A Framework for Measurement Based Performance Modeling

Dharmesh Thakkar
University of Victoria
Victoria, BC Canada
dharmesh@uvic.ca

Ahmed E. Hassan
Queen's University
Kingston, ON Canada
ahmed@cs.queensu.ca

Gilbert Hamann, Parminder Flora
Research In Motion
Waterloo, ON Canada

ABSTRACT

Techniques for performance modeling are broadly classified into measurement, analytical and simulation based techniques. Measurement based performance modeling is commonly adopted in practice. Measurement based modeling requires the execution of a large number of performance tests to build accurate performance models. These performance tests must be repeated for every release or build of an application. This is a time consuming and error-prone manual process.

In this paper, we present a framework for the systematic and automated building of measurement based performance models. The framework is based on our experience in performance modeling of two large applications: the DVD Store application by Dell and another larger enterprise application. We use the Dell DVD Store application as a running example to demonstrate the various steps in our framework. We present the benefits and shortcomings of our framework. We discuss the expected reduction in effort due to adopting our framework.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques, Modeling techniques.

General Terms

Measurement, Performance

Keywords

Framework, Measurement, Modeling, Performance

1. INTRODUCTION

Performance modeling of software applications is of prime importance. Problems after the deployment of software applications are rarely due to functionality errors. Rather, most problems are concerned with the application not responding fast enough, crashing or hanging under heavy load, and other performance or capacity related problems [7].

Performance modeling, a part of software performance engineering, is concerned with building performance models to better understand the performance characteristics of an application under different workloads and deployment (hardware and platform) settings. Performance modeling techniques are broadly classified into measurement, analytical and simulation based techniques. Measurement based techniques rely on conducting extensive performance tests on the application being studied. Measurement based techniques can only be conducted once the application is fully developed and available. To overcome this limitation, analytical and simulation techniques build models to study and predict ahead of time the performance characteristics of an application. Analytical techniques use theoretical models. Simulation techniques emulate the functionally of the application using a computer simulation whose performance can be probed. There has been an in-depth research on the use all three techniques for performance modeling of software applications [4].

Both simulation and analytical based techniques require a good understanding of the application and require the presence of accurate documentation of its behavior. However, up-to-date and complete documentation and understanding of an application rarely exists in practice. The source code in many cases represents the only source of accurate information about the application [18]. Therefore practitioners commonly use measurement based techniques. Instead of building mathematical models or computer simulations, practitioners use the best model for a software application, the application itself! Measurement based techniques are often the only type of performance analysis used in practice, as noted by Sankarasetty et al. [1].

Measurement based techniques require the execution of a large number of performance tests for every release or build of a software application. A performance test measures the performance characteristics (e.g., response time) of the application for a specific workload under a particular hardware and software configuration. Performance tests are typically conducted after functional and load testing of an application is complete. Functional testing checks whether an application meets its functional requirements. Load testing checks whether the application works well under heavy workloads. Both functional and load testing result in a pass or failure classifications for each test. In contrast the results of a performance test are summarized quantitatively in metrics like response time, throughput and hardware resource utilizations. Using the results of a large number of performance tests, a performance model can be built. Deployers of enterprise applications use this performance model to determine the most suitable capacity configurations when deploying a new application [6, 11, 19]. This process is commonly refereed to as capacity planning.

To ensure that a performance model is complete and accurate a large number of performance tests must be conducted. The large number of tests leads to many challenges when performing measurement based modeling in practice. Setting up the environment for executing each test is usually a manual process, which is lengthy and error prone. Setup mis-configurations are common, costly, and are usually hard to detect. The test setup process is repeated a large number of times since tests are repeated many times. Tests are repeated to ensure the statistical validity of results and to study the performance of an application in different hardware and platform settings. With each build or version of a software application, the measurement based models must be updated by re-running most of the performance tests. Building and maintaining measurement based models is a time consuming and resource intensive process. For instance, if a bug is discovered in an application during performance modeling then the full performance modeling is usually repeated once the bug is fixed.

Much of the practice has focused on automating performance testing instead of modeling. Industry is primarily focused on building sophisticated load testing tools, such as WebLOAD [13] and HP LoadRunner [14]. Such tools although very valuable for performance testing, do not help address the full life cycle of measurement based performance modeling. Since measurement based performance modeling is one of the final steps in an already late release schedule, techniques are needed to speed up the modeling process. Practitioners require tools to assist them in building and updating measurement based models by automating the various steps in performance modeling.

In this paper, we propose a framework that encompasses the full life cycle of measurement based performance modeling. The framework automates the process of picking the appropriate load tests to execute in order to build an accurate and representative model. The framework then automates the setup of the load tests using off-the-shelf load testing tools such as HP LoadRunner. The framework also assists performance analysts in analyzing the results. The main contribution of our work is the proposal of a framework that brings together various venues of research to support analysts in their day-to-day activities. Using our framework researchers can explore contributing and fitting their own research work into the proposed framework. Moreover, analysts can compare various tools and techniques using the structure of our framework.

Paper Organization

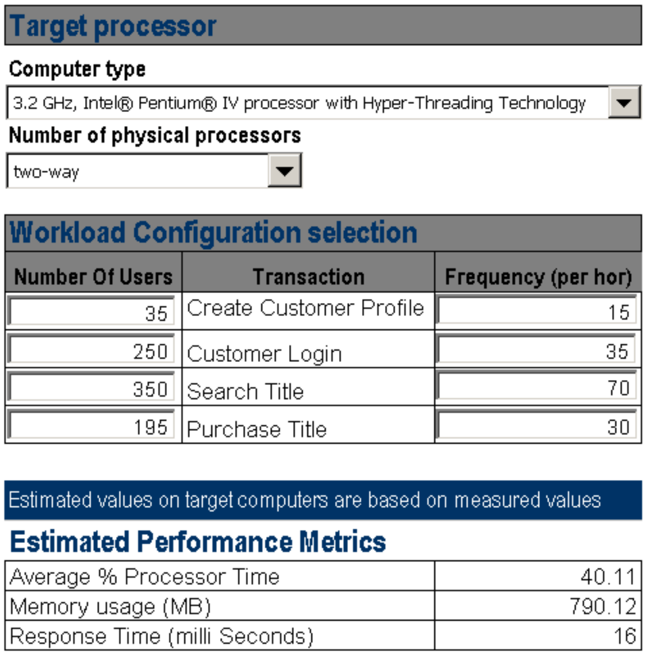
The paper is organized as follows. In section 2, we discuss the application of performance modeling in practice. Section 3 discusses the challenges associated with measurement based performance modeling. In section 4 we discuss how our framework addresses those challenges and present the various steps in our framework. Section 5 discusses the efforts needed to customize our framework. Section 6 covers the limitations of our framework. Section 7 presents related work and section 8 concludes the paper.

2. APPLICATION OF PERFORMANCE MODELING IN PRACTICE

Measurement based modeling of a software application is commonly used in practice to produce capacity calculators and performance white papers. Such calculators and white papers are commonly developed for hardware platforms (e.g. [6]) and large enterprise applications (e.g., [11, 19]). These calculators help customers in capacity planning activities. Capacity planning involves selecting the most appropriate configurations for deploying an application while satisfying performance requirements and financial constraints. When deploying enterprise applications, customers must determine whether their current deployment infrastructure, is over-engineered (then they can reduce deployment costs) or under-engineered (then they can invest more to improve the user's experience). For example, a capacity analysis for a web application may indicate that a desired response time of 8 milli-seconds cannot be achieved, if the application is servicing 200 requests per second (i.e., usage workload) while running on a dual P4-1.2Ghz machine with 2 GB of memory (i.e., hardware configuration). Other hardware configurations should be explored to achieve the desired response time. Customers and support staff would like to address issues such as:

1. What hardware is sufficient to deploy product X and offer a good user experience?
2. If I upgrade to version 3.x, will my current quality of service be affected? Will I need new hardware?
3. How much quality of service improvements should I expect if I upgrade my I/O subsystem?
4. If I enable another 100 users on my current hardware, what will be my CPU and disk utilizations?
5. When should I upgrade my current hardware given my expected workload growth?

Figure 1 shows an example of a capacity calculator for the online DVD Store application by Dell. The predictions produced by the calculator are based on the inputs given in the UI and a performance model for that application. For example, given a particular hardware configuration of a 3.2 two-way P-IV CPU, and the various workload parameters, as shown in Figure 1, the calculator produces the predications by the model. The model predicts an average CPU utilization of 40%, a memory usage of 790MB and a response time of 16ms. A customer could modify the hardware or workload configurations to determine a suitable configuration that would meet future demands and their budget.



Target processor

Computer type: 3.2 GHz, Intel® Pentium® IV processor with Hyper-Threading Technology

Number of physical processors: two-way

Workload Configuration selection

Number Of Users	Transaction	Frequency (per hor)
35	Create Customer Profile	15
250	Customer Login	35
350	Search Title	70
195	Purchase Title	30

Estimated values on target computers are based on measured values

Estimated Performance Metrics

Average % Processor Time	40.11
Memory usage (MB)	790.12
Response Time (milli Seconds)	16

Figure 1: An example of capacity calculator

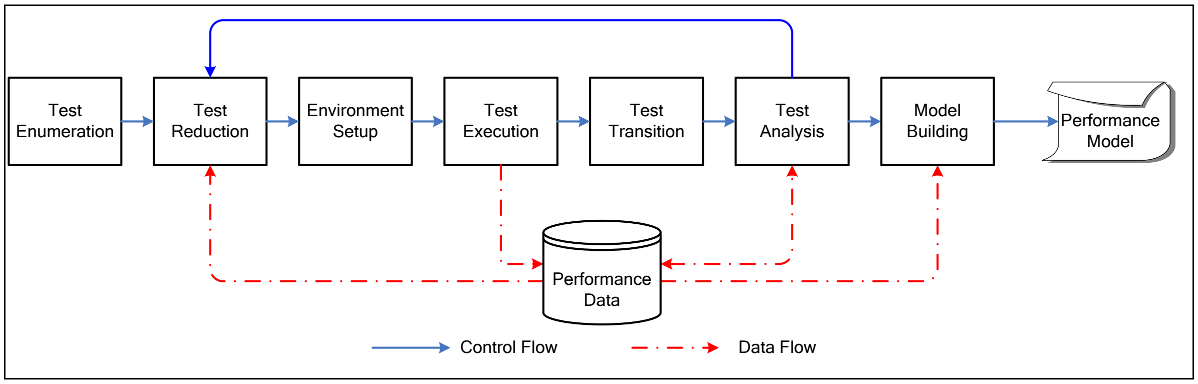


Figure 2: The conceptual framework for measurement based performance modeling

3. CHALLENGES OF MEASUREMENT BASED PERFORMANCE MODELING IN PRACTICE

Building a measurement based performance model is a challenging task in practice due to many of the following reasons:

1) **The large number of tests that must be executed.** A large number of tests must be executed in order to ensure that the model captures the various possible workload and configuration options for an application. For example, tests should be conducted for various configuration settings of an application. Tests may be repeated several times to gain statistical confidence in the captured performance metrics. Tests must be conducted on multiple platforms to model and benchmark the effects of changing underlying hardware platforms.

2) **The limited time that is available for performance modeling.** Performance modeling is usually done as the last step in an already tight and usually delayed release schedule. Hence managers are always hoping to reduce the time allocated for performance modeling.

3) **The risk of error due to the manual process that is followed to setup, execute and analyze the tests.** There exist many tools to help in automating the generation of loads for performance testing. However, there exist no tools for configuring the application under tests, setting up the tests, and analyzing the results in an automated fashion. In practice, all these tasks are done manually and are especially error prone.

4) **The risk of having to repeat the full modeling process.** All too often the modeling process reveals problems or mis-configurations are discovered. Once the identified problems are addressed, the modeling process must be restarted from scratch while having minimal impact on the time allocated for performance modeling.

Such challenges have been noted by other researchers and practitioners as well. For instance, Gunther cites lengthy measurement and modeling phases as the main reasons for management's skepticism towards performance modeling and capacity planning [8].

4. OUR PERFORMANCE MODELING FRAMEWORK

In this paper we propose a performance modeling framework which addresses the aforementioned challenges as follows:

1) **The large number of tests that must be executed.** Our proposed framework supports the use of advanced test selection and prioritization techniques such as ANOVA selection [9] and screening designs [10], to reduce the number of tests. The framework also supports the re-use of data from previous releases or builds of an application.

2) **The limited time that is available for performance modeling.** The framework automates many of the time consuming tasks needed for building performance models. The framework also reduces the time needed for tests.

3) **The risk of error due to the manual process that is followed to setup, execute and analyze the tests.** The framework automates the processes for setting up the environment, executing the tests and analyzing the tests. This automation ensures that errors are minimal. Moreover the framework contains a validation step which uses prior performance tests and heuristics to flag possible bad tests and to rerun them or remove them from the model building step.

4) **The risk of having to repeat the full modeling process.** The framework detects and flags possibly problematic or mis-configured performance tests. The modeling process can be automatically executed incrementally after the problems are addressed.

Figure 2 shows the various steps in our framework. The framework constitutes of the following steps:

1) **Test enumeration** determines the set of performance tests that should be executed. The aim of the test enumeration step is to define the search space of all the tests which should be executed to build an accurate performance model.

2) **Test reduction** uses domain knowledge and historical information from prior runs to reduce the number of performance tests. Moreover test reduction uses statistical and experimental design techniques to reduce the number of tests that should be run.

3) **Environment setup** automates setting up the environment for performance testing. This includes installing the application and load testing tools. The application and the tools may be required to run on different operating system platforms. To support multi platform applications, practitioners can customize this step and reuse the other steps across platforms.

4) **Test execution** automates the task of running the test suite. It has three major activities of: Test Setup, Test Run, and Test Shutdown. This step is customizable to allow the use of performance/load testing tool that can be invoked automatically (e.g., from the command line).

5) **Test transition** prepares the environment to execute the next performance test from the tests specified in the first step in our framework. The practitioner can configure the framework between one extreme of full restore and restart of the system under test and the other extreme of directly starting the load for the following test. Once configured, the framework automatically executes the transition steps after finishing each performance test.

6) **Test analysis** step first compares the test results against other test results and against heuristics to detect any issues with the performance test itself. Next, the metrics from the performance counters are analyzed to draw the relation between performance counters and load injected.

7) **Model Building** In this final step, a regression model is built using statistical analysis tools, which models the application performance as a function of its load parameters.

A performance database stores the performance test and analysis data. The database is used in the test reduction, test analysis and model building steps. The database could be implemented using sophisticated database systems, or using files.

The framework permits performance analyst to encode the various heuristics that are used in their model building process on a daily basis. By encoding the heuristics they ensure that their model building process is repeatable. The documentation of the heuristics permits analysts to closely examine these heuristics and update them as their understanding of the application matures. Analysts could also replace their heuristics with more sophisticated techniques as they evolve their modeling process. In the following subsections we describe in detail each step of our framework. We use the Dell DVD Store application as a running example to demonstrate the various steps of our framework.

We now briefly introduce the Dell DVD Store application. The DVD Store (DVD Store 2 or DS2) application is an open source enterprise software application. The DS2 application is developed by Dell as a benchmarking workload for white papers and demonstrations of Dell's hardware solutions [2]. DS2 seeks to emulate today's online stores, architecturally and functionally. DS2 has a three-tier architecture. DS2 includes application server components, database server components and a load generator engine (client emulator). The source code for the load generator is available and runs on various platforms. The load generator can generate load on different application servers, or directly generate load on the database server, skipping the application server altogether. The load generator emulates website users by sending HTTP requests to the application front-end. The application front-end encodes the various business rules, e.g. ordering new titles, declining an order in case of insufficient inventory. All customers, titles and transactional data are stored in the database server tier. We chose DS2 over other applications for many reasons. First, it is an open-source application, allowing us to debug and fix many problems with the application. Second, it is simple and straight forward to use, through a command line interface. Third, it does not require any commercial software to get it running; we could use Apache Tomcat as its application server and MySQL as its database server. We now explain each step of our framework for performance modeling using DS2 as a running example.

4.1 Test Enumeration

The first step towards performance modeling of a software application is to enumerate the list of performance tests which should be performed to build a performance model that would fulfill the requirements of customers. This step is the only manual step in our framework. Our framework automates the execution of the remaining steps. The test enumeration step consists of four phases. We discuss below each phase using the DS2 application.

Phase 1: Enumeration of functional transactions

The performance analyst begins with enumerating the functional transactions available in the application. For our case study, the functional transactions in the DS2 application are:

i. Creation of a new customer profile

ii. Customer login

iii. Searching for titles by category, actors, genre, etc.

iv. Purchasing a title

Phase 2: Mapping functional transactions to workload classes

The performance analyst needs to map the functional transaction to workload classes. Multiple transactions can be grouped and modeled as a single workload, or each transaction can be modeled as a separate workload. The analyst should decide based on the granularity and level of details required in the model. For example for the DS2 application, if we are not interested in modeling the performance demands of each individual transaction, we can consider a sequence of login-search-purchase transactions as a single workload, as done by researchers at Dell [6]. Rather, we decide to consider each transaction as a workload class. We consider that the sequence of login-search-purchase as a single workload may not be a valid assumption since a user might do several search operations before making a purchase.

Phase 3: Prioritizing workload classes for test execution

The workload classes should be prioritized since the framework will execute tests to ensure that each workload is represented in the final performance model. For instance, if the performance model is being built for a new release in which the purchase functionality has been modified to accept a new method of payment, the analyst may decide to only execute the tests corresponding to the purchase workload and to reuse the data for other tests from the older model of the application.

Phase 4: Picking the ranges for each workload class and the step size within a range

The range for each workload class and the step size within the range are picked based on the experience of the analyst, the requirements imposed on the final performance model, and historical knowledge about the application. For instance, if a particular setting would peg a hardware resource at full utilization, the workload might be too high for the system to handle so the range should be adjusted. In the absence of historical data, some trial and error might be required to decide the ranges and stepping size, so that the measurement points are evenly distributed. Now we enlist the settings available for the DS2 workload classes, so that we can enumerate the tests with different values of those:

1. Frequency of a transaction: Number of transactions per hour.
2. Concurrency: Number of processes or threads concurrently generating the load on the application.
3. Search categories: Search by name, category, actors or genre.
4. Purchase quantity: Number of DVDs purchased in one transaction.

The frequency and concurrency settings are applicable to all four workload classes. The search category settings are applicable only to search workload. The purchase quantity settings are applicable only to purchase workload. Table 1 shows the relation between the various settings and the workload classes. All four settings (frequency, concurrency, search, and purchase) have four levels.

Performance tests should be conducted at various combinations of the available settings for each workload. For instance, the Login workload class has 4 levels for the frequency and concurrency settings resulting in 16 possible combinations, for which a performance test needs to be conducted. Based on studying the documentation of the DS2 application, we decided not to consider the interaction between the workload classes, since each workload class has a service demand that is independent of the demands of any other workload class. Based on our assumption and the number of settings, we have enumerated a total of 144 performance tests, as detailed in Table 1. If a performance analyst were to consider the interaction between the workload classes, then the number of tests would be quite larger using a factorial experiment design technique [4], as the total number of tests would be the multiplication of the number of possible tests for each workload class.

Table 1: Performance test enumeration

Workload Class	Frequency Levels	Concurrency Levels	Other	Performance Tests
Create Profile	4	4	-	16
Login	4	4	-	16
Search	4	4	4 (search parameters)	64
Purchase	4	4	4 (purchase quantity)	64
Total Performance Tests				**144**

4.2 Test Reduction

Test reduction is the second step in our framework, shown in Figure 2. As discussed in Section 2, the large number of performance tests and long test durations are some of the key challenges in measurement based performance modeling. Hence, it is necessary to introduce this step in the framework to reduce the number performance tests. However, there has been little research interest in performance test reduction methods. In this section, we propose a few performance test reduction methods, borrowing ideas from other research areas. We classify these methods as one of two types: static and dynamic. The static test reduction is a manual process, requiring good knowledge of the requirements of the performance model and the implementation of the application. The dynamic test reduction methods are based on mathematical tools and techniques, which are built into the framework and are carried out automatically.

4.2.1 Static Test Reduction

There usually are several functional transactions in a large software application. However, all of the functional transactions may not be important for performance modeling. For instance, customers who want to deploy a DVD Store application would not be much interested in the performance of the admin functionalities. Rather, they would like to know how the store front performs in regards to customer operations. Hence, uninteresting functional transactions can be filtered out. Such a reduction method draws from the knowledge of the requirements.

Another set of reduction methods draws from the knowledge about the implementation. For instance, if two features are similar to each other, it might be sufficient to conduct performance tests on only one of them. For example, purchasing a DVD and purchasing a DVD Collection features might differ by only a few code modules, so the performance analyst can decide to build a model that captures only one of the features to reduce the number of needed tests, at least in the first iteration of model building.

4.2.2 Dynamic Test Reduction

The idea of test reduction has been researched thoroughly in the functional testing area [22, 23]. However, this idea has not been explored much for performance testing and modeling. We present a few approaches, which although used for other purposes, can be practically used here.

The Pareto principle [25] suggests that a small number of the application features account for majority of the issues. This principle is applicable to functional as well as performance issues. The dynamic test reduction techniques seek to identify those few features which contribute significantly to application performance and only execute the tests that correspond to these features.

In large applications, a few important workload classes with large service demands have great impact on the overall application performance, while other workload classes might have minimal or negligible performance impact. Once those workload classes with large service demands are uncovered using a few performance tests, further testing for such less important workload classes can be avoided with a little or no loss in accuracy. In [10] Porter et al. propose a method called Main Screen Analysis to find out the important configuration parameters that affect the application performance. Menascé and Sopitkamol used two-way ANOVA to rank the configuration parameters that significantly impact overall application performance in [9]. Both these works can be used to rank the workload classes according to their significance on overall performance. Once the performance parameters are ranked by their significance on performance, tests corresponding to the least affecting parameters can be dropped from analysis with minimal loss of accuracy. Techniques used by Porter et al., and Menascé and Sopitkomal are based on experimental design theory. A detailed discussion of experimental design techniques is presented in [4].

The framework supports using the aforementioned methods or other research work in a plug-and-play fashion. In our case study, we used a simplistic method for test reduction. We ran the two extreme performance tests for each workload class: one with the lowest value and another with the highest value from the entire array of workload sizes, as derived after the test enumeration step. For instance, we ran the test for Purchase workload with quantities: one, and one thousand. If the framework does not discover significant differences in performance between these two tests, the framework skips the tests corresponding to the intermediate values. However, if the framework discovers significant differences in performance due to the parameter settings (such as concurrency, frequency and search type), it conducts the remaining tests for those settings. Using this simplistic method we could reduce the number of tests from 144 tests to 64 tests. The reduced list of performance tests is shown in Table 2.

Table 2: Reduced list of performance tests for DS2

		Frequency (transactions per hour)			
		20	40	60	80
Concurrency 250	Creating Customer Profile	T111	T112	T113	T114
	Customer Login	T121	T122	T123	T124
	Search Title	T131	T132	T133	T134
	Purchase Title	T141	T142	T143	T144
Concurrency 500	Creating Customer Profile	T211	T212	T213	T214
	Customer Login	T221	T222	T223	T224
	Search Title	T231	T232	T233	T234
	Purchase Title	T241	T242	T243	T244
Concurrency 1000	Creating Customer Profile	T311	T312	T313	T314
	Customer Login	T321	T322	T323	T324
	Search Title	T331	T332	T333	T334
	Purchase Title	T341	T342	T343	T344
Concurrency 1500	Creating Customer Profile	T411	T412	T413	T414
	Customer Login	T421	T422	T423	T424
	Search Title	T431	T432	T433	T434
	Purchase Title	T441	T442	T443	T444

4.3 Environment Setup

The environment setup is the third step in our framework, as shown in Figure 2. This step is designed to install the application and the performance/load testing tools. Currently environment setup in the industry is a manual, ad-hoc and error-prone process. There has not been much research work on automating this step.

In our framework, we automated and implemented this step using a set of scripts in a stand-alone module, which is invoked by the framework engine. The scripts set up multiple computer systems – the application servers, database servers, load generators and performance tracking machines. The scripts then verify the correctness of environment setup by making sure that the relevant processes and services are running. However, each application and load testing tool has its own installation steps. Hence, we anticipate a significant amount of rework is required in this step when customizing the framework to another application or platform. We discuss the efforts needed for customizing our framework in Section 5. Despite the large customization efforts needed for this step, our experience using the framework shows that it is worthwhile to automate this step, considering that the customization effort is a one-time effort.

4.4 Test Execution

Conducting performance tests is a lengthy and tedious step. This major step is further divided into three sub tasks: test setup, test run and test shutdown.

4.4.1 Test Setup

Each component of the application may need a set of test data for a particular test. For instance, the DVD Store application in our case study needs to be loaded with test data of DVD titles, registered customers, and their purchase history. Another important task in test setup is the configurations of the application server, the database server and the load generator. Different setting of the configuration parameters values can lead to drastically different performance results. It is important to associate a performance test result with its configuration for the test analysis step. Our framework archives the configuration files of the application with each performance test.

Being confident that the tests are not affected by any one-off anomalies includes making sure that the application is in a correct state before triggering the test. Problems with test setup are not usually captured until the test analysis step, when the counters contradict themselves or do not match expectations. For this reason, it is of prime importance to validate the test setup.

Our framework allows the writing of custom routines for test data setup, configuration, and setup validation. These routines are invoked by the framework before triggering the test, thus allowing complete automation of test setup tasks. Our experience at using the framework shows that once these custom routines are implemented, they provide significant time savings.

4.4.2 Test Run

There has been considerable work in recent years in automating the running of load and performance tests. Sophisticated performance/load testing programs like LoadRunner and WebLOAD are available. These programs include 1) tools to record a script which represents the workload class that is being tested, and 2) tools to generate workload by playing multiple instances of the recorded scripts that emulate real-life concurrent users. To conduct the tests, multiple instances of the recorded scripts are played from the load generating machines, simultaneously probing the performance of the application. Once the scripts representing the workload classes are recorded, running of each test is a three step process:

1. Start the *performance counters*.
2. Turn on the *application*.
3. Start the *load generating tools*.

Starting of the performance counters can be the last or the first step in the process. However, starting the counters first allows capturing the transient response of the application while it is being turned on and the load is building up. Each of these three main components of the test setup might have multiple subcomponents that need to be turned on in appropriate sequence. Appropriate time gaps might be needed between the successive steps.

Similar to test setup, the framework achieves automation in running tests by allowing scripting and error checking of this important step in a modular way.

Each performance test goes through three phases:

1. Warm-up: Also known as ramp-up phase, during which the application is being subjected to the workload. However the workload is not at its full strength but it is building towards the designated workload level.
2. Steady-state: The warm-up phase gives way to the steady state phase if the environment is well configured and the application can sustain the workload. During this phase, the performance metrics are normally distributed with respect to the average.
3. Cool-down: Also known as ramp-down phase, during which the load generator gradually stops injecting the workload and the resource utilizations gradually drop as the workload is winding down.

4.4.3 Test Shutdown

The load generating tools should be shutdown. Often, the load generating tools are timed and can be setup to shutdown once a test is completed. The application under test may need to be triggered for shutdown or may continue running for the following tests. The decision to shutdown or to continue running the application is taken by the Test transition step. To remove the need for manual intervention, the framework manages this process with scripting and error checking.

4.5 Test Transition

Test transition is the process of switching from one performance test to the next. There are various approaches for test transition. The fastest way to transition is to conduct the tests back to back, meaning to start loading the application with the new workload, as soon as testing with the previous one is completed. This approach results in very fast test transition. However, it may not be recommended in all instances, since the residual load from the previous test may interfere with the next test. A slightly better transition approach is to add a delay, ranging from a few seconds to few minutes, between performance tests, so that the residual load would flow out of the system. The length of the delay can be determined experimentally. In practice, it is preferable to use a heuristic based transition approach. The approach uses heuristics which monitor a few metrics to determine if the residual load has flowed out and the system has reached idle state. For example, a check can be made on application resources to ensure that the next test is not triggered until the processor utilization of the application machine is below a particular threshold (e.g., 5%).

For some application domains, previous test data if continually accumulated can affect the results of the following tests. For example, mail server applications continuously accumulate emails so if the mail store is not cleaned up after every test, then the size of mail store will keep on increasing. With an ever increasing mail store size, the disk resource might show sluggish performance in the following performance tests. A regular archival process should be setup. After archival, fresh test data for a particular test should be loaded. The best approach for such application is to clean-up and restart the application after every test. The clean-up and restart approach would ensure that there is no interference between performance tests.

Similar to the previous steps, the framework manages to automate these tasks with modularity of invoking custom routines that carry out these transition tasks.

4.6 Test Analysis

Data derived from each executed test should be analyzed for absence of errors. Manually analyzing the performance counters and application logs for these purposes could be time consuming, tedious and repetitive task due to the large amount of produced data. Our framework goes a step beyond by not only automating the analysis for errors, but also using the analysis for test reduction and model building.

The framework triggers the analysis of the results automatically after a test is completed. The major tasks of validating the test and analyzing the metrics are discussed in the subsections below.

4.6.1 Validate the Test

Several problems can arise during a performance test. For example:

- A functionality bug, e.g. a memory leak or inefficient implementation which results in a drift of the hardware resources towards instability during the test.
- An interference from other processes or applications such as automatic download and install of critical OS patches, or disk backup, These processes would cause abrupt changes in resource availability and would lead to invalid values for the performance counters.
- A physical aspect, such as the rise in the operating temperature of the data center housing the application under test. This temperature rise may lead to invalid performance counters.

Such problems leave the performance test data unusable for analysis and model building. To detect such problems, the framework invokes validation routines, which check if the application reached and maintained stability during the performance test and all counters are within their expected bounds. Moreover the logs produced by the application are mined to detect any execution anomalies which may indicate bugs in the application. There exists various log mining techniques to detect bugs from logs [5]. A performance analyst can choose a technique based on their needs.

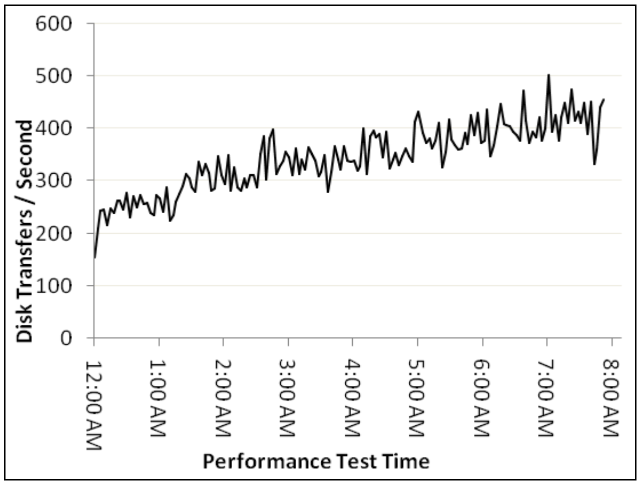


Figure 3: Instability in Resource Utilization

A simple way to detect instability is the method of central moving average, which filters short term fluctuations and highlights long-term trends. The instability in Figure 3 could be easily detected algorithmically using this method. The method would show that the hardware resource usage keeps on growing throughout the test and never stabilizes.

Our implementation of the validation module for the DS2 does four types of validations:

1. If the application reaches and maintains steady state during the test, but the utilization of a resource is above 90% then we flag that test as unusable for modeling purposes. The reason being, that measurement data at high utilizations are hardly reliable and repeatable [20]. Furthermore, all scheduled tests at higher settings than the current test are skipped (as part of the test reduction step). This technique helps avoid wasting time in conducting performance test which would produce invalid data due to overloading of the application.
2. If the application does not reach steady state (exhibit ever increasing or ever decreasing trend in resource utilization), then we flag the test as unusable for modeling purposes. However, the framework continues executing the tests at higher workload settings, unlike the previous case, because instability in the current test may not necessary result into instability in tests at the higher workload settings.
3. If a performance test with the same workload was executed previously (for a previous or same build/version), and the measured metrics (utilization, response time, throughput) differ by a configurable boundary value, the framework flags the current test as a possibly bad run. The performance analyst can then do further analysis of such bad runs. After solving any issues, the framework can run incremental modeling tests, only executing the performance tests that were flagged out previously.
4. If the logs of the application show errors during a performance test show, then we flag the test as unusable for modeling purposes. However, the framework continues executing the tests at higher workload. The performance analyst can override this decision and incorporate the results of this test in the modeling if they deem that the reported errors are not performance critical.

After all the tests are automatically executed by the framework and results are presented to the performance analysts. If there are any failures, manual debugging may be required to find the root cause of test failure. Once problems are fixed, the flagged test can be re-run. The modularity and automation in the framework allows the re-running of all or only a subset of the performance tests.

Using our framework's validation step, we identified a dead-lock bug in the DS2 application. The application server tier would first open a connection to the database in order to allow customer to login and query its purchase history. Then the application would open a second connection to browse the titles related to the titles in the customer history. Within a few minutes of running a test, all the threads in the application server would end-up waiting for the second connection after capturing the first connection. As it was not possible with the current MySQL driver to reuse connection, we modified the code to do not query the purchase history and related titles. We had to fix the bug to allow us to conduct performance tests at concurrency levels that are significantly higher than previously modeled for DS2. Once we fixed the bug, we could perform modeling at higher concurrency levels

4.6.2 *Metric Analysis*

For each performance test, the counters collected during the warm-up and cool-down periods should be pruned from the analysis, while the counters from the steady state time period are carried forward for analysis. Then, counters are imported in a statistical analysis package, such as R [24], and statistical functions are applied to derive the average performance metric values.

Traditionally metric analysis has been a tedious manual task in performance modeling studies. We automated this task by creating a script module that is invoked by the framework. The scripts chop off the performance counter data captured during the warm-up and cool-down periods of each test. We keep the length of the warm-up and cool-down period configurable in the framework, to allow it to be easily customized for different applications. Finally, the framework obtains the average metric values and stores the values in the performance database (see Figure 2) for the modeling effort.

We observed that for the DS2 application, a warm-up period of ten minutes was enough to reach steady state. The cool-down period for DS2 was negligible because of the way the load generating tool operates – it does not ramp-down the load during the trailing period of a test, it rather drops the load from its determined levels to zero when the test time is up. However, many performance analysts choose to keep the warm-up and cool-down period quite longer, to show the longevity and sustainability that are desired in commercial application.

4.7 Model Building

In the previous step, the framework produced the performance metrics at different workload sizes. In this step, the framework invokes the R [24] statistical tool which builds a linear or nonlinear regression model for the performance of the application. Figure 4 shows an example of fourth order regression model between response time and processor utilization. Once a regression model is built, performance predictions at arbitrary load levels are done using the fitted model. For a comprehensive discussion of regression models and prediction techniques, refer to [4], particularly chapters 14 and 15. The developed regression model could be used as a backend for a capacity calculator.

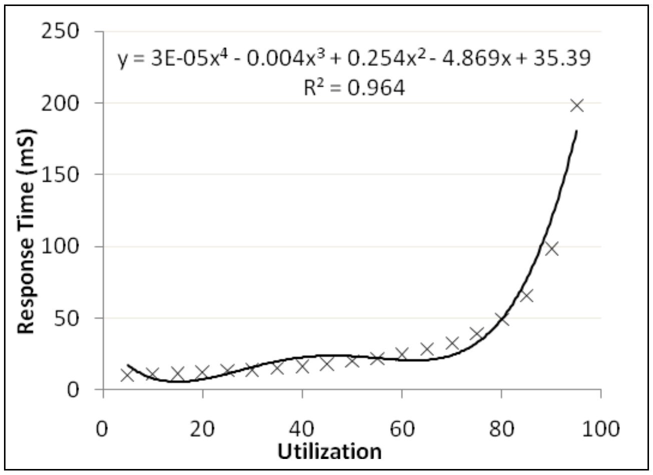


Figure 4: Processor Utilization vs Response Time

5. CUSTOMIZATION EFFORT

A major benefit of adopting our framework is the ability to reuse modeling efforts when building performance models for other applications; or other platforms, versions and builds of the same application. In addition to using the framework for building a performance model for the DS2 application, we are using the framework for performance modeling of a large multi-platform enterprise application.

When reusing the framework, several steps in the framework need to be customized to achieve automation. Table 3 indicates the amount of effort needed to customize the steps in our framework. We classify the customization efforts as Minimal, Reasonable or Extensive. Minimal efforts are characterized by a quick review of the step; most of the implementation would be applicable as it is, with little changes needed. Reasonable efforts imply the need for changing or rewriting of some parts of the implementation of that step. Extensive efforts are characterized by a major rewrite of the implementation for that step.

We anticipate that the efforts to customize the framework for another build to be minimal, because all the steps would be applicable, as they are. For another version of the same application, reasonable efforts may be required in test enumeration and reduction, considering that new features introduced in the version would result in additional workloads which should to be tested and modeled. The rest of the framework would still be applicable as is. To customize the framework for the same application running on a different platform, the setup and transition steps would need a rewrite of most of the implementation, resulting in extensive effort requirement. However, spending the extensive efforts to customize the framework would show returns many times, which would easily justify the cost. To customize the framework for a different application deployed on the same platform, reasonable efforts are required in the setup, execution and analysis steps because changes to the automation scripts are needed.

Table 3: Estimated efforts for customizing our framework

Framework Step	Another Build	Another Version	Another Platform	Another Application
Test Enumeration	Minimal	Reasonable	Minimal	Extensive
Test Reduction	Minimal	Reasonable	Minimal	Reasonable
Environment Setup	Minimal	Minimal	Extensive	Extensive
Test Execution	Minimal	Minimal	Reasonable	Extensive
Test Transition	Minimal	Minimal	Reasonable	Reasonable
Test Analysis	Minimal	Minimal	Minimal	Reasonable
Model Creation	Minimal	Minimal	Minimal	Minimal

6. LIMITATIONS

The proposed framework is based on our research and experience in measurement based modeling of two applications: the Dell DS2 application and another large enterprise application. These applications are complex enterprise applications but they may not represent the entire class of enterprise applications. Additional steps and limitations may be discovered while applying the framework to other applications.

We integrated research from other researchers to automate various steps in our framework. However, limited research was available in a few of the steps, so we employed heuristics in those steps. However our framework directs researchers to focus on these areas. Moreover the encoding of those heuristics in the framework ensures that the repetitive tasks corresponding to those heuristics are not missed and could be later revisited by practitioners.

Some of the dynamic analysis activities are currently not automated in the framework and a performance analyst must conduct these activities manually. This is our first attempt at building this framework, which can be extended further with research work focusing on each of the following points.

- Adjusting the performance tests to precisely determine various key operational points or objects, e.g. knee capacity and bottleneck resources. Unless the tests are carefully designed, the built model can be inaccurate near such operational points.
- Adjusting the performance testing period and lengths of ramp-up and cool-down periods. This mainly involves determining how long the application takes to reach steady state condition and how many data points we need in each test, to be confident enough about the input data and analysis results.

7. RELATED WORK

Goldsmith et al. present a measurement based technique for modeling computational complexity, to avoid relying only on theoretical asymptotic analysis [3]. Similar to their work, our framework aids in measurement based modeling, rather than analytical or simulation modeling. Their modeling effort is for algorithmic performance of non-Markovian applications [4]. In contrast, our modeling effort is for enterprise applications which are Markovian in nature, i.e., service demands for each new request in a workload is independent of previous requests and the current state of the application. Moreover, the presented framework would prove to be extremely useful to Goldsmith et al. in performing and managing the numerous performance tests required to empirically measure computational complexity.

A tool called JUnitPerf built by Clarkware Consulting helps automate performance testing during the development cycle [15]. JUnitPerf helps reuse the unit tests written in JUnit [16] for performance testing of code units, as the developers finish coding and refactoring. JUnitPerf is valuable for performance testing during the development cycle. However, our framework is used to model the overall performance of the whole application before shipping instead of modeling a particular unit of code.

Mania and Murphy present a framework for automated LQN based performance modeling [12], which is derived from the trace-based modeling technique proposed by Woodside et al. [17]. Mania and Murphy's work is limited to analytical performance modeling, in particular LQN based modeling. In contrast, our work derives its models using measurement based modeling techniques.

Smith et al. propose a process for building software and system performance models from UML models [21]. Their work provides a framework for analytical performance modeling. In comparison, our framework is for measurement based performance modeling.

8. CONCLUSION

We presented a framework for performance modeling of software applications. The need for such a framework is felt from the current challenges in performance modeling practices in industry. Performance analysts are continuously building and updating performance models for enterprise applications. These models are produced through a labor intensive and error prone process which always occurs at the end of already late release schedules.

Our proposed framework automates the building of measurement based performance models. The framework is based on our experience in performance modeling of two large applications: the DVD store application by Dell Corporation and another larger enterprise application. We presented the limitations of our framework and highlighted our experience in using it. Moreover we discussed the effort involved in customizing our framework for other applications and other platforms. We conclude that more work is required to unify and automate the processes for performance modeling across the industry. More attention is required from academia on the use of measurement based techniques, which have wider acceptance in the industry, compared to other analytical and simulation based techniques.

Acknowledgement

We are grateful to Research In Motion (RIM) for providing access to large enterprise systems. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of RIM, its subsidiaries or affiliates. Our results do not in any way reflect the quality of products of RIM, its subsidiaries or affiliates.

9. REFERENCES

[1] Sankarasetty, J., Mobley, K., Foster, L., Hammer, T., and Calderone, T. 2007. Software performance in the real world: personal lessons from the performance trauma team. In *Proceedings of the 6th international Workshop on Software and Performance* (Buenes Aires, Argentina, February 05 - 08, 2007). WOSP '07. ACM, New York, NY, 201-208.

[2] Jaffe, D., Muirhead T. 2005. The Open Source DVD Store Application. http://www.dell.com/downloads/global/power/ps3q05-20050217-Jaffe-OE.pdf

[3] Goldsmith, S. F., Aiken, A. S., and Wilkerson, D. S. 2007. Measuring empirical computational complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, September 03 - 07, 2007). ESEC-FSE '07. ACM, New York, NY, 395-404.

[4] Jain R. 1992. The art of computer systems performance analysis. John Wiley.

[5] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceeding of the 28th international Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 282-291.

[6] Muirhead T., Jaffe, D. 2005. Migrating enterprise databases from Sun servers to the Dell PowerEdge 2850 running Microsoft Windows Server 2003. http://www.dell.com/downloads/global/power/ps1q05-20040270-Jaffe.pdf

[7] Woodside, M., Franks, G., and Petriu, D. C. 2007. The Future of Software Performance Engineering. In *2007 Future of Software Engineering* (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 171-187.

[8] Gunther, N. J. 2006 *Guerrilla Capacity Planning: a Tactical Approach to Planning for Highly Scalable Applications and Services*. Springer-Verlag New York, Inc.

[9] Sopitkamol, M. and Menascé, D. A. 2005. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *Proceedings of the 5th international Workshop on Software and Performance* (Palma, Illes Balears, Spain, July 12 - 14, 2005). WOSP '05. ACM, New York, NY, 53-64.

[10] Yilmaz, C., Krishna, A. S., Memon, A., Porter, A., Schmidt, D. C., Gokhale, A., and Natarajan, B. 2005. Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *Proceedings of the 27th international Conference on Software Engineering* (St. Louis, MO, USA, May 15 - 21, 2005). ICSE '05. ACM, New York, NY, 293-302.

[11] Research In Motion. Capacity calculator for BlackBerry Enterprise Server 4.1 for Microsoft Exchange. http://www.blackberry.com/select/toolkit/dls/BlackBerry_Enterprise_Server_Version_4.1.0_for_Microsoft_Exchange_Capacity_Calculator.xls

[12] Mania D. and Murphy J. 2002. Framework for predicting the performance of component-based systems. In *Proceedings of IEEE 10th International Conference on Software, Telecommunications and Computer Networks* (Italy, October 2002). SoftCOM 2002. pp. 46-50, ISBN 953 6114 52 6.

[13] WebLOAD load testing stress testing tool. http://www.webload.org/

[14] HP LoadRunner Software. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5E8_4000_100__

[15] http://www.clarkware.com/software/JUnitPerf.html

[16] http://www.junit.org

[17] Israr, T. A., Lau, D. H., Franks, G., and Woodside, M. 2005. Automatic generation of layered queuing software performance models from commonly available traces. In *Proceedings of the 5th international Workshop on Software and Performance* (Palma, Illes Balears, Spain, July 12 - 14, 2005). WOSP '05. ACM, New York, NY, 147-158.

[18] S. E. Sim. 1998. Supporting multiple program comprehension strategies during software maintenance. *Masters thesis*, University of Toronto, 1998.

[19] Research In Motion. BlackBerry Enterprise Server for Microsoft Exchange version 4.1 performance benchmarking. http://www.blackberry.com/knowledgecenterpublic/livelink.exe/fetch/2000/8067/645045/7963/7965/1180408/Performance_Benchmarking_Guide.pdf?nodeid=1367404&vernum=0

[20] Pentakalos, O. and Friedman, M. 2002. *Windows 2000 Performance Guide: Help for Windows 2000 Administrators*. UMI Order Number: 4665., O'Reilly & Associates, Inc.

[21] Smith, C. U., Lladó, C. M., Cortellessa, V., Marco, A. D., and Williams, L. G. 2005. From UML models to software performance results: an SPE process based on XML interchange formats. In *Proceedings of the 5th international Workshop on Software and Performance* (Palma, Illes Balears, Spain, July 12 - 14, 2005). WOSP '05. ACM, New York, NY, 87-98.

[22] Xie, T., Marinov, D., and Notkin, D. 2004. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In *Proceedings of the 19th IEEE international Conference on Automated Software Engineering* (September 20 - 24, 2004). Automated Software Engineering. IEEE Computer Society, Washington, DC, 196-205.

[23] Rothermel, G. and Harrold, M. J. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (Apr. 1997), 173-210.

[24] The R project for statistical computing. http://www.r-project.org/

[25] Juran, J. M., Godfrey A. B. 1988. Juran's Quality Handbook. McGraw-Hill Professional.