iModel: Automatic Derivation of Analytic Performance Models

MAHMOUD AWAD and DANIEL A. MENASCÉ, George Mason University

Deriving analytic performance models requires detailed knowledge of the architecture and behavior of the computer system being modeled as well as modeling skills. This detailed knowledge may not be readily available (or it may be impractical to gather) given the dynamic nature of production computing environments. This article presents a framework, called *iModel*, for automatically deriving and parameterizing analytic performance models for multi-tiered computer systems. Analytic performance models consist of a workload model and a system model. *iModel* uses system logs and configuration files to generate a high-level characterization of the system; e.g., open queuing network (QN) model versus closed QN model. By harvesting more information from the system logs and configuration files, *iModel* generates a workload model by inferring user-system interaction patterns in the form of a Customer Behavior Model Graph (CBMG) and generates a system model by discovering system components and their interaction patterns in the form of a Client-Server Interaction Diagram (CSID). *iModel* includes a library of well-known single-queue and QN models and their solutions stored in an XML-based repository. The generated workload model and system model are compared to the model repository to determine which model in the repository best matches the system's observable behavior and architecture. This article also presents a black-box optimization approach that is used to derive analytic model parameters by observing the input-output relationships of a real system. This optimization approach can be used in any computer system (multi-tier or not) that can be modeled by single queues or QNs. The important question is whether the automatically generated and parameterized performance model has predictive power, i.e., can the derived model predict the output values that would be observed in the real system for different values of the input? The results presented in this article demonstrate that the analytic performance models derived by *iModel* are relatively robust and have predictive power over a wide range of input values.

CCS Concepts: • **Mathematics of computing** → **Queueing theory**; Cluster analysis; Nonlinear equations; • **Applied computing** → *Operations research*;

Additional Key Words and Phrases: Queuing network models, parameter estimation, non-linear optimization, black-box optimization, MVA, AMVA

ACM Reference format:

Mahmoud Awad and Daniel A. Menascé. 2020. *iModel*: Automatic Derivation of Analytic Performance Models. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 5, 2, Article 7 (March 2020), 30 pages.
<https://doi.org/10.1145/3374220>

The work of D. A. Menascé was partially supported by AFOSR Grant No. FA9550-16-1-0030.

Authors' addresses: M. Awad and D. A. Menascé, Computer Science Department, George Mason University, 4400 University Blvd., Fairfax, VA 22030, USA; emails: mawad1@masonlive.gmu.edu, menasce@gmu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

2376-3639/2020/03-ART7 \$15.00

<https://doi.org/10.1145/3374220>

1 INTRODUCTION

Computer systems are becoming increasingly complex in terms of hardware and software. Consequently, it is becoming difficult and costly to manually configure, maintain, and protect computer systems, especially in large data centers. Analytic performance models are generally used to plan the capacity and help determine the most cost-effective architectural alternatives for computer systems. One of the main barriers to the adoption and wide-spread use of analytic performance models is that they require a thorough understanding of the system architecture, software architecture, application behavior, resource usage, and workload characteristics. This knowledge is then provided to highly skilled performance engineers who use specialized modeling techniques to develop analytic performance models tailored to a particular computer system. Because large production systems may evolve over time to cope with changes in the workload and user requirements, added functionality, and adoption of new technologies, existing analytic performance models may have to be constantly updated or rebuilt. Additionally, complex environments may require different performance models for different times of the day or times of the year as well as different models for different application environments, such as in the case of multi-tenant data centers. Therefore, it is very costly and, sometimes, impractical to manually build analytic performance models.

This article describes a framework, called *iModel*, for the automatic derivation of analytic performance models, such as single queues and open and closed multiclass queuing network (QN) models. The framework uses the system's software logs and configuration files to infer observable system behavioral and architectural patterns, which are then used to automatically determine the workload model parameters and the model queue interconnections. The inferred model parameters and queue structure are compared to a library of workload models and system models to find the best-fit queuing model that matches the characteristics of the observed system behavior and configuration.

To fully parameterize the best-fit model, *iModel* calculates unknown model parameters, such as the service demand per class and per device using non-linear optimization techniques. System measurements captured in the system logs are divided into time intervals and, for each interval, the difference between the measured response time and model-derived response time are calculated. The optimization techniques minimize the sum of square differences between measured and model-computed response times under the constraint that the sum of all differences across all intervals is zero.

The framework presented in this article uses semantic analysis of the system behavior and configuration to produce accurate workload model parameters using Customer Behavior Model Graphs (CBMG) [20] and Client-Server Interaction Diagrams (CSID) [18]. A CBMG is a state transition graph that represents the behavior exhibited by a group of users when accessing an application system. Each node in a CBMG represents a business function and the arcs of the graph capture user navigational patterns. A Client-Server Interaction Diagram (CSID) represents the internal system component interconnections required to execute a business function.

System logs are described using a formal language for efficiently and dynamically harvesting log contents instead of manually developing custom-built parsers for every type of system log. At the same time, queuing models and their solution techniques are stored in a knowledge base using an XML-based syntax inspired by the Performance Model Interchange Format (PMIF) [28], which speeds up the process of comparing the system's observed behavior and workload parameters with the stored analytic performance model characteristics.

The major contributions of this article are: (1) A methodology and framework for automatically deriving accurate performance models using system logs and configuration files. The methodology is presented in the context of multi-tiered systems and includes (a) an approach for automatically

analyzing system behavior and components based on observable system parameters and (b) an approach for estimating service demands for all transaction classes at all queues. This approach can be applied to any computer system (multi-tiered or not) that can be modeled by single queues or QNs. (2) A knowledge base to store well-known queuing models and their solutions expressed in closed-form or algorithmic techniques. (3) The implementation of the methodology and framework, which includes parsing and analyzing the system logs, deriving CBMGs and CSIDs, querying the knowledge base and estimating service demand values using optimization methods. This article aggregates and adds new material to several of our previous publications [2–5], so that the *iModel* framework is presented in a holistic and enhanced way.

The rest of this article is organized as follows. Section 2 provides a summary of performance modeling approaches. Section 3 includes notation and terminology. Section 4 describes the *iModel* workflow. Section 5 describes the non-linear optimization techniques used to derive unknown service demands in single queues (Section 5.2) and in single and multi-class open (Section 5.3.2) and closed QNs (Section 5.3.1). Section 6 discusses related work. Section 7 presents concluding remarks and future work.

2 APPROACHES FOR BUILDING PERFORMANCE MODELS

Three approaches can be considered for building performance models:

- (1) *Bottom-up modeling (aka white-box modeling)*. This approach begins by taking an inventory of all the system components and their interconnections, followed by instrumenting the system to obtain model parameters and, finally, building the model and calibrating it. This detailed approach can be as accurate as needed. However, it requires full access to the computer system, which may be difficult in production systems. It is also time consuming, costly and requires modeling expertise.
- (2) *Top-down modeling (aka black-box modeling)*. This modeling approach considers the system's inputs and outputs only and finds the performance model that satisfies those relationships. This approach is generally simple, fast, and cheap. However, the resulting model tends to have low accuracy and little or no predictive power. For example, regression can be used to model the relationship between the system's input (e.g., average arrival rate) and output (e.g., average response time). The disadvantage of using regression alone in performance modeling is that it does not have predictive power. Formulas produced by regression are limited to establishing the relationship between input and output variables at a certain point in time. Changes in the input-output measurements may produce a different regression model. For example, a regression model is not able to predict the impact of replacing a one-CPU server with a dual-CPU server on the system's response time.
- (3) *Intermediate-level modeling (aka gray-box modeling)*. This approach (used in this article) relies on logs generated by the production system and its configuration files to obtain knowledge about the system components and their interconnections, and the system's workload model, thereby inferring the values of system parameters without the need to take measurements. The approach can be fully automated, does not require modeling expertise, is cheap (since it is automated) and more accurate than top down modeling, and in many cases as accurate as bottom up modeling. System logs and configuration files are mostly available in production systems, and access to them does not disturb the systems operation.

3 NOTATION AND TERMINOLOGY

Table 1 presents the terminology used in this article and Table 2 the notation used here.

Table 1. Terminology

Term	Explanation
CBMG	Customer Behavior Model Graph. Each node in a CBMG represents a business function and the arcs capture user navigational patterns [20]
CSID	Client-Server Interaction Diagram. Represents the internal system component interconnections required to execute a business function [18]
Service time	Time spent by a transaction receiving service during a single visit to a device
Service Demand	Average of the sum of the service times of a transaction over all visits to a device
KB	Knowledge Base. XML-based representation of performance models and their solutions
Concurrency level	Average number of jobs of a class concurrently executing

Table 2. Notation

Notation	Explanation
For Queuing Networks	
R	Number of job classes in a QN model
K	Number of devices in a QN model
$D_{i,r}$	Service demand of class r jobs at device i .
λ_r	Average arrival rate of class r jobs
$\vec{\lambda}$	$(\lambda_1, \dots, \lambda_r, \dots, \lambda_R)$
N_r	Concurrency level of class r requests
$\vec{N}$	$(N_1, \dots, N_r, \dots, N_R)$
For Single queues	
λ	Average arrival rate of requests
$\bar{x}$	Average request service time
$\bar{x}^2$	Second moment of the service time
C_a	Coefficient of variation of the interarrival time of requests
C_s	Coefficient of variation of the service time
T	Average response time

4 IModel WORKFLOW

We introduced a preliminary version of the *iModel* framework in Reference [4]. This section expands on it by adding an XML-based Knowledge Base repository that stores performance models and their solutions (see Section 5.2).

Figure 1 shows the steps of *iModel* and the inputs of each of these steps, shown as numbered rectangles for ease of reference. The inputs to the various steps are shown as database icons or as ovals. In the description that follows, we consider, by way of example, that the system is a three-tier system that consists of an Apache Web server, an Apache Tomcat application server running Apache OFBiz, which is an open source enterprise automation system, and a MySQL database server. The software servers were hosted on two Ubuntu 14.04 Linux machines. Any other servers and machines could be used instead.

Step 1: Identify System Components. This step identifies the system components, i.e., servers, machines, and deployments of servers to machines. This is accomplished by parsing system

Fig. 1. Flowchart of *iModel*.

Table 3. Snippet of an Apache Tomcat Configuration File

```

worker.balance1.type=lb
worker.balance1.balance_workers=worker1, worker2
worker.worker1.type=ajp13
worker.worker1.host=host1
worker.worker1.port=8080
worker.worker2.type=ajp13
worker.worker2.host=host2
worker.worker2.port=8080

```

configuration files such as the Apache system configuration file, Tomcat's system configuration file, and MySQL configuration file. In this example, Tomcat's configuration file and debug log were parsed to find the application host and database host. Table 3 shows a snippet of an Apache Tomcat configuration file indicating two load-balanced servers.

To automatically parse log and configuration files in steps 1–3, one needs to be able to automatically parse and extract meaningful information from logs having various formats. Different web servers, application servers, database servers, and operating system logs vary in terms of log format between required and optional parameters, their meaning and syntax. To address this problem, we used Logstash, which is part of the ELK stack (see <https://www.elastic.co/products>). Logstash is an open source data processing tool that ingests and transforms data from a multitude of sources.

The output of step 1 is a list of system components (i.e., servers and devices) and the deployment of servers to devices (i.e., machines, identified by IP addresses in this case). Figure 2 shows

Fig. 2. Example of the output of Step 1. There are three servers (a web server and an application server deployed to machine M1 and a database server deployed to machine M2).

Table 4. Filter for Apache Access Log File

```
GROK FILTER:
COMMONAPACHELOG %-IPORHOST:CLIENTIP" %-USER:IDENT"
  %-USER:AUTH"["%-HTTPDATE:TIMESTAMP" "]" "(?:%-WORD:VERB"
  %-NOTSPACE:REQUEST"(?: HTTP/%-NUMBER:HTTPVERSION")?—"
  %-DATA:RAWREQUEST")"%-NUMBER:RESPONSE" (?:%-NUMBER:BYTES"—)"
COMBINEDAPACHELOG %-COMMONAPACHELOG" %-QS:REFERRER" %-QS:AGENT"
GROK -MATCH => - "MESSAGE" => "%-COMBINEDAPACHELOG"
  %-NUMBER:HTTP"REQUEST"DURATION`MS"
  %-QS:HTTP"SESSION"ID" ""
```

an example of the output of this step. Shown in the figure are three servers (Web Server (WS), Application Server (AS), and Database Server (DS)), two devices (machines 1 and 2), and the deployment of servers to devices indicating that servers WS and AS are deployed to machine 1 (M1) and server DS is deployed to machine 2 (M2).

In this implementation, the objective was to avoid intrusive techniques that assumed access to the database server and its operating system. In future implementations, server names and IP addresses could be used to identify the hardware configuration of every hardware server to produce more detailed hardware queuing network (HQN) models. In cloud platforms, *iModel* can utilize telemetry data collection services, such as OpenStack's Ceilometer module [23], to build the system's HQN at the infrastructure level more accurately, thereby eliminating the need for parsing system logs.

Step 2: Derive User-System Interaction Patterns. This step parses the Apache Web Server access logs using an access log filter template, shown in Table 4, to derive the interaction patterns between users and the functions provided by the system. An exemplary snippet of an Apache access log is given in Table 5. The access log filter shown in Table 4 is a pattern used by grok, a tool of Logstash that parses unstructured log data into structured data.

An example of the output of this step is shown in Table 8, which shows that for a given session from a user identified by IP address 192.168.1.207, the user has made several interactions in the accounting/control path. These interactions are indicated by "From" state to "to" state as in from [1-/path/findInvoices] to [2-/path/invoiceOverview]. A detailed description of step 2 was presented in Reference [3] and will not be described in further detail here.

Step 3: Derive Client-Server Interaction Diagrams (CSID). Client-Server Interaction Diagrams (CSID) represent the internal system component interconnections and service invocations during the execution of a business function [18] (see, e.g., Figure 3 for an example of the CSID for the *findPayments* function). A client is represented by a rectangle with a "C" inside and servers are represented by circles with the acronym of the server inside. Directed arrows indicate the pattern

Table 5. Snippet of an Apache Access Log

```

192.168.1.207 - - [10/APR/2016:21:32:29 -0400]
"GET /ACCOUNTING/CONTROL/INVOICEOVERVIEW?INVOICEID=CI1 HTTP/1.1" 200 27731
"HTTPS://192.168.1.111:8443/ACCOUNTING/CONTROL/FINDINVOICES"
2061 "CFA387B88E6378F6BCBA6EDD8A2A4988.ws2"
192.168.1.207 - - [10/APR/2016:21:32:36 -0400]
"GET /ACCOUNTING/CONTROL/NEWINVOICE HTTP/1.1" 200 17565
"HTTPS://192.168.1.111:8443/ACCOUNTING/CONTROL/INVOICEOVERVIEW?INVOICEID=CI1"
872 "CFA387B88E6378F6BCBA6EDD8A2A4988.ws2"

```

Fig. 3. Example of CSID for the *findPayments* function shown in the CBMG of Figure 5.

of communication, i.e., server request invocations. The labels associated with these arcs represent the probabilities that a given arc is traversed during a single execution of the function (*findPayments* in the example of Figure 3). Servers in a CSID may have a multiplicity factor indicated by a numeric label next to the CSID node that represents the server. This factor indicates the number of load-balanced instances of the server.

An analysis of a CSID can provide the average number of visits per server during a single execution of the function associated with the CSID. This number has to be multiplied by the number of visits to the function (derived from the CBMGs as explained in References [18, 20]) to obtain the average number of visits to each server over all executions of the function.

Step 3 uses the following inputs: (1) output of step 1, (2) output of step 2, (3) application log filter template, and (4) application logs. The application log filter template is used to parse the application logs using Logstash. The output of this step is a set of CSIDs.

Deriving the CSIDs was achieved by parsing the Tomcat application server debug log. Given the interaction patterns from Step 2, the CSIDs were generated for each business function in the CBMG by identifying the transaction boundaries of every business function in the debug log, and determining when the application server was communicating with the web server, and when it was communicating with the database server. This can be done by correlating unique session ids and date/time information as they appear in the web server and application server logs.

Table 6 shows the application log filter using grok's syntax and Table 7 shows a snippet of the application log.

An example of a CSID obtained by step 3 is shown in Figure 3 for the *findPayments* function. The figure also shows the average number of visits V for each server within the nodes of the CSID. Of note here is the fact that 369 visits on average were required to the database server (DS) for the execution of this function. The reason is that this particular application uses auto-complete as a user types a character string and a visit to the database is required as each additional character is typed by the user.

Table 6. Application Log Filter File

GROK FILTER:	
TOMCAT`DATESTAMP	20%-YEAR"-%-MONTHNUM"-
%-MONTHDAY" %-HOUR":	
?%-MINUTE"(?:?%-SECOND")	JAVACLASS (?:[A-Za-Z0-9-]+)+
[A-Za-Z0-9\$]+	
GROK -MATCH => -"MESSAGE" => "	%-TOMCAT`DATESTAMP:TIMESTAMP"
(%-DATA:THREAD")	"["s+%-JAVACLASS:JCLASS" "-%-NUMBER:LINENUMBER"
“-%-LOGLEVEL:SEVERITY""]	MAHMOUD "- %-GREEDYDATA:SHORTMESSAGE"" ""

Table 7. Snippet of the Application Log

2015-11-17 19:02:08,351 (HTTP-BIO-0.0.0.0-8443-EXEC-1)
[ORDERLOOKUPSERVICES.JAVA:63 :INFO] MAHMOUD - FINDORDERS

Step 4: Determine System Component Interconnections. This step uses two inputs: (1) the output of step 1 (i.e., a list of system components (i.e., servers and devices) and the deployment of servers to devices (i.e., machines)); and (2) the CSIDs produced by step 3.

In n-tier application systems, a single interaction on one tier could produce one or more interactions on the next tier. For example, a single hyperlink labeled Proceed to Checkout on an e-commerce website may correspond to multiple web service API calls, each of which might be implemented using a Java class on the application server. Each Java class may issue one or more database transactions to finalize the checkout process and store the purchase transaction history along with audit and security log entries. One of the problems addressed by *iModel* is the ability to correlate the various logged events in the components of a multi-tier application system, and as a result determine which low-level interactions in one tier correspond to the higher-level interactions on the previous tier.

Some authors suggested embedding unique identifiers in system logs via probes or specialized software agents and passing them as parameters while the transaction passes from one application tier to the next [11]. The approach used by *iModel*, however, is non-intrusive and relies on existing system logs with certain assumptions related to the content and format of those logs, such as logged events must have a timestamp (preferably down to the sub-second). From prior experience, all logs of major web, application and database servers satisfy such requirement. System logs collected by telemetry data collection services in cloud platforms eliminate the need for correlating multiple system log events at the application level, since they collect measurements at the infrastructure level.

An example of the output of step 4 is depicted in Figure 4, showing a system model in the form of a SQN-HQN (i.e., software queuing network-hardware queuing network) as defined in References [19, 21]. The SQN is a queuing network in which each queue is associated with a software server. The HQN is a queuing network in which the queues are associated with hardware servers or their components (e.g., processors and I/O devices). An SQN requires as input the service demand of each software module on each hardware component [19, 21]. Other representations of system models, such as the method of layers (LQN), could have been used as well [26].

It is worth noting here that some steps of the *iModel* workflow, such as inferring hardware and software component interconnections, can be simplified (or replaced) in systems that use Service

Fig. 4. Example of the output for Step 4.

Fig. 5. Example of a CBMG for the Accounting-Payments application.

Function Chaining [12, 31] to model system component interconnections or web service orchestration in service-oriented systems.

Step 5: Build CBMGs. This step takes as input the interaction patterns produced by step 2 and generates CBMGs as discussed in detail by the authors in Reference [3]. This step entails generating a CBMG for each business function whose steps are executed by the user, including the probability of transitioning from each state of the CBMG to another. This probability is estimated by computing the relative frequency of transitions out of each state from the log. Then, CBMGs for different sessions can be clustered using the methods described in Reference [18]. See Figure 5 for an example of a CBMG.

Step 6: Combine CBMGs with CSIDs. The CBMGs produced by step 5 are combined in step 6 with the CSIDs produced by step 3 to generate a complete workload model and system model. This step also uses the output of step 1, because it needs to know the number of each type of server used in the system. The CBMGs can be used to compute the average number of visits to each external business function as discussed in Reference [18]. Finally, the analysis of the paths of each CSID allows us to compute the combined probability of traversing each path of the CSID. Then, the output of step 6 is a complete workload model that combines the number of visits to each external business function with the number of visits generated to each server of the system as a result of executing the business function. The deployments of servers to devices (obtained in step 1) allow us to obtain the number of visits to each device as the sum of the number of visits to each server on that device. Consider for example Figure 2, where the number of visits to device M1 is the sum of the number of visits to servers WS and AS.

Table 8 shows the user-system interaction patterns, which were used to build the Payments CBMG in Figure 5 and the Invoices CBMG (not shown here). These interaction patterns are the result of parsing the Tomcat access log.

Table 9 shows the results of parsing the application server debug log, part of which was used to build the CSID in Figure 3. A business function executing in the application server may need to

Table 8. Interaction Patterns (to be Read Top to Bottom, Left to Right)

Interaction Pattern for IP: [192.168.1.207] p = path= accounting/control			
From	To	From	To
/myportal/control/main	/p/main	/p/invoiceOverview	/p/newInvoice
/p/main	/p/findPayments	/p/newInvoice	/p/LookupPartyName
/p/findPayments	/p/findPayments	/p/newInvoice	/p/createInvoice
/p/findPayments	/p/newPayment	/p/createInvoice	/p/LookupPartyName
/p/newPayment	/p/LookupPartyName	/p/createInvoice	/p/listInvoiceItems
/p/newPayment	/p/createPayment	/p/listInvoiceItems	/p/LookupProduct
/p/createPayment	/p/LookupPartyName	/p/listInvoiceItems	/p/createInvoiceItem
/p/createPayment	/p/findInvoices	/p/createInvoiceItem	/p/LookupProduct
/p/findInvoices	/p/findInvoices	/p/createInvoiceItem	/p/logout
/p/findInvoices	/p/invoiceOverview	/p/createInvoiceItem	/p/main

Table 9. Result of Parsing the Application Server Debug Log

Vws: Number of visits from AS to WS Vds: Number of visits from AS to DS					
WS Vws	<— AS —> Business Function Running at AS	DS Vds	WS Vws	<— AS —> Business Function Running at AS	DS Vds
1	main	158	1	LookupPartyName	10
1	findPayments	154	1	LookupPartyName	10
1	findPayments	215	1	LookupPartyName	10
1	newPayment	151	1	LookupPartyName	10
1	LookupPartyName	14	1	LookupPartyName	10
1	LookupPartyName	12	0	createInvoice	172
0	createPayment	165	1	LookupPartyName	14
1	LookupPartyName	10	1	listInvoiceItems	160
1	LookupPartyName	12	1	LookupProduct	14
1	findInvoices	153	1	LookupProduct	38
1	findInvoices	585	1	LookupProduct	12
1	invoiceOverview	189	0	createInvoiceItem	168
1	newInvoice	156	1	LookupProduct	10

visit the database server to retrieve data from the database tables. It may also need to communicate with the web server, mainly to render web pages based on the results retrieved from the database and processed inside the application server.

One can think of the CSID of Figure 3 as a graphical representation of the information shown in Table 9. The CSID shows the average number of visits to the web server and to the database server as the business function steps (e.g., *findPayments*) are being processed at the application server. For example, the second line of Table 9 shows that the *findPayments* business function step communicated with the web server once and with the database server an average of 154 times. The third line shows a second execution of *findPayments*, which communicated 215 times, on average, with the database server (in this case, *findPayments* calls itself due to an asynchronous JavaScript request (AJAX) to set session variables). Therefore, in the process of executing the *findPayments*

Table 10. Workload Model Results Obtained by Parsing the Debug Log

Workload Model	
Total experiment time	191252 msec
Total Application Server (AS) completions	23
AS throughput	0.12 tps
Total Database Server (DS) completions	2611
DS throughput	13.65 tps

step, the interaction between the user and the internal system components is represented in the CSID as: User $\rightarrow$ WS $\rightarrow$ AS $\rightarrow$ DS $\rightarrow$ AS $\rightarrow$ WS $\rightarrow$ User. The average number of visits to DS is 369 (=154 + 215).

Note that the average number of visits in the transition WS $\rightarrow$ AS might be different from the average number of visits in the transition DS $\rightarrow$ AS. This is the reason why CSIDs do not use a loop back arrow to a previous node, and instead repeat the node (as in AS and WS) to account for that possible difference in the average number of visits.

The remaining results of the workload model, obtained by parsing the debug log, are shown in Table 10.

Step 7: Finalize Model Parameterization. Before the system model generated by step 4 is ready to be solved, one needs its workload model, obtained as an output of step 6, and the service demands for all servers and devices. If a manual process were to be used, then one would take detailed measurements of the system, which is typically time-consuming and not possible in many production systems. Given that the goal of *iModel* is to automate the entire process, we resort to techniques that estimate the service demands as described in detail in Section 5. In that section, we cast the problem of finding parameters for single queue systems (Section 5.2), multiclass closed QNs (Section 5.3.1) and open QNs (Section 5.3.2) as minimization problems in which the decision variables are the unknown parameters (e.g., service demands), the objective function is the sum of the squares of the differences between the end-to-end response time (computed as a function of the decision variables) and the measured end-to-end response time.

Step 8: Solve the Model. This step uses well-known techniques to solve multiclass QNs that include software servers as well as hardware devices to generate the steady-state solution of the model [19, 21, 26].

Step 9: Calibrate the Model. The accuracy of the model, with respect to measured response times, is checked at regular intervals. If the accuracy is not within acceptable limits, then the model needs recalibration by recomputing the service demands using the methods described in step 7. Often, calibration refers to reparameterizing and solving the model by calculating new (more accurate) service demands. However, in some cases, calibration may require re-inspecting the system logs or re-evaluating the accuracy of the system components inferred and the various interaction patterns between the user and the system and between the system components.

5 AUTOMATIC COMPUTATION OF SERVICE DEMANDS

This section is partially based on References [2, 5] by the authors but also extends that work to single queues. Additionally, it is worth noting that the approach described in this section can be used for any computer system (multi-tiered or not) that can be modeled as a single queue or a QN. Figure 6 illustrates the general process used by *iModel* in its step 7 to derive the service demand (or service time in case of single queues) parameters for single queues, open and closed QNs. First, we consider an input sequence $S = \{(t_i, R_i)\}$, for $i = 1, \dots, n$, which is an ordered set

Fig. 6. Automatic service demand parameterization of queuing models.

of transaction completion instants, t_i , and their response times R_i . That sequence is broken down by the algorithm shown in Section 5.1 into subsequences $S_j, j = 1, \dots, m$ such that the workload intensity in each subsequence, measured either in terms of average arrival rate for open systems or concurrency level for closed systems, is pretty much homogeneous, within a certain tolerance. The set of average response times for all subsequences is used as input to the computation of the service demands as illustrated in Sections 5.2 and 5.3. Another input of this step is the system model, i.e., the queues and their interconnections. This step solves an optimization problem in which the decision variables are the service demands (and other unknown parameters) and the objective function to be minimized is the sum of the squares of the errors between measured response times and those computed by the model for all subsequences.

To validate the method, we use the service demands found by the optimization problem to solve the model for a set of subsequences different from the ones used in the optimization. We then compare response time and throughput results from the model with those obtained from the system. To that end, we first used analytic models (either single queues or QNs) as proxies for an actual system. Then, we used a three-tiered OFBiz real system in the validation (see Section 5.4).

5.1 Generation of Subsequences

The following steps are taken to generate the subsequences $S_j, j = 1, \dots, m$ from the input sequence $S = \{(t_i, R_i)\}$, for $i = 1, \dots, n$ (see Figure 6). Then, for each subsequence, the values of the average throughput, X_j^a , average concurrency level n_j , and average response time, R_j^a , are computed according to the steps below.

- Step 1: initialization:** $\text{TotalRespTime} \leftarrow 0$; $\text{StartTime} \leftarrow 0$; $j \leftarrow 1$; $n_j^{\text{prev}} \leftarrow 0$; $i \leftarrow 0$; $\text{ntrans} \leftarrow 0$;
- Step 2: accumulate response time:** $i \leftarrow i + 1$; $\text{ntrans} \leftarrow \text{ntrans} + 1$; $\text{TotalRespTime} \leftarrow \text{TotalRespTime} + R_i$;
- Step 3: compute concurrency level [19]:** $n_j^{\text{curr}} \leftarrow \text{TotalRespTime} / (t_i - \text{StartTime})$
- Step 4: check tolerance η :** If $|n_j^{\text{curr}} - n_j^{\text{prev}}| / n_j^{\text{curr}} > \eta$, then $n_j^{\text{prev}} \leftarrow n_j^{\text{curr}}$ and go to Step 2;
- Step 5: compute outputs:** $n_j \leftarrow n_j^{\text{curr}}$; $R_j^a \leftarrow \text{TotalRespTime} / \text{ntrans}$; $X_j^a = \text{ntrans} / (t_i - \text{StartTime})$.
- Step 6: new subsequence?:** If $(i + 1) \leq n$, then $j \leftarrow j + 1$; $\text{ntrans} \leftarrow 0$; $\text{TotalRespTime} \leftarrow 0$; $\text{StartTime} \leftarrow t_i$; Go to Step 2

Note that the above algorithm can be used to determine subsequences for both open and closed systems: For open systems, we use the throughput X_j^a in each subsequence j computed in Step 5 as the average arrival rate for the subsequence, and for closed systems, we use the average concurrency level n_j , also computed in Step 5, as the workload intensity for subsequence j . An example

Table 11. Example of Interval Computation Based on Real Data for $\eta = 0.005$

i	Completion time (t_i)	Response time (sec) (R_i)	TotalRespTime	n_j^{curr}	n_j^{prev}	Error	$\bar{n}_j^{\text{curr}}$
29	27.52	13.23 0	132.872	4.83	4.62	0.0431	-
30	27.63	3.336	136.208	4.93	4.83	0.0207	-
31	29.06	18.77	154.978	5.33	4.93	0.0755	-
32	28.95	3.663	158.641	5.48	5.33	0.0267	-
33	30.3	4.01	162.651	5.37	5.48	0.0207	-
34	30.49	3.197	165.848	5.44	5.37	0.0132	-
35	32.6	16.311	182.159	5.59	5.44	0.0264	-
36	33.39	4.102	186.261	5.58	5.59	0.0017	5.32
-	-	-	-	-	-	-	-
46	41.42	5.129	260.197	6.28	6.39	0.0164	-
47	41.85	6.563	266.76	6.37	6.28	0.0144	-
48	42.19	29.899	296.659	7.03	6.37	0.0936	-
49	42.28	4.993	301.652	7.13	7.03	0.0144	-
50	42.10	22.807	324.459	7.71	7.13	0.0744	-
51	44.10	29.812	354.271	8.03	7.71	0.0405	-
52	43.87	5.583	359.854	8.20	8.03	0.0206	-
53	47.01	3.722	363.576	7.73	8.20	0.0606	-
54	47.33	3.038	366.614	7.75	7.73	0.0016	7.36

of the execution of the above algorithm is given in Table 11 that was generated from snippets of an actual analysis of the log of a real operational system. The table shows two identified subsequences, one including entries numbered 29–36 and the other for entries numbered 46–54. The average concurrency level for the first subsequence shown in the table is 5.32 and is equal to 7.36 for the second. Their coefficients of variation are 0.05 and 0.09, respectively, i.e., very small. The average response times are 8.3 and 12.4 sec, respectively, in these subsequences.

The next two subsections illustrate how service demands are estimated from measured workload intensity levels and measured response times for these workload intensity levels. Workload intensity level is defined as average throughput (the same as average arrival rate for open systems) and average concurrency level (for closed systems).

5.2 Parameterization of Single Queues

iModel includes a knowledge base (KB) of single queues and open and closed QN models with their solutions. The documents in the KB are described in an XML-based syntax inspired by the Performance Model Interchange Format (PMIF) [28]. More details about the KB component of *iModel* can be found in [4]. The XML schema of the KB can be found at https://cs.gmu.edu/~menasce/KB_XML_Schema/. From the workload characterization step, *iModel* computes the average arrival rate λ_j for subsequence S_j from the analysis of the input trace. If C_a , the COV for the interarrival times, is approximately equal to 1, then *iModel* determines that the arrival process is Poisson. Then, using XQuery, an XML query language, *iModel* retrieves from the KB all models (and their solutions) that fit the arrival process. For example, if $C_a \approx 1$, then *iModel* searches the KB for all models that have a Poisson arrival process and returns for example the solutions for M/M/1, M/G/1, M/M/c, and M/G/c. Otherwise, it retrieves models for a generic arrival process (e.g., G/G/1, G/G/c).

Some of the solutions in the KB are exact (e.g., M/M/1, M/M/c, and M/G/1) and some are approximate (e.g., G/G/1, G/G/c, and M/G/c) [19]. For all of the single queue models, the values of the average arrival rate λ and coefficient of variation C_a of the interarrival times are known from

the analysis of the input trace. The unknown parameters needed to solve the models in the KB are the average service time $\bar{x}$, the coefficient of variation C_s of the service time, and the number of servers c . The solution to any of these models is of the form $T = f(\lambda, C_a, \bar{x}, C_s, c)$, where $f()$ is a function of the known and unknown parameters, which are obtained by solving an optimization problem as described below.

The measured average response time in subsequence S_j is R_j^a . Then, the relative error ξ_j between the measured response time in subsequence S_j and the response time that would be computed by a model applied to subsequence S_j , with arrival rate λ_j is

$$\xi_j = \frac{R_j^a - T}{R_j^a} = \frac{R_j^a - f(\lambda_j, C_a, \bar{x}, C_s, c)}{R_j^a}. \quad (1)$$

Note that $\bar{x}$ and C_s do not change with the workload intensity in subsequence S_j .

As an example, the function $f(\lambda_j, C_a, \bar{x}, C_s, c)$ for an M/G/1 system is

$$f(\lambda_j, C_a, \bar{x}, C_s, c) = \bar{x} + \frac{\lambda_j \bar{x}^2 (1 + C_s^2)}{2(1 - \lambda_j \bar{x})}. \quad (2)$$

A non-linear optimization problem is used to find the values of decision variables $\bar{x}$, C_s , and c that minimizes the sum of the square errors across all subsequences subject to the following constraints: (1) The sum of the errors across all subsequences is zero. This constraint is used here and in all optimization problems henceforth to minimize the variance of errors (see Reference [13]). (2) The average service time has to be positive. (3) The coefficient of variation of the service time cannot be negative (this constraint is not used for models that assume exponential service times, because C_s is known). (4) The number of servers has to be a positive integer (this constraint is not used for models that have one server, because c is known). (5) The utilization has to be less than 1.

$$\text{Minimize} \quad \sum_{j=1}^m \xi_j^2, \quad (3)$$

$$\text{s.t.} \quad \sum_{j=1}^m \xi_j = 0, \quad (4)$$

$$\bar{x} > 0, \quad (5)$$

$$C_s \geq 0, \quad (6)$$

$$c \in \mathbb{Z}^+, \quad (7)$$

$$\lambda \bar{x} / c < 1. \quad (8)$$

Table 12 shows an example of deriving parameters for a single queue system known to have a single server and a Poisson arrival process. The unknown parameters are the average service time $\bar{x}$ and the coefficient of variation of the service time C_s . The second column has arrival rates in tps, and the third column has measured response times R_j^a for these arrival rates. The response time values in the odd numbered rows were used to solve the optimization problem above. The result was $\bar{x} = 1.807$ sec and $C_s = 1.578$. We then used these values and the M/G/1 formula to compute the response time values T_{pred} for the even numbered rows. The last column shows the percent relative error defined as $100 \times (T_{\text{pred}} - R_j^a) / R_j^a$. As we can see, the error is very small and less than 0.68% in absolute value.

5.3 Parameterization of QNs

In general, a QN model computes the average response time R_r for class r transactions through a function f_r such that $R_r = f_r(\mathbf{D}, \vec{W})$ where $\mathbf{D} = [D_{i,r}]$ is the matrix of service demands and $D_{i,r}$

Table 12. Example of Parameter Estimation for Single Queues

	λ_i	R_j^a	T_{pred}	percent rel. error
1	0.05	2.11	-	-
2	0.18	3.28	3.27	-0.43
3	0.28	5.06	-	-
4	0.33	6.50	6.47	-0.51
5	0.38	8.70	-	-
6	0.39	9.40	9.34	-0.68
7	0.41	10.83	-	-
8	0.42	11.70	11.74	0.33
9	0.44	13.40	-	-
10	0.45	15.50	15.54	0.23

is the service demand of class r ($r = 1, \dots, R$) transactions at queue i ($i = 1, \dots, K$). The vector $\vec{W} = (w_1, \dots, w_r, \dots, w_R)$ is the vector of workload intensities per class. In the case of open QNs, $w_r = \lambda_r$, the average arrival rate of class r transactions. For closed QNs, $w_r = N_r$, i.e., the average concurrency level of class r . The techniques described in this section are used to derive the matrix of service demands $\mathbf{D}$ given a series of values of response times R_r and the workload intensity levels for which they were obtained.

5.3.1 Parameterization of Closed QNs. This subsection focuses on deriving the service demands that can be used to build a closed QN model of an operational system for which it is possible to measure the response time and throughput of transactions submitted to the system. To illustrate the idea, we start with a single-class closed QN and then generalize to multiple classes. In the single-class case, the matrix of service demands $\mathbf{D}$ becomes the vector $\vec{D} = (D_1, \dots, D_i, \dots, D_K)$ of service demands where D_i is the service demand of transactions at device i .

Let $f(\vec{D}, n)$ be a function of the vector $\vec{D}$ of service demands and of the concurrency level n . This function computes the average response time $R(n)$. There is no closed form equation for the function f , only computational algorithms such as Mean Value Analysis (MVA) and Approximate Mean Value Analysis (AMVA) [19, 25]. The problem can then be formalized as follows. Given the subsequences $(n_1, R_1^a), \dots, (n_j, R_j^a), \dots, (n_m, R_m^a)$ find the vector of service demands $\vec{D}$. Remember that n_j is the average concurrency level and R_j^a is the average response time, both measured for subsequence S_j . For each subsequence S_j define the response time error ξ_j as

$$\xi_j = R(n_j) - R_j^a = f(\vec{D}, n_j) - R_j^a. \quad (9)$$

The service demands $\vec{D}$ can now be obtained by solving the following optimization problem.

$$\begin{aligned}
 &\text{Minimize} && \sum_{j=1}^m \xi_j^2, \\
 &\text{s.t.} && \sum_{j=1}^m \xi_j = 0, \\
 &&& D_i \geq 0 \quad i = 1, \dots, K, \\
 &&& \sum_{i=1}^K D_i \leq \min_{j=1, \dots, m} \{R_j^a\}.
 \end{aligned} \quad (10)$$

Table 13. Sequence of Values
of n_j , R_j^a , and X_j^a

n_j	R_j^a (in sec)	X_j^a
2	7.09	0.2821
4	10.71	0.3736
5	12.53	0.3991
6	14.36	0.4177
8	18.06	0.4430
9	19.92	0.4518
10	21.79	0.4589

The decision variables for the above problem are the service demands in $\vec{D}$, the objective function to be minimized is the sum of square errors, the first constraint forces the sum of the errors to be equal to zero, the second constraint indicates that the service demands cannot be negative, and the last constraint says that the sum of the service demands is the lowest possible value for the response time, i.e., the response time when there is no congestion.

This is a non-linear optimization problem that does not have a closed form solution for the objective function nor for its first constraint. Thus, we resort to black-box optimization methods. Solving the optimization problem above for the data in Table 13 using Excel's Generalized Reduced Gradient (GRG) non-Linear solver provides the following values for service demands: $D_1 = 1.512$ sec, $D_2 = 1.781$ sec, and $D_3 = 2.008$ sec. It is important to note that the errors for each subsequence are all exactly equal to zero even though we do not have a constraint that requires that.

To check if the above service demands provide predictive capabilities, an MVA model is solved using these service demands for $n = 7$ and 11 and the following response time values were obtained: 16.291 and 23.850 sec, respectively. Figure 7 shows the measured throughput as a function of the concurrency level according to Table 13 (black triangles) and the two points, for $n = 7$ and 11 (unfilled squares) predicted by a closed QN model that uses the derived service demands resulting from the solution of the optimization problem described above. The original service demands (i.e., ground truth) for Figure 7 are $D_1 = 1.5$ sec, $D_2 = 2.0$ sec, and $D_3 = 1.8$ sec. As it can be seen, the points computed by the QN model that uses the service demands estimated by this method match the curve obtained through measurements.

To further illustrate the power of the technique described here, we present another example in which the system is balanced (i.e., the service demands are the same for all devices). The plot in Figure 8 shows throughput values measured from a system (black triangles) for concurrency levels equal to 2, 4, 6, 8, 9, 11, and 14. Applying this method yielded the following service demands: $D_1 = 1.503$ sec, $D_2 = 1.501$ sec, and $D_3 = 1.496$ sec for original service demand values equal to $D_1 = D_2 = D_3 = 1.5$ sec. The red curve in the figure shows the throughputs obtained by using a closed QN model that takes as input the service demands obtained by solving the optimization model. The figure shows that the throughput values obtained for concurrency levels equal to 1, 3, 5, 7, 10, and 13 (i.e., values not used in solving the optimization model) match exactly what they should be.

Having presented the single-class closed QN case, we show below an obvious generalization for the case with R classes:

$$R_r(\vec{N}) = f_r(\mathbf{D}, \vec{N}), \quad (11)$$

$$\xi_{j,r} = R_r(\vec{N}_j) - R_{j,r}^a = f_r(\mathbf{D}, \vec{N}_j) - R_{j,r}^a, \quad (12)$$

Fig. 7. Throughput (in tps) vs. concurrency level. Triangles: measured. Squares: predicted by a model.

Fig. 8. Throughput (in tps) vs. concurrency level for a balanced system. Triangles: measured. Red Curve: predicted by the model.

where $\vec{N}_j = (n_{j,1}, \dots, n_{j,r}, \dots, n_{j,R})$ is the population vector observed at subsequence S_j , $n_{j,r}$ is the observed average concurrency level of class r transactions within S_j , and $f_r(\mathbf{D}, \vec{N})$ is computed using AMVA [19]. The optimization problem then becomes

$$\begin{aligned}
 & \text{Minimize} && \sum_{r=1}^R \sum_{j=1}^m \xi_{j,r}^2, \\
 & \text{s.t.} && \sum_{r=1}^R \sum_{j=1}^m \xi_{j,r} = 0, \\
 & D_{i,r} &\geq 0 & i = 1, \dots, K \text{ and } r = 1, \dots, R,
 \end{aligned} \tag{13}$$

$$\sum_{i=1}^K D_{i,r} \leq \min_{j=1,\dots,m} \{R_{j,r}^a\} \quad r = 1, \dots, R. \quad (14)$$

5.3.2 Parameterization of Open QNs. Similar to the closed QN case, we consider the measurements within each subsequence. For each of them, we can measure the average response time per class, $R_{j,r}^a$, and the average arrival rate $\lambda_{j,r}^a$ per class.

The equations for the closed QN case are easily transformed into the open QN case by changing the workload intensity vector from $\vec{N}$ into the vector of arrival rates $\vec{\lambda} = (\lambda_1, \dots, \lambda_r, \dots, \lambda_R)$. The last constraint was added to the optimization problem to guarantee that the utilization $U_i = \sum_{r=1}^R \lambda_r \times D_{i,r}$ of device i for all devices $i = 1, \dots, K$ is less than one:

$$R_r(\vec{\lambda}) = f_r(\mathbf{D}, \vec{\lambda}), \quad (15)$$

$$\xi_{j,r} = R_r(\vec{\lambda}_j) - R_{j,r}^a = f_r(\mathbf{D}, \vec{\lambda}_j) - R_{j,r}^a, \quad (16)$$

where $\vec{\lambda}_j$ is the vector of arrival rates within S_j and the optimization problem is

$$\begin{aligned} \text{Minimize} \quad & \sum_{r=1}^R \sum_{j=1}^m \xi_{j,r}^2, \\ \text{s.t.} \quad & \sum_{r=1}^R \sum_{j=1}^m \xi_{j,r} = 0, \\ & D_{i,r} \geq 0 \quad i = 1, \dots, K \text{ and } r = 1, \dots, R, \\ & \sum_{i=1}^K D_{i,r} \leq \min_{j=1,\dots,m} \{R_{j,r}^a\} \quad r = 1, \dots, R, \\ & U_i = \sum_{r=1}^R \lambda_r D_{i,r} < 1 \quad i = 1, \dots, K. \end{aligned} \quad (17)$$

In the case of open QNs, the function $f_r(\mathbf{D}, \vec{\lambda})$ is computed through the following well-known closed-form equation [19]:

$$f_r(\mathbf{D}, \vec{\lambda}) = R_r(\vec{\lambda}) = \sum_{i=1}^K \frac{D_{i,r}}{1 - \sum_{r=1}^R \lambda_r D_{i,r}}. \quad (18)$$

Figure 9 shows an example in which the (arrival rate, response time) pairs shown by the solid triangles were used to solve the optimization problem described in Equation (17) for an open QN. The solution to this problem yields the following set of service demands: $D_1 = 2.30$ sec, $D_2 = 1.60$ sec, $D_3 = 1.60$ sec, and $D_4 = 1.58$ sec. We then used a single class open QN model to compute the response times for the following values of the arrival rate (in tps), which were not used as input to solve the optimization problem: 0.04, 0.10, 0.14, 0.16, 0.22, 0.26, 0.30, 0.32, 0.36, 0.40, and 0.41 tps. The values of the response times predicted by the model are plotted in Figure 9 on the red curve. As the figure illustrates, the predicted values match perfectly the curve that underlies the measured points.

We now consider an example with two classes and three devices. Table 14 shows the measured arrival rates and measured response times per class for six subsequences.

Table 15 shows the matrix of service demands that resulted from solving the optimization problem above.

Fig. 9. Average response time (in sec) vs. average arrival rate (in tps). Solid triangles: measured. Red curve: predicted by a model.

Table 14. Measured Arrival Rates (in tps) and Average Response Times (in sec) per Class

Subsequence	Class 1		Class 2	
	$\lambda_{i,1}^a$ (tps)	$R_{i,1}^a$ (sec)	$\lambda_{i,1}^a$ (tps)	$R_{i,2}^a$ (sec)
S_1	0.030	3.300	0.015	2.827
S_2	0.050	3.475	0.045	2.977
S_3	0.100	3.832	0.080	3.282
S_4	0.300	5.792	0.150	4.910
S_5	0.350	7.375	0.220	6.241
S_6	0.450	13.992	0.320	11.656

Table 15. Derived Service Demands (sec)

Queue	Class 1	Class 2
CPU	1.00	0.80
Disk 1	1.25	0.80
Disk 2	0.90	1.10

Figures 10 and 11 use solid markers (triangles and squares) to show the average response times for classes 1 and 2, respectively, for the six subsequences shown in Table 14. Additionally, the unfilled markers in these figures show response time values predicted for the average arrival rates shown in Table 16 and computed by Equation (18). The two predicted values in each figure do not show a perfect continuity, because the values of the arrival rates for all classes impact all other classes through the utilization of all devices (see denominator of Equation (18)). Nevertheless, since the arrival rates for which the prediction is shown in Table 16 are within the arrival rates for two successive subsequences, one can see how the predicted values fall within the measured values. For

Fig. 10. Average response time (in sec) for class 1 vs. average arrival rate (in tps). Solid triangles: measured. Unfilled triangles: predicted by the derived model.

Fig. 11. Average response time (in sec) for class 2 vs. average arrival rate (in tps). Solid squares: measured. Unfilled squares: predicted by the derived model.

Table 16. Computed Average Response Times (in sec) per Class

Class 1		Class 2	
λ_1 (tps)	R_1 (sec)	λ_1 (tps)	R_2 (sec)
0.20	4.522	0.10	3.854
0.40	9.132	0.25	7.680

Table 17. Order Entry Workflow

no.	Workflow Step
1	/ordermgr/main
2	/ordermgr/login
3	/ordermgr/orderentry
4	/ordermgr/LookupUserLoginAndPartyDetails (2)
5	/ordermgr/LookupCustomerName (3 times)
6	/ordermgr/initorderentry
7	/ordermgr/setOrderCurrencyAgreementShipDates
8	/ordermgr/LookupProduct (4 times)
9	/ordermgr/additem (2 times)
10	/images/GZ-1001/small.png
11	/images/GZ-9290/small.png
12	/ordermgr/quickcheckout
13	/ordermgr/updateCheckoutOptions/quickcheckout
14	/ordermgr/checkout

example, the first pair of arrival rates for which predictions are shown in Table 16 is 0.2 and 0.09 tps for classes 1 and 2, respectively. In both cases, these arrival rates fall within those of subsequences S_3 and S_4 . The corresponding predicted response times also fall between the measured response times for these subsequences.

5.4 Experimental Validation

This section presents a validation of the method described here on an experimental testbed. The implementation used a number of supporting tools and technologies, including Apache OFBiz (ofbiz.apache.org) and Apache JMeter (jmeter.apache.org) for running the experiment, and LogStash (www.elastic.co) for parsing system logs. jMetal (see <http://jmetal.sourceforge.net>) was used as the optimization framework for solving the non-linear optimization problem. This framework allows one to use various optimization algorithms. After experimenting with several, MOEA was selected. MOEA is an open-source evolutionary computation library for Java that specializes in multi-objective optimization. Also an in-house implementation of the Approximate Mean Value Analysis (AMVA) algorithm (provided by Noor Bajunaid) was used.

The testbed uses Ubuntu Linux servers to host the OFBiz ERP software suite, which was installed on a Tomcat 7.0 application server, and to host the MySQL database supporting the OFBiz suite. Apache JMeter was used to record and run a test plan that simulates different groups of users performing the business function Order Entry.

Table 17 shows the Order Entry sequence of steps, called workflow heretofore, performed by the users and captured by JMeter. Most of the workflow steps are interactive, such as login (step 2) and initorderentry (step 6). Other steps are performed by the system and do not require user think time, such as image loading and search lookahead, as in steps 4, 5, and 8, where the system performs multiple AJAX (Asynchronous JavaScript And XML) requests while the user is typing a customer name or product name. Steps 4, 5, 8, and 9 are repeated multiple times for a total of 21 transactions in a workflow.

JMeter was used to run the workflow shown in Table 17 using 5, 10, 15, 20, 22, 25, 27, 30, 32, 35, 40, and 45 users, called threads heretofore, under the following conditions:

Table 18. Apache OFBiz Results

Subsequence	X_j^a (in wps)	R_j^a (in sec)	n_j
S_1 (5 users)	0.130	4.406 ± 0.145	0.575
S_2 (10 users)	0.255	4.481 ± 0.149	1.141
S_3 (15 users)	0.389	4.602 ± 0.149	1.788
S_4 (20 users)	0.516	5.234 ± 0.123	2.700
S_5 (22 users)	0.565	5.337 ± 0.141	3.016
S_6 (25 users)	0.622	5.751 ± 0.398	3.574
S_7 (27 users)	0.659	6.722 ± 0.168	4.427
S_8 (30 users)	0.738	7.193 ± 0.191	5.307
S_9 (32 users)	0.746	8.430 ± 0.225	6.286
S_{10} (35 users)	0.773	10.077 ± 0.257	7.785
S_{11} (40 users)	0.805	15.002 ± 0.372	12.076
S_{12} (45 users)	0.806	20.462 ± 0.465	16.497

- (1) Response time is defined as the time it takes to perform all 21 steps of the workflow.
- (2) JMeter's "Ramp-Up Period" is set to 30 sec, which is the time JMeter takes to ramp up to the full number of threads.
- (3) JMeter's "Delay Thread Creation Until Needed" is set to true to eliminate the overhead of starting all threads at the beginning of the experiment.
- (4) JMeter's Loop Count is set to 10, which means that, for each experiment, threads ran 10 times consecutively. For example, an experiment with 10 users means the workflow in Table 17 ran 100 times producing 2,100 transactions.
- (5) Experiment ramp-up and ramp-down times: To capture measurements during the system's steady state, the measurement start time, T1, and end time, T2, for each experiment were defined as follows:

T1 = maximum start time of all workflows for all threads.

T2 = minimum end time of all workflows for all threads.

For each of the 10 experiments, all threads that started or ended prior to T1 or after T2 were removed from the final measurements. By doing that, we eliminate all partial workflows that might skew the average response time or average throughput given the definition of response time mentioned earlier.

- (6) The think time between workflow steps is uniformly distributed between 0 and 3 seconds and is only used between interactive steps. For example, think time is added between all steps except steps 10 and 11 (image loading) and between the multiple AJAX calls of steps 5 and 8 (LookupCustomerName and LookupProduct).

To validate the service demand derivation method, the following steps were performed:

Step 1—Measurements. Perform several measurements on a system running the OFBiz ERP software suite for several subsequences to obtain the throughput and response time. The results of the measurements are reported in Table 18 for subsequences S_1 – S_{12} . Each row shows the number of clients generating requests in each subsequence. The number of workflows generated in each subsequence varied from 42 to 362, excluding the ramp-up and ramp-down times (the actual total workflows varied from 50 to 450). Little's Law [17] was used to obtain the concurrency level n_j for each subsequence S_j as the product of the measured throughput and the measured response time (i.e., $n_j = X_j^a \times R_j^a$). Figure 12

Fig. 12. Average response time (in sec) vs. concurrency level.

Fig. 13. Average throughput (in wps) vs. concurrency level.

shows the average response time per workflow execution and Figure 13 shows the average throughput in workflows per second (wps).

Step 2—Estimating Service Demands. Apply the optimization technique described in the previous subsections to estimate the service demands for the closed QN model that represents the experimental testbed. In this example, a closed QN model was built with three queues representing the web server, the application server, and the database server. The service demands derived by solving the optimization model, using as inputs the data for subsequence S_1 , S_2 , S_3 , S_6 , and S_{10} are 0.937, 1.114, and 1.002 sec, respectively. Because the concurrency levels for those five intervals are not integer, Approximate MVA was used, which uses the Bard-Schweitzer approximation to solve closed QNs with non-integer concurrency levels [27].

Table 19. Validation with Experimental Results Obtained Using OFBiz Order Entry Transactions

Sub-sequence	No. Users	X_j (wps)	R_j^a (sec)	n_j	X^{model} (wps)	R^{model} (sec)	% Rel. Error X_j^a	% Rel. Error R_j^a
S_1	5	0.130	4.406 ± 0.145	0.575	NA	NA	NA	NA
S_2	10	0.255	4.481 ± 0.149	1.141	NA	NA	NA	NA
S_3	15	0.389	4.602 ± 0.149	1.788	NA	NA	NA	NA
S_6	25	0.622	5.751 ± 0.398	3.574	NA	NA	NA	NA
S_{10}	35	0.773	10.077 ± 0.257	7.785	NA	NA	NA	NA
Comparison between measurements and model predicted results								
S_4	20	0.516	5.234 ± 0.123	2.700	0.56	4.80	-9%	8%
S_5	22	0.565	5.337 ± 0.141	3.016	0.59	4.84	-4%	9%
S_7	27	0.659	6.722 ± 0.168	4.427	0.67	6.58	-2%	2%
S_8	30	0.738	7.193 ± 0.191	5.307	0.71	7.49	4%	-4%
S_9	32	0.746	8.430 ± 0.225	6.286	0.74	8.51	1%	-1%
S_{11}	40	0.805	15.002 ± 0.372	12.076	0.83	14.60	-3%	3%
S_{12}	45	0.806	20.462 ± 0.465	16.497	0.85	19.32	-6%	6%

Step 3—Prediction. Using the closed QN model with the service demands obtained in step 2, compute the throughput and response time for concurrency levels not used in step 2. See the X^{model} and R^{model} columns for subsequences S_4 , S_5 , S_7 , S_8 , S_9 , S_{11} , and S_{12} in Table 19.

Step 4—Validation. Compare the response time values predicted by the model in step 3 with measured response times for the same concurrency levels. If the values are relatively close (i.e., on the order of 10% difference), then the service demand estimation process is validated and the resulting closed QN model can be considered to have predictive power. As it can be seen in the last two columns of Table 19, the percent absolute error defined as $100 \times (\text{measurement} - \text{model})/\text{measurement}$ for subsequences S_4 , S_5 , S_7 , S_8 , S_9 , S_{11} , and S_{12} is small (less than 10% and—in most cases—less than 5%), which indicates that the derived model has predictive power for concurrency levels not used in the derivation of the service demands. It is important to note that the largest errors occurred for small concurrency levels (2.7 and 3.0) due to the fact that we used AMVA, which does not provide a good approximation for low concurrency levels. However, production systems rarely operate at such low concurrency levels.

Figure 14 shows five measured response times as a function of the concurrency level according to Table 19 (see black triangles) and the seven points S_4 , S_5 , S_7 , S_8 , S_9 , S_{11} , and S_{12} (red squares). As it can be seen, the points computed by a model that uses the service demands estimated by this method match the curve obtained through measurements. Figure 15 shows the measured throughput as a function of the concurrency level according to Table 19 (see black triangles) and the seven points S_4 , S_5 , S_7 , S_8 , S_9 , S_{11} , and S_{12} (red curve) predicted by a closed QN model that uses the derived service demands resulting from the solution of the optimization problem described above. As it can be seen and as expected, the points computed by the QN model that uses the service demands estimated by this method match the curve obtained through measurements.

5.5 Discussion

Correlating log entries: In our experiments, we used timestamps along with session IDs to determine the beginning and ending transaction boundaries logged by OFBiz application server.

Fig. 14. Response time (in sec) vs. concurrency level. Solid Triangles: measured response time. Red Squares and Red Curve: predicted by a model.

Fig. 15. Throughput (in workflows per sec) vs. concurrency level. Solid Triangles: measured throughput. Red Curve: predicted by a model.

Although it is possible to rely on timestamps alone, we decided to use an additional identifier to get more accurate visit counts because of possible clock deviations among servers. Besides session IDs, one could use the IP address or any custom message produced by the application server logging software.

Correlating web server access log entries with the application server log entries: Along with the transaction timestamp, the web server access log includes a session ID for each request. We sorted the Apache (Tomcat) access log by date and IP address. For each IP address and date range in the access log, we extracted the session IDs and URLs called (which correspond to mid-tier application functions). For each user session ID, we looked in the application server log for

the following markers: when the session was created (e.g., “session created session-ID” log record) and when it was destroyed. From the point of session creation, we searched for the first time the application function (URL) was referenced in the application server log and the last time it was referenced. In addition, between the session creation and destruction, we looked for the log record that indicated “page rendered.” Rendering a page amounts to a visit from the application server to the web server.

Correlating application server log entries with database log entries: We did not need to correlate AS log entries with DS log entries, because initiating and using a database connection is recorded in the AS log itself. The transaction boundaries for the same session ID inside the AS log allowed us to count how many times JDBC connections were created and how many times the HTML page was rendered and sent back to the web server. Using these techniques, we were able to count the number of visits between WS, AS, and DS.

Log content, format and accuracy: Access logs and audit traces of production systems vary in terms of content and accuracy. Although standard software servers provide minimum default log requirements, software development frameworks allow software developers to add custom content for security and debugging purposes. We addressed this issue in detail in Reference [3], where we discussed how the default time format for the Apache web server access log affects accuracy, and we show an example in that paper. We also discussed in Reference [3] the difficulties in determining the boundaries of a user session inside a WS access log if the requests are considered individually, since it is hard to distinguish between gaps in WS requests due to think time versus the beginning of a new user session. Our approach, however, clusters web requests based on full business functions instead of individual requests. We also addressed in Reference [3] the difference between HTTP’s POST and GET methods, which would impact the accuracy of the clustering method. Requests issued using the POST method would show the exact same URL even though the parameters passed are different. In our approach (and in our experiments), we accepted that level of accuracy versus using intrusive methods to capture the URL parameters, especially since such capability would be both a security risk and also difficult to achieve in a production system.

Tradeoffs in the choice of the number of queues in a queuing network: *iModel* aims to infer system structure and model parameters by analyzing information found in the system logs. Usually, we get a black box view of the system as opposed to how many CPUs, how many cores or how many hard disks. This is why the framework treats black boxes as corresponding to servers. With additional access to the production systems, it is possible to determine the detailed hardware server specifications, for example using the Linux operating system built-in utility `lshw`.

Simplifying assumptions: We discussed in Reference [3] some simplifying assumptions when using our approach. For example, the same user is not allowed to work on multiple business functions simultaneously (for example, accounting and human resources applications), and a user can only use one computer to accomplish a single business function. In Reference [4], we discussed the minimum logging requirements for the automation process to be successful. Besides the accurate event time requirement, we mention the need to log the name of the high-level module or business function being executed in the application server. Alternatively, recording the application server URL request as requested by the web server would suffice. In our experiments, we used a system that has a few more attributes logged, such as the session ID and application server business function response times.

6 RELATED WORK

Becker et al. present the Palladio Component Model (PCM) in Reference [7] to aid in the performance prediction of component-based software systems. PCM and its supporting tool is aimed at software design time and requires some level of internal knowledge (e.g., software architecture,

service demands, and allocation of components to resource environments). In contrast, our approach deals with the problem of deriving a performance model from an operational system for which very little is known.

Some of the prior work that addressed the parameterization of analytic models includes References [8–10], where the problem of estimating unknown model parameters is treated as an optimization problem that is solved using derivative-free optimization. The objective function to be optimized is based on the distance between the observed measurements and the corresponding points derived from the model. The authors pointed out that the main problem is determining how to couple these two sets of points to arrive at an objective function to be minimized. The proposed approach is applied only to a small set of single-queue models, whereas the approach presented in this article applies to single and multiclass open and closed QNs. In Reference [29], the authors presented a survey of service demand estimation techniques, such as Kalman filter, optimization, machine learning, and linear regression. The authors introduced a classification scheme based on the input parameters, output metrics and the ability of the estimation technique to tolerate input data anomalies. Another parameter estimation method is Maximum Likelihood Estimation (MLE), a classic optimization method that uses a sample data set and an analytic function and calculates unknown variables by maximizing the probability of observing that particular sample data set [15]. MLE was shown to have relatively high computational and space requirements, by a factor of 10–100 when compared to other estimation methods [29]. This would make MLE infeasible to use in production systems, which are the primary target of our automatic model derivation framework.

Menascé considered model parameterization of open QNs when some input parameters are already known [22]. The author derived a closed-form solution when a single service demand is unknown, and a constrained non-linear optimization solution when a feasible set of service demands are unknown. However, that work did not propose a solution when none of the service demands are known a priori or for closed QNs.

In Reference [14], the authors presented a survey of performance modeling approaches focusing mainly on business information systems. They described the general activities involved in workload characterization, especially estimating service demands, and the various methods and approaches used to estimate it. Some of these methods include general optimization techniques, linear regression, and Kalman filters. The work in References [16, 33, 34] addressed the problem of estimating model parameters in highly dynamic autonomic environments in which Service Level Agreements (SLAs) (in the form of QoS) have to be maintained while offering optimal use of data center resources. The authors proposed the use of a model-based estimator based on extended Kalman filters, where the current state depends on prior knowledge of previous states. The approach presented in this article, however, relies on solving an optimization problem where only the current input and output values and system structure are known. Woodside [32] discusses fitting performance models to data as it becomes available so that models can adapt to new situations that may arise. That paper summarizes the fitting process using standard tools of nonlinear regression analysis, but does not present a complete framework for automatically generating performance models from operational logs and configuration files and does not include any validation with real systems.

In Reference [24], the authors presented three service demand estimation methods (RPS, MLPS and MINPS) for multi-threaded applications. RPS is used in single-processor systems, MLPS is used in multi-processor systems and MINPS consolidates both methods. Both RPS and MLPS methods over-estimate the service demand in the type of workload they handle. MINPS runs both methods and chooses the method that produces the smaller estimated mean service time. The work in Reference [6] used Kalman filters to estimate resource service demands for the purpose of system performance testing. The authors attempted to find the workload mix that would

eventually saturate a certain system resource in a test environment to determine the system's bottlenecks. Apte et al. [1] discuss AutoPerf, an automated load testing and resource demand profiling tool for multi-tier Internet applications. Differently from *iModel*, which is non-intrusive, AutoPerf requires the execution of a profiler daemon in each host. Moreover, that work does not provide any validation that AutoPerf's profiling component has predictive power.

In Reference [30], an agent-based reference architecture is introduced for performance model extraction in virtualized environments. Software agents have specialized functions, such as system monitoring and model parameterization and constantly communicate with a model repository.

7 CONCLUDING REMARKS AND FUTURE WORK

This article presented a framework, called *iModel*, for automatically deriving and parameterizing performance models. Analytic performance models consist of a workload model and a system model. *iModel* uses system logs and configuration files to generate a high level characterization of the system; e.g., open queuing network (QN) model versus closed QN model. Then, it builds an approximate workload model in the form of a Customer Behavior Model Graph (CBMG). *iModel* includes an XML repository of performance models and their solutions. Our framework selects the appropriate workload model and system model from the model repository and combines them to arrive at a candidate model for the application system. The candidate model is parameterized by estimating the per-class service demands by solving a non-linear optimization problem. The results obtained here, for single queues, open and closed multiclass QNs, show that the candidate model is relatively robust and has predictive power over a range of input values.

In terms of future work, some aspects of *iModel*, such as its algorithms and heuristics, can be improved upon or expanded. One essential improvement on *iModel*'s implementation is the introduction of a more generalized configuration and log file parsing language. In this article, *iModel* used Logstash, which utilizes Grok filters, which consist of regular expressions to describe file contents. A generalized parsing language could be a context free grammar (CFG) used to precisely describe log and configuration file syntax. Tokenizers and parsers can be generated from the CFG using well-known softwares, such as lex and Yacc. The CFG syntax could be provided as input to *iModel*'s implementation workflow to allow for a multitude of log and configuration file formats.

Another use case that can be explored is the use of *iModel* in virtualized cloud platforms. Some *iModel* workflow steps can be simplified by the cloud platform's telemetry data collection services and service function chaining [12, 31].

REFERENCES

- [1] Varsha Apte, T. V. S. Viswanath, Devidas Gawali, Akhilesh Kommireddy, and Anshul Gupta. 2017. AutoPerf: Automated load testing and resource usage profiling of multi-tier Internet applications. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE'17)*. ACM, New York, NY, 115–126.
- [2] Mahmoud Awad and Daniel A. Menascé. 2014. On the predictive properties of performance models derived through input-output relationships. In *Computer Performance Engineering (EPEW'14)*, András Horváth and Katinka Wolter (Eds.). Lecture Notes in Computer Science, Vol. 8721. Springer International Publishing, Berlin, 89–103.
- [3] Mahmoud Awad and Daniel A. Menascé. 2015. Automatic workload characterization using system log analysis. In *Proceedings of the CMG Conference on Performance and Capacity*. Computer Measurement Group, San Antonio, TX, November 2-5, 2015.
- [4] Mahmoud Awad and Daniel A. Menascé. 2016. Performance model derivation of operational systems through log analysis. In *Proceedings of the IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'16)*, November 7-10, 2016, La Jolla, CA. DOI: [10.1109/MASCOTS.2016.41](https://doi.org/10.1109/MASCOTS.2016.41)
- [5] Mahmoud Awad and Daniel A. Menascé. 2017. Deriving parameters for open and closed QN models of operational systems through black box optimization. In *Proceedings of the 8th ACM/SPEC Intl. Conf. Performance Engineering (ICPE'17)*. ACM, New York, NY, 127–138.
- [6] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. 2011. Autonomic load-testing framework. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC'11)*. ACM, New York, NY, 91–100.

- [7] Steffen Becker, Heiko Kozirolek, and Ralf Reussner. 2009. The Palladio component model for model-driven performance prediction. *J. Syst. Softw.* 82, 1 (2009), 3–22. Special Issue: Software Performance—Modeling and Analysis.
- [8] T. Begin, B. Baynat, F. Sourd, and A. Brandwajn. 2010. A DFO technique to calibrate queueing models. *Comput. Operat. Res.* 37, 2 (2010), 273–281.
- [9] Thomas Begin, Alexandre Brandwajn, Bruno Baynat, Bernd E. Wolfinger, and Serge Fdida. 2007. High-level approach to modeling of observed system behavior. *SIGMETRICS Perform. Eval. Rev.* 35, 3 (Dec. 2007), 34–36.
- [10] Thomas Begin, Alexandre Brandwajn, Bruno Baynat, Bernd E. Wolfinger, and Serge Fdida. 2007. Towards an automatic modeling tool for observed system behavior. In *Formal Methods and Stochastic Models for Performance Evaluation*, Katinka Wolter (Ed.). Springer, Berlin, 200–212.
- [11] Andreas Brunnert, Christian Vögele, and Helmut Krcmar. 2013. Automatic performance model generation for Java Enterprise Edition (EE) applications. In *Computer Performance Engineering*, Maria Simonetta Balsamo, William J. Knottenbelt, and Andrea Marin (Eds.). Springer, Berlin, 74–88.
- [12] M. S. Castanho, C. K. Dominicini, R. S. Villaca, M. Martinello, and R. N. Moises Ribeiro. 2018. PhantomSFC: A fully virtualized and agnostic service function chaining architecture. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC'18)*. 354–359.
- [13] R. Jain. 1991. *The Art of Computer Systems Performance Analysis*. John Wiley.
- [14] Samuel Kounev, Nikolaus Huber, Simon Spinner, and Fabian Brosig. 2012. Model-based techniques for performance engineering of business information systems. In *Business Modeling and Software Design*, Boris Shishkov (Ed.). Springer, Berlin, 19–37.
- [15] Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. 2009. Estimating service resource consumption from response time measurements. In *Proceedings of the 4th International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'09)*. ICST, Brussels, Belgium.
- [16] Marin Litoiu, Murray Woodside, and Tao Zheng. 2005. Hierarchical model-based autonomic control of software systems. *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1–7.
- [17] J. D. C. Little. 1961. A proof for the queueing formula: $L = \lambda W$. *Operat. Res.* 9, 3 (1961).
- [18] D. A. Menascé and V. A. F. Almeida. 2000. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*. Prentice Hall.
- [19] D. A. Menascé, V. A. F. Almeida, and L. Dowdy. 2004. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall.
- [20] D. A. Menascé, V. Almeida, R. Fonseca, and M. A. Mendes. 1999. A methodology for workload characterization of E-commerce sites. In *Proceedings of the ACM Conference on Electronic Commerce*. ACM.
- [21] D. A. Menascé. 2002. Two-level iterative queueing modeling of software contention. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. 267–276.
- [22] Daniel A. Menascé. 2008. Computing missing service demand parameters for performance models. In *Proceedings of the CMG Conference on Performance and Capacity*. Computer Measurement Group, 7–12.
- [23] OpenStack. [n.d.]. OpenStack ceilometer data collection service. Retrieved from <https://docs.openstack.org/ceilometer/latest/>.
- [24] J. F. Perez, S. Pacheco-Sanchez, and G. Casale. 2013. An offline demand estimation method for multi-threaded applications. In *Proceedings of the IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. 21–30.
- [25] M. Reiser and S. S. Lavenberg. 1980. Mean-value analysis of closed multichain queueing networks. *J. ACM* 27, 2 (Apr. 1980), 313–322.
- [26] J. A. Rolia and K. C. Sevcik. 1995. The method of layers. *IEEE Trans. Softw. Eng.* 21, 8 (Aug 1995), 689–700.
- [27] P. Schweitzer. 1979. Approximate analysis of multiclass closed networks of queues. In *Proceedings of the International Conference on Stochastic Control and Optimization*.
- [28] C. U. Smith and L. G. Williams. 1999. A performance model interchange format. *J. Syst. Softw.* 49, 1 (1999), 63–80.
- [29] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. 2015. Evaluating approaches to resource demand estimation. *Perform. Eval.* 92 (2015), 51–71.
- [30] Simon Spinner, Jürgen Walter, and Samuel Kounev. 2016. A reference architecture for online performance model extraction in virtualized environments. In *Proceedings Companion Publication for ACM/SPEC International Conference on Performance Engineering (ICPE'16 Companion)*. ACM, New York, NY, 57–62.
- [31] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling network function parallelism in NFV. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'17)*. ACM, New York, NY, 43–56.
- [32] Murray Woodside. 2008. The relationship of performance models to data. In *Performance Evaluation: Metrics, Models and Benchmarks*, Samuel Kounev, Ian Gorton, and Kai Sachs (Eds.). Springer, Berlin, 9–28.

- [33] M. Woodside, Tao Zheng, and M. Litoiu. 2005. The use of optimal filters to track parameters of performance models. In *Proceedings of the 2nd International Conference on Quantitative Evaluation of Systems (QEST'05)*. 74–83.
- [34] Tao Zheng, Jinmei Yang, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. 2005. Tracking time-varying parameters in software systems with extended Kalman filters. In *Proceedings of the Conference Centre for Advanced Studies on Collaborative Research (CASCON'05)*. IBM Press, 334–345.

Received September 2017; revised August 2019; accepted November 2019