite. This is shown at a high-level in Fig. 3.

The software performance testing approach described by Avritzer and Weyuker [2] relied on a substantial amount of detailed data collected during nine months of field usage of the system on the existing platform. That meant that they were able to determine precisely average workloads and peak loads encountered during this period and use this information as the basis for software performance test suite creation. The primary challenge they faced was the fact that they did not have a version of the software that could run on the new platform. Therefore, they had to decide which parameters were most relevant for use in modeling test workloads so that they could synthesize workloads that could be input to the hardware that would behave the way the ported software would behave with production workloads. In contrast, for the software performance testing described in this paper, we did have the new software that would be running on the new platform, but we did not have access to detailed historical usage data and had only limited access to the new hardware platform. We therefore had to make estimates to design appropriate workloads for testing.

By studying the software architecture of the gateway, shown in Fig. 3, we observed that its performance is closely related to the performance of three processes: SNASrv, GwySrv, and ComMgr. This decision relied on our experience working with other similar systems and discussions with senior members of the project development and test teams.

The first process, SNASrv, was a new software component that did not exist on the old platform. There is one such process on the gateway and its purpose is to communicate on one end with all 3270-based hosts and, on the other end, with the different SNA/3270 emulators. The second process, GwySrv, was a modified version of one found on the old platform. Modifications were made so that this process could work with the new process mentioned above. No additional functionality was incorporated. There is one of these processes for each link to a host and each of these processes may interact with up to 25 different emulators. Each emulator supports a maximum of 10 Logical Units (LUs), which are ports through which end users or programs communicate in the SNA architecture. The third process, ComMgr, interfaces directly with the VRU. There is one of these processes on the gateway and it was ported without any modifications from the old platform.

Since the SNASrv process deals with connections to customer host systems, as well as with processes that communicate with the end-user, and the GwySrv process also deals with those processes that communicate with the end-user, we reasoned that the number of simultaneous active connections to customer host systems and the number of simultaneous callers should be two of the parameters that would most significantly impact the software performance. Additional parameters that would noticeably affect the overall performance of the gateway were the number of transactions per call, transaction sizes, call holding times, call interarrival rates, and the traffic mixture (i.e., percent of SNA vs. TN3270 vs. TCP/IP).

To determine realistic values for these parameters, we collected and analyzed data from the current production gateways. We used tracing to record certain types of information for each transaction. These included such things as time-stamps, response time, and LU number. We did this for several days with six different applications at two different sites. We believed that, although this data was collected over a relatively short period of time rather than many months, as was the case for the earlier study described in [2], it would nonetheless be very valuable and provide us with an accurate picture of the usage levels that would be seen in the field. By using this information, as well as some estimated values from earlier field experience, we determined that:

- Average usage was typically about 40 LUs per link, with 24 simultaneous callers, three transactions per call, 2.5 minute call holding time, and a five second call interarrival rate on each channel. This corresponds to approximately one transaction every three seconds for each link.
- Heavy usage was determined to be 120 LUs per link, with 120 simultaneous callers, four transactions per call, 3.5 minute call holding time, and a five second call interarrival rate on each channel. This is equivalent to approximately 2.2 transactions per second for each link.

We were unable to obtain precise data regarding transaction sizes and the traffic mixture. For transaction sizes, the primary concrete information that we were able to obtain was that the minimum and maximum sizes were 100 and 1,920 bytes, respectively. As a first approximation, our solution was to run the test cases for both usage scenarios, assuming that each transaction was 1,100 bytes (100 bytes sent to the host and 1,000 bytes received from the host). This was the number assumed in the analytical study that had been performed and, since it was consistent with the team's experience, we considered it to be a reasonable assumption.

Alternatively, we could have assumed a nonuniform distribution of transaction sizes such as assuming that different segments of the transaction population would have different transaction sizes. For example, for the typical usage scenario, we could have assumed that 25 percent of the transactions were 100 bytes, 25 percent were 1,920 bytes, and the remaining 50 percent were equally distributed to represent transactions of 500 bytes, 1,000 bytes, and 1,500 bytes. For the heavy usage scenario, we could have assumed that 25 percent of the transactions were 1,000 bytes, 25 percent 1,500 bytes, and 50 percent 1,920 bytes. We decided that using the single transaction size of 1,100 bytes would minimize the number of test cases while still being sufficiently accurate for our purposes.

With respect to the traffic mixture, our primary concern was the SNA/3270 protocol since all of the traffic on the existing gateway used the SNA/3270 protocol. Support for the TN3270 protocol was new and the system engineers anticipated that, at least initially, only a very small number of customer host systems would be using that protocol. Furthermore, both the TN3270 and the SNA/3270 used some of the same processes, so we expected few, if any, performance surprises from TN3270 traffic. As for the TCP/IP protocol, its implementation does not require significant resources on the gateway and we already had data that indicated that the effect of the TCP/IP protocol on the overall performance of the gateway was negligible. Thus, we decided to focus our testing effort on the SNA/3270 protocol.

If we had felt that it was necessary to test the SNA/3270 and TN3270 protocols separately because there had been, or was expected to be, significant TN3270 traffic, we would have created different test cases by varying the distributions. This could be done by starting with an assumption that the SNA/3270 protocol represented 90 percent of the traffic, while the TN3270 protocol represented 10 percent. Then, by decreasing and increasing the distributions by 10 percent until SNA/3270 represented 10 percent of the traffic and TN3270 accounted for 90 percent of the traffic, we could have provided a range of scenarios. These nine tests, ((SNA/3270 = 90 percent, TN3270 = 10 percent), (SNA/3270 = 80 percent, TN3270 = 20 percent), ..., (SNA/3270 = 10 percent, TN3270 = 90 percent)), ordered in decreasing order of importance, would then be run with both the average and high usage scenarios and could be used to provide us with insights into the role that each protocol plays in the consumption of resources, as well as the overall impact on the performance of the gateway.

6.4 Prescription for Developing Performance Test Cases

We now summarize the approach outlined above. It is important to recognize that, since every system is different, this cannot be in the form of a precise recipe for preparing

test cases. However, we have described the architecture of the system we were building and indicated the level of granularity at which we identified relevant parameters. Typical steps to form performance test cases include:

- Identify the software processes that directly influence the overall performance of the system.
- For each process, determine the input parameters that will most significantly influence the performance of the system. It is important to limit the parameters to the essential ones so that the set of test cases selected will be of manageable size.
- Determine realistic values for these parameters by collecting and analyzing existing usage data. These values should reflect desired usage scenarios, including both average and heavy workloads. A determination must also be made of the window of observation that will be used.
- If there are parameters for which historical usage data are not available, then estimate reasonable values based on such things as the requirements used to develop the system or experience gathered by using an earlier version of the system or similar systems.
- If, for a given parameter, the estimated values form a range, then select representative values from within this range that are likely to reveal useful information about the performance behavior of the system. Each selected value should then form a separate test case.

7 Performance Testing Results

To implement the test cases representing these scenarios, the project team developed two programs to simulate callers. These programs send transaction request messages to the gateway server process and receive replies. Both programs simulate a number of simultaneous callers, performing transactions of a size specified on the command line. One program sends transaction requests as quickly as possible, i.e., as soon as the reply to a request is received, another request is sent out. The other program more closely simulates human users by accepting parameters for the call duration and number of transactions per call and using them to add "think time," thereby simulating realistic call arrival rates. For a host application, we used an existing simple application that had been previously used for functionality testing. Since all customer applications require similar processing cycles by the gateway, the choice of customer application was not important for our testing.

Testing was conducted on a hardware configuration in the vendor's testing lab, similar to the one to be used in the field. Testing revealed some surprising results that differed markedly from the vendor's assurances regarding the performance of the gateway. We learned that, for both the average and heavy usage scenarios, the gateway could handle only a small fraction of the transactions estimated to be required by the analytical study, even though we had been assured by the vendor that the hardware was capable of handling loads of this size. We identified three problems in the vendor-supplied software that caused very high CPU utilization and contributed to the poor performance of the gateway.

The first problem was related to the polling frequency of the gateway by the host. The second problem was related to the algorithm used by the gateway server process, GwySrv, to search the various LUs for the arrival of new data. The third problem was related to the inefficient implementation of some library modules used by the SNA server process. We were able to resolve the first two problems fairly quickly, which led to a substantial improvement in the performance of the gateway. The solution to the third problem is much more subtle and involves making modifications to the vendor's software that must be carefully designed. It is expected that, when completed, these modifications will further improve the performance of the gateway.

8 Conclusions

For many industrial software systems, performance requirements play a central role in determining the overall quality and usability of the system. Many project teams expend a great deal of resources testing the functionality of the system, but spend little or no time doing performance testing, even though performance problems often significantly impact the project's ultimate success or failure. The issues that have to be addressed when doing performance testing differ in a number of ways from the issues related to functionality testing. In this paper, we have presented many of these issues and discussed our experience dealing with them.

We have also described our experience doing performance testing for an existing gateway system that was being redeployed on a new platform. We firmly believed that doing performance testing before the system was released to the user community would be a wise investment of resources and our experience with this system bore this out. We were able to uncover and correct several software faults that substantially compromised the performance behavior of the system. We are convinced that if the system had been deployed without having identified and corrected these problems, the system would have provided performance that would have been viewed by our customers as being entirely unacceptable. In addition, we identified three platform-related problems mentioned in the previous section that the vendor helped us address. Again, had they not been uncovered prior to system deployment, there would likely have been disastrous consequences.

It was essential to clearly identify the objectives of performance testing before beginning the actual test case selection. This allowed testers to identify unrealistic or untestable goals. Although this project had begun with stated performance objectives, these objectives had to be refined. This was achieved through repeated discussions with the project team. The result of this effort was a system that provided the resources required by its customers in a satisfactory way.

We used the software architecture of the system whose performance we wanted to evaluate as the basis for identifying the parameters that affected the performance most directly. This allowed us, for example, to save

significant time by focusing our testing effort on the SNA/3270 traffic. We concluded that this was acceptable because, as part of our performance testing preliminary work, we had determined that almost all of the traffic used the SNA/3270 protocol, rather than the TN3270 protocol. In addition, we had determined that the software for the two protocols was almost identical. Since we had very limited access to the test platform, minimizing the test suite was essential. This was accomplished by removing test cases that could be viewed as being redundant in the sense that they yielded little in the way of additional information about the system's performance. This, in turn, permitted additional scenarios to be tested which did yield new information.

We defined usage scenarios by assigning realistic values to the parameters that most directly affected performance. For most parameters, we were able to collect and analyze data from systems currently in use in the field. For a few parameters, we made assumptions based on our experiences with the existing system. We benefited greatly from the availability of a production-like test facility and commercially available tools used to collect and analyze performance data. The primary lesson learned was that it was far better to uncover performance problems before the system was deployed with customers dependent on it, than finding out about them when users' needs cannot be met.

It would be nice if we were able to provide a precise algorithm for the generation of test cases to use for the performance testing of a system being built, as we have for different levels of correctness testing. However, that is not possible. As we discussed above, however, there are a number of steps that we followed to assure that the workloads we used for performance testing would yield meaningful information that would be indicative of what users would likely see when the system was operational in the field. Just as it is essential that test suites used for functionality testing be created in a systematic way, it is also critical that we do carefully planned software performance testing. We believe that the steps that we have outlined above, as reflected in the case study described, should help testers charged with doing performance testing avoid ad hoc testing and focus on the most relevant parameters and workloads.

Even though this was a very large and complex system and no usage data had been collected prior to our involvement with the performance testing effort, we were nonetheless able to identify a manageable number of critical parameters that served as the focus for workload definition for performance testing. By providing moderate detail in the case study and describing the level of granularity that we found adequate for testing, we hope that readers will be encouraged to begin a similar performance testing effort.

ACKNOWLEDGMENTS

Ron DeBlock helped us understand the application and the software architecture of the gateway. He implemented the caller simulation programs, executed the test cases, and collected the performance results. This work was performed when F.I. Vokolos was with AT&T Labs.

REFERENCES

[1] http://www.digital.sapients.com/softtest.html.
[2] A. Avritzer and E.J. Weyuker, "Deriving Workloads for Performance Testing," *Software—Practice and Experience,* vol. 26, no. 6, pp. 613–633, June 1996.
[3] A. Avritzer and E.J. Weyuker, "Monitoring Smoothly Degrading Systems for Increased Dependability," *Empirical Software Eng. J.,* vol. 2, no. 1, pp. 59–77, 1997.
[4] A. Avritzer and E.J. Weyuker, "Investigating Metrics for Architectural Assessment," *Proc. IEEE Fifth Int'l Conf. Software Metrics (Metrics '98),* pp. 4–10, Nov. 1998.
[5] B. Beizer, *Black-Box Testing.* New York: John Wiley & Sons, 1995.
[6] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems.* Prentice Hall, 1981.
[7] E. Kit, *Software Testing in the Real World.* New York: Addison-Wesley, 1995.
[8] M. Loukides, *System Performance Tuning.* O'Reilly & Associates, 1990.
[9] B. Marick, *The Craft of Software Testing.* Englewood Cliffs, N.J.: Prentice Hall, 1995.
[10] C.U. Smith, *Performance Engineering of Software Systems.* New York: Addison-Wesley, 1990.
[11] F.I. Vokolos and E.J. Weyuker, "Performance Testing of Software Systems," *Proc. ACM Workshop Software and Performance (WOSP '98),* pp. 80–87, Oct. 1998.

Elaine J. Weyuker received a PhD in computer science from Rutgers University, an MSE. from the Moore School of Electrical Engineering, University of Pennsylvania, and a BA degree in mathematics from the State University of New York in Binghamton. She is currently a technology leader at AT&T Labs-Research in Florham Park, New Jersey. Before moving to AT&T Labs 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 NYU, she was on the faculty of the City University of New York. Her research interests are in software engineering, particularly software testing and reliability, and software metrics, and she has published more than 100 papers in those areas. She is also interested in the theory of computation and is the author of a book (with Martin Davis and Ron Sigal), *Computability, Complexity, and Languages*, 2nd ed., published by Academic Press.

Dr. Weyuker is a ellow of the ACM, an AT&T Fellow, and a senior member of the IEEE. She is on the board of directors of the Computer Research Association (CRA) and is a member of the editorial boards of *ACM Transactions on Software Engineering and Methodology (TOSEM)*, the *Empirical Software Engineering Journal*, and an advisory editor of the *Journal of Systems and Software*. She has been the secretary/treasurer of ACM SIGSOFT, on the executive board of the IEEE Computer Society Technical Committee on Software Engineering, is currently on the ACM Fellow Selection Committee, and was an ACM national lecturer.

Filippos I. Vokolos received the BS and MS degrees from the New Jersey Institute of Technology in 1982 and 1984, respectively, and the PhD degree from Polytechnic University in 1998. From 1980 to 1984, he was with RCA Astro-Electronics, developing software systems for satellite manufacturing and control. He spent 14 years with AT&T Bell Laboratories, where he worked to develop and implement methods and tools for testing large telecom applications. Currently, he is with Lucent Technologies, working on a software system for switch provisioning. Dr. Vokolos' technical interests are in the areas of software testing and quality assurance. In particular, he is interested in developing techniques that improve the effectiveness and efficiency with which we test industrial-strength software systems.