An Offline Demand Estimation Method for Multi-Threaded Applications

Juan F. Pérez
Department of Computing
Imperial College London
London, UK
j.perez-bernal@imperial.ac.uk

Sergio Pacheco-Sanchez
SAP Research Belfast
Belfast, Northern Ireland, UK
sergio.pacheco-sanchez@sap.com

Giuliano Casale
Department of Computing
Imperial College London
London, UK
g.casale@imperial.ac.uk

***Abstract*—Parameterizing performance models for multi-threaded enterprise applications requires finding the service rates offered by worker threads to the incoming requests. Statistical inference on monitoring data is here helpful to reduce the overheads of application profiling and to infer missing information. While linear regression of utilization data is often used to estimate service rates, it suffers erratic performance and also ignores a large part of application monitoring data, e.g., response times. Yet inference from other metrics, such as response times or queue-length samples, is complicated by the dependence on scheduling policies. To address these issues, we propose novel scheduling-aware estimation approaches for multi-threaded applications based on linear regression and maximum likelihood estimators. The proposed methods estimate demands from samples of the number of requests in execution in the worker threads at the admission instant of a new request. Validation results are shown on simulated and real application datasets for systems with multi-class requests, class switching, and admission control.**

I. Introduction

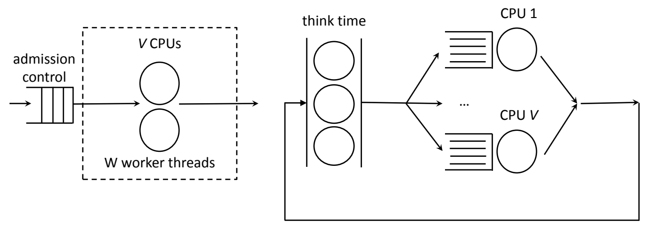
Predictive models of enterprise applications are important decision-making tools for managing the quality-of-service (QoS) offered by data center applications to end-users [1]. They are enjoying a resurgence of interest in cloud computing since deployment automation involves a significant amount of decision-making, e.g. for cost minimization [2]. For complex applications, it is however difficult to parameterize models appropriately, in particular to specify the service *demands* placed by each request on resources of the deployment environment. By service demand we refer to the total effective time a request seizes the resource, e.g. a CPU, to fulfill its requirements. Furthermore, the focus is often on modelling CPU consumption, for which existing methods mostly leverage utilization samples. However, these may not be available in all deployment environments, for example in platform-as-a-service (PaaS) clouds where the resource layer is hidden to the application and the system administrator can access only response time measurements and performance counters monitored internally to the application.

To cope with these issues, we propose two new estimation methods, RPS and MLPS, and build on top of them a combined approach, MINPS, that is able to accurately estimate the mean service demand placed on resources by a multi-threaded application. The need for advanced statistical estimation methods arises from the fact that demands describe time-varying resource consumption, which is often difficult to characterize in a deterministic fashion. For example, caching is a common source of unpredictability for request execution times [3]. More generally, demands depend on workload properties (e.g., file size distributions), on the data provided as input to the system, and on the contention at the physical resources (e.g, cache hit ratios, number of concurrent requests, memory contention, etc.). This complexity poses challenging estimation problems and has thus lead to a significant increase of the number of papers on demand estimation in the last five years [4]–[14].

The method we propose differs from existing approaches in that, to the best of our knowledge, it is the first one to apply probabilistic descriptions in estimation problems for multi-threaded applications. For example, maximum likelihood formulations have been attempted only for simpler first-come first-served queues [9]. We show that current methods are not always well suited for the multi-threaded application considered, as these are based on utilization measures, and on analytic results valid for product-form queueing networks, which do not describe well our reference application.

In essence, our main contributions are as follows: i) we introduce RPS, a regression-based scheduling-aware algorithm, that is able to accurately estimate the mean service demand of users belonging to multiple classes, in a multi-threaded application running on a single processor; ii) to deal with the multi-processor case we introduce MLPS, a maximum-likelihood scheduling-aware demand estimation algorithm, that, relying on a Markovian description, is able to estimate the mean resource demand of a multi-threaded application deployed on multiple processors; iii) we combine RPS and MLPS into a single method, MINPS, which uses both methods to produce accurate estimates at all loads of the service demands for multi-threaded applications. The goal of the method is to parameterize processor sharing (PS) queueing models, which are useful idealizations of contention at resources such as CPUs that operate under round robin scheduling or egalitarian disciplines. Using validation on simulation data and on empirical datasets from a real enterprise application, SAP ERP, we show that the methods proposed have a performance similar to the attainable lower bound provided by a method based on complete information. This performance holds for many different setups, including a broad load range, heterogeneous service demands, and non-exponential service times.

The paper is organized as follows. In Section II we overview related work on demand estimation. In Section III we present the system model under analysis, illustrate how current methods fail to analyse this type of system, and introduce the



(a) *Server layer.* (b) *Resource layer.* Population: W workers.

Fig. 1. Layered queueing model of a multi-threaded application server

estimation methodology. Next, Section IV presents a method based on complete information, which provides us with a practical lower bound on the attainable estimation error. Sections V and VI introduce the RPS and MLPS methods, respectively, together with some initial results. In Section VII we present the MINPS method, and provide experimental results that validate its performance. Section VIII presents a case study that shows how the MLPS method behaves on a production system. This is followed by conclusions in Section IX.

II. Related work

The problem of resource demand estimation has received significant attention recently, especially due to the rise of self-adaptive systems, where resource allocation decisions are based on analytic resource models [11]. These models allow predicting the impact of allocation decisions, based on an explicit description of the resources involved and the workload to be faced. The effectiveness of these models depends heavily on the accuracy of the estimated parameters. While arrival rates are usually available or easy to obtain from server logs, resource demands are difficult to estimate as they cannot be measured directly. A number of approaches are based on regressing total CPU utilization and class throughputs to obtain class estimates of resources demands [5], [12], [13], [15]. Considering multiple classes is important since requests belonging to different classes can have very different resource demands. While these methods have gained wide acceptance, their drawback are multiple and include, for example, their suffering of multicollinearity, which can heavily affect the estimates and their confidence intervals [4], [5]. Other approaches have been put forward to overcome some of the problems of regression-based methods, including the use of Kalman filters [6], [14], [16], clustering [7], [8], and pattern recognition methods [17]. Utilization-based methods require direct resource measurements, which may not be available, such as in PaaS deployments.

Other estimation methods rely on a different set of measurements, such as class response times. In [11], for instance, the problem of estimating a set of unknown resource demands is posed as an optimization problem where the objective function is the difference between the measured response time and the same metric obtained from an analytic model when using the estimated resource demand. Similar approaches have been taken in [10], [18], where the resources are modeled as a product-form open queueing network, from which explicit expressions for the mean response times can be obtained. The methods proposed in this paper also make use of the measured response times to estimate the class resource demands. In our case, however, we face a non-product form model, including a fixed capacity region, an admission control mechanism, and the possibility of requests switching class. While these features make the model more realistic, they also pose additional difficulties to the estimation problem. In this direction, a previous work [9] considers a similar setup as ours, but assuming a single FCFS server, while we focus on a PS server, which is a much closer abstraction of the actual behavior of the CPUs. Furthermore, we also consider a multi-core architecture that has been scarcely considered in previous models, as it increases the model complexity [3].

III. Background

The reference system may be modelled as a layered queueing network, such as the one represented in Figure 1. The reference application, shown in Figure 1(a), runs on a multi-threaded server, for example an application server or a servlet container, configured with a set of W worker threads. A worker thread is an independent software processing unit capable of serving requests. Requests arrives from the outside world, either in an open or closed-loop fashion, and sit in an admission buffer until they are dispatched to a worker thread. The admission control policy is first-come first-served and work-conserving, i.e., no worker can remain idle if there is a request sitting in the admission buffer.

The resource layer that processes the requests is shown in Figure 1(b) and may represent a physical host or a virtual machine. Worker threads are assumed indistinguishable from each other and run on a single CPU out of V available. We take the simplifying assumption that the operating system dispatches active workers to CPUs to maximize the available capacity. Thus, the first $V \leq W$ workers are placed on independent CPUs in order to exploit all available CPU capacity. Further, we do not keep track of the specific CPU on which a worker executes and ignore possible CPU affinities.

The main goal of the analysis is to characterize the *service rates* μ_r by which an application processes class-r requests after admission. We refer to the service demand posed by class-r requests as D_r, and to its expected value as $E[D_r] = \mu_r^{-1}$. Notice that, as the application works in a multi-threaded fashion, at any time t, if a class-r request is being processed at a CPU together with other $n(t) - 1$ requests, it will be processed at an effective rate of $\mu_r/n(t)$. In the next section we illustrate how existing methods have difficulties in dealing with this situation.

A. Motivating Example

In this section we consider the use of the utilization-based approach introduced in [18] to estimate the resource demand for the reference system detailed in the previous section. One of the advantages of [18] is that it explicitly models the resources through a product-form queueing network. In our case, however, the network model lacks a product-form solution as it features a hard bound in the number of jobs in processing at any moment by means of an admission control system. We consider a system with $V = 1$ processor and $W = 2$ working threads, and a total number of users N that varies between 20 and 180, which allow for a broad range of load values. The users may belong to one of two classes,

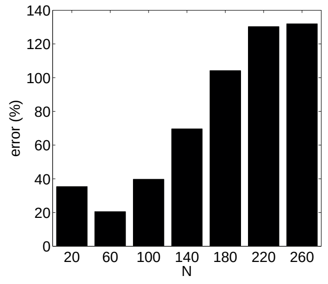


Fig. 2. Utilization-based method - Two classes - $V = 1$ - $W = 2$

each one with a different mean resource demand. With these parameters, we set up a simulation (the details of which are described later) and take samples, after a warm-up period, using sampling windows of fixed size. For each window we sample the overall server utilization, as well as the average response time and the throughput for each job class. Figure 2 illustrates the mean relative absolute estimation error (see Eq. (2)) obtained with the utilization-based method of [18]. Although the method produces reasonable estimation errors under low loads, its performance degrades significantly under medium and high loads. While this method has been shown to be very effective under different circumstances, it relies on a network representation that does not match the behavior of our reference application. In addition, the utilization measurements required by this method may not be available, as when the application in deployed on a PaaS.

B. *Estimation methodology and complete information*

The demand estimation methodology we propose requires the ability to collect a training dataset of I system state samples $\boldsymbol{n}(t_i) = (n_0(t_i), n_1(t_i), n_2(t_i), \ldots, n_R(t_i))$ at a finite sequence of instants $t_1 < t_2 < \ldots < t_I$, where

- $n_r(t_i), 1 \leq r \leq R$, represents the number of active workers serving requests of class r at time t_i
- $n(t_i) = \sum_{r=1}^{R} n_r(t_i)$ is the number of active workers.
- $n_0(t_i) = W - n(t_i)$ is the number of idle worker threads at time t_i

Throughout the paper, we assume that the instant t_i corresponds to the time an event happens in the system, i.e., a request enters or leaves a worker. We will consider two main alternatives for the collection of this information. First, we consider the *complete* information case, where the system is observed during a monitoring period of length T, along which all samples $\{\boldsymbol{n}(t_i)\}_{0 \leq t_i \leq T}$ are collected, i.e, every time a request enters or leaves a worker. While this is unrealistic for most production systems, due to the overhead necessary to collect this information, it serves us to set a lower bound on the estimation error attainable with a given sample size. The second, more realistic, alternative is the one of *incomplete* information, which considers collecting a set of samples $\{(\boldsymbol{n}(t_i), r_i)\}_{i=1}^{I}$, where the instants t_i correspond to arrival times, and, together with the system state at time t_i, we also collect the response time of the i-th sampled request. Contrary to the first sample type, in this case the samples need not be consecutive and are treated as completely unrelated. Note that samples of the system state at these instants can be obtained in several ways, e.g., logs of the application server, internal logs of the application, etc.

Recall that the samples are assumed to come from an application with server layer as depicted in Figure 1(a). In addition, we consider a closed-loop topology, where users that leave a worker thread go through a think-time phase before generating a new request. The think time for class-r is exponentially distributed with rate λ_r. Further, after leaving the worker thread, and before going through the think time, a request can change its class randomly according to a discrete probability distribution. This class-switching behavior accounts for systems where users may change the type of requests they generate. We assume there are a total of N users, switching among R classes.

IV. CI: Complete Information Algorithm

We consider here the complete information case, thus the dataset includes *all* samples $\boldsymbol{n}(t_i)$ for the instants t_i at which requests arrive and depart from the system between any two instants t_1 and t_I where the system is empty, i.e., $n_0(t_1) = n_0(t_I) = 0$. Since we assume complete information, the estimation algorithm in this section, named CI, will serve to test the performance of other methods with the same number of samples, but that rely on incomplete information. In the next section, we introduce these methods and evaluate their performance by comparing them against CI.

A. *CI Algorithm Description*

In a single processor system ($V = 1$), complete information allows reconstructing exactly the sample path of the system and the individual history of each processed job. In this case, it is straightforward to determine the empirical values of the demand of each processed job. That is, consider a class-r request j, $1 \leq j \leq J$, arrived at time $t_{j,1}$ and departed at time $t_{j,I}$, such that $t_{j,i}$, $1 < i < I$, corresponds to an arrival or departure event from the system. Then the demand placed by job j is

$$d_{j,r} = \sum_{i=1}^{I-1} \frac{(t_{j,i+1} - t_{j,i})}{n(t_{j,i}^{+})}, \tag{1}$$

where $n(t_{j,i}^{+})$ refers to the state of the system just after time $t_{j,i}$. We therefore approximate the distribution F_{D_r} of the demand D_r by the empirical distribution $F_{\widetilde{D}_r}$ of the values $\{d_{j,r} \,|\, 1 \leq j \leq J\}$. As the number of samples grows asymptotically, $F_{\widetilde{D}_r}$ converges in distribution to F_{D_r}.

Since we do not assume that the state of each individual processor is tracked separately, in the multi-processor case ($V > 1$) we estimate the value $d_{j,r}$ by considering two cases. The first case occurs when the number of active workers $n(t_{j,i}^{+})$ is less than or equal to the number of processors V. In this case we assume that each of the active workers is assigned to a different processor, and therefore $d_{j,r}$ is equal to the numerator in Eq. (1), as the workers do not need to share the processors' capacity. In the second case, where $n(t_{j,i}^{+}) > V$, we assume that all processors are busy and the system can be approximated by a single super processor with V times the

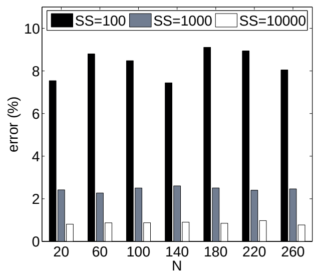


Fig. 3. CI - Two classes - $V = 2$ - $W = 8$

speed of the actual processors. The demand of job j is thus computed as

$$d_{j,r} = \sum_{i=1}^{I-1} \frac{(t_{j,i+1} - t_{j,i}) \min(n(t_{j,i}^{+}), V)}{n(t_{j,i}^{+})}.$$

B. Results

As mentioned before, the CI method allows us to set a lower bound on the estimation error attainable for a given number of samples. This is illustrated in Figure 3, where the mean estimation errors, as defined in Eq. (2), are depicted for a system with $V = 2$ processors, and $W = 8$ worker threads. We see that the error is about 8% for a sample size of 100, irrespectively of the number of users (load) in the system. The error diminishes to about 2% for 1000 samples and to less than 1% for 10000 samples.

This experiment suggests the following considerations. First, since the samples here are not affected by noise and are drawn from a model that behaves according to our assumptions, the values found are empirical lower bounds on the achievable performance of estimation methods which rely on incomplete information. Our results indicate that even with 100 samples the estimates are reasonably accurate, but better results are obtained with at least 1000 samples. It is quite interesting to note, that the CI estimation approach proposed here avoids to explicitly represent the state of each server in the multi-core case. Thus, it should be interpreted as an empirical lower bound on the achievable performance with this kind of representation. Techniques that describe the state of each station are possible, but they may require a much deeper instrumentation of the application to collect the complete information.

V. RPS: A REGRESSION-BASED APPROACH

In this section, we describe RPS, a regression-based estimation method that makes use of response times and queue lengths observed at arrival times. The method is based on the mean-value analysis [19] theory for product-form closed queueing networks. Let $\boldsymbol{K} = (K_1, \ldots, K_R)$ be the population vector of a closed queueing network, that is, its entries contain the number K_r of jobs in each class, for $r = 1, \ldots, R$. Now, for a specific processor-sharing (PS) station, let $E[R_r]$ be the expected response time for a class-r job, $E[D_r]$ its expected service demand, and $E[Q]$ the expected queue length at the station. We add $\boldsymbol{K}$ as an argument to the previous expressions to make explicit that these are for a network with population $\boldsymbol{K}$. For the single PS server case, the main result from mean-value analysis states that

$$E[R_r](\boldsymbol{K}) = E[D_r]\left(1 + E[Q](\boldsymbol{K} - \boldsymbol{e}_r)\right),$$

that is, the expected response time for a class-r job in this station, in a network with population vector $\boldsymbol{K}$, can be expressed as a function of its expected service time, and the expected queue length in this station in a network with one class-r customer less. From the arrival theorem [19] we also know that $E[Q](\boldsymbol{K} - \boldsymbol{e}_r)$ is equal to the expected number of jobs seen upon arrival (excluding itself) by a class-r customer in a network with population $\boldsymbol{K}$, referred to as $E[Q^A] \equiv E[Q^A](\boldsymbol{K})$. We therefore can write

$$E[R_r] = E[D_r]\left(1 + E[Q^A]\right),$$

and letting $E[\bar{Q}^A] = 1 + E[Q^A]$ be the expected number seen upon arrival, including the arriving job, we have

$$E[R_r] = E[D_r]E[\bar{Q}^A].$$

Therefore, to estimate the expected service demand for each class we can perform a linear regression on observations of R_r against $\bar{Q}^A$. These observations are taken from the sample $\{(\boldsymbol{n}(t_i), r_i)\}_{i=1}^{I}$ discussed in Section III-B, as r_i is the response time and $\sum_{r=1}^{R} n_r(t_i)$ the number of jobs seen upon arrival for the i-th sample.

The previous result, however, holds for a single processor only. For $V > 1$ processors we split the expected queue length equally among the processors, thus using

$$E[R_r] = E[D_r]E[\bar{Q}^A]/V,$$

as the regression equation to estimate $E[D_r]$ for each r.

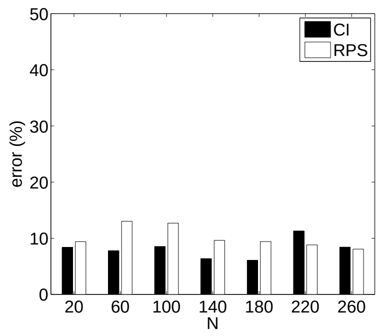
A. Results

In this section we present some results for the RPS method, which allow us to illustrate its behavior under different system setups. We generate the data by means of a discrete-event simulation, implemented in Java Modelling Tools (JMT) [20], with the closed-loop topology described in Section III-B. For each experiment we run each of the estimation methods (here CI and RPS) with the same number of samples (100 unless otherwise stated) and obtain an estimated mean service time for each class. Letting $E[D_r]$ and $\bar{D}_r$ be the actual and the estimated mean service time for class-r jobs, we compute the mean estimation error as

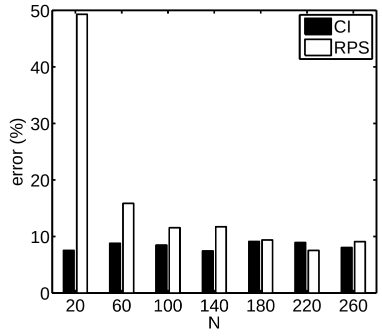
$$\text{error} = \frac{1}{R} \sum_{r=1}^{R} \frac{|E[D_r] - \bar{D}_r|}{E[D_r]}, \tag{2}$$

that is, the mean relative error among all user classes. For each system setup, we run 30 experiments and present the mean estimation error.

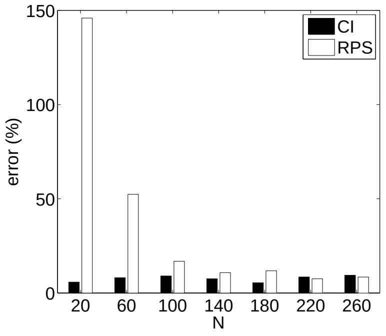
Figure 4(a) shows the error rate for the CI and the RPS methods for the case with one processor and 4 threads. The CI method provides us with a lower bound on the error attainable in this setup using 100 samples. The RPS method shows a very good behavior, comparable with that of CI, for the whole range considered for the number of users. In Figure 4(b) we consider a larger number of processors $V = 2$, while keeping the thread-to-processor ratio equal to 4, that is with 8 worker



(a) $V = 1$ - $W = 4$



(b) $V = 2$ - $W = 8$



(c) $V = 4$ - $W = 16$

Fig. 4. CI - RPS - Two classes

threads. While the CI still shows very small errors, as expected, the error for the RPS method increases significantly for low loads. This error increases for a larger number of processors, as shown in Figure 4(c), the reason being that the RPS method assumes that the queue length is split equally among the V servers, but under a small load, less than V servers are actually active, and the service rate is less than the total capacity. As a result, we have a method that can estimate the resource demand accurately for the single-processor case, as well as for a larger number processors, but only under medium to high loads. In the next section we introduce MLPS, a method able to handle the multi-processor case under low loads.

VI. MLPS: A maximum-likelihood approach

In this section we introduce MLPS, a maximum likelihood estimation method to determine the mean service demand in multi-threaded enterprise applications.

A. Maximum likelihood estimation

A maximum-likelihood estimation procedure is an optimization method that aims at finding the value of the parameters of a probabilistic model, such that the probability of obtaining a given set of samples is maximal. In the case of the MLPS method, we assume the sample set $\{(\boldsymbol{n}(t_i), r_i)\}_{i=1}^{I}$, introduced in Section III-B, is available. From this set of samples, we aim at estimating the values of the mean service time $1/\mu_r$ for each class $r = 1, \ldots, R$. These values should be such that the likelihood of obtaining the sample $\{(\boldsymbol{n}(t_i), r_i)\}_{i=1}^{I}$ is maximized. Therefore, the MLPS method is an optimization problem with objective function

$$\max_{\mu_1,\ldots,\mu_R} \mathcal{L}\left((\boldsymbol{n}(t_1), r_1), \ldots, (\boldsymbol{n}(t_I), r_I) | \mu_1, \ldots, \mu_R\right),$$

and constraints $\mu_r \geq 0$, for $r = 1, \ldots, R$, that ensure the non-negativity of the service demands. The expression $\mathcal{L}(\cdot)$ refers to the aforementioned likelihood, which can take the form of a probability mass function for discrete random variables, or a density function for continuous ones. By assuming that samples are independent and applying logarithms, the objective function can be written as

$$\max_{\mu_1,\ldots,\mu_R} \sum_{i=1}^{I} \log\left(\mathcal{L}\left((\boldsymbol{n}(t_i), r_i) | \mu_1, \ldots, \mu_R\right)\right).$$

Therefore, to solve this problem we need to define a probability model that allows us to compute $\mathcal{L}\left((\boldsymbol{n}(t_i), r_i) | \mu_1, \ldots, \mu_R\right)$, which is the topic of the next sections.

B. The approximate model

While the system model introduced in Section III is able to capture the limited threading level commonly found in enterprise applications, it is not amenable for performance evaluation. This is because, due to the fixed capacity region, the queueing network model lacks a product-form solution. Further, the number of users is typically large, making the use of direct numerical methods, based on a Markov-chain representation, computationally prohibitive. To cope with this limitation, we introduce an approximate model, that focus on the behavior inside the fixed-capacity region. The model represents the fixed-capacity region as a closed queueing network, with a total of W users, i.e., the number of users in this model equals the number of worker threads in the original model. Further, these users can be either at a PS processing node, where class-r users are served with rate μ_r, or at a delay node, where class-r users spend an exponentially distributed think time with mean $1/\bar{\lambda}_r$, for $r = 1, \ldots, R$. Notice that, while the service time in the approximate model is the same as in the original model, the mean think time is different. This change attempts to capture the dynamics of the arrivals at the processing node, without explicitly modeling the class-switching mechanism, the original delay node, nor the admission control queue.

C. The absorbing Markov chain representation

Recall that the purpose of the approximate model is to enable us to estimate the mean processing times $\boldsymbol{\mu} = \{\mu_r\}_{r=1}^{R}$ for each class, by means of a maximum-likelihood procedure. For this we need to compute the likelihood of obtaining a given sample, that is $\mathcal{L}\left((\boldsymbol{n}(t_i), r_i) | \mu_1, \ldots, \mu_R\right)$. To this end, we define an absorbing Markov chain (MC) such that the time until absorption reflects the total processing time received by an arriving job. In general, a continuous-time absorbing MC is defined by a sub-generator matrix $\boldsymbol{T}$, and an initial probability vector $\boldsymbol{\alpha}$, such that the time until absorption has probability density function

$$f(x) = \boldsymbol{\alpha} \exp(\boldsymbol{T}x)\boldsymbol{t},$$

where the exit vector $\boldsymbol{t}$ is given by $\boldsymbol{t} = -\boldsymbol{T}\mathbf{1}$, and $\mathbf{1}$ is a column vector of ones [21].

a) Sub-generator $\boldsymbol{T}$ definition: In our model, given a sample $(\boldsymbol{n}(t_i), r_i)$ for a tagged job i, the parameters $\boldsymbol{\alpha}$ and $\boldsymbol{T}$ of the absorbing MC are a function of the observed number of jobs in service $\boldsymbol{n}(t_i)$ and the service rates $\boldsymbol{\mu}$, while the absorption time is equal to the total processing time. In order to keep track of the tagged job, we extend the set of classes with a tagged class, and allow only one job, the tagged job, to belong to it. We can therefore define the state variables $X_r^k(t)$ as the number of jobs of class r in node k, for $r = 1, \ldots, R{+}1$, and $k \in \{D, S\}$, at time $t \geq 0$. Here $k = D$ stands for the delay (think time) node, while $k = S$ for the service node. The variables $\{X_r^k(t), r = 1, \ldots, R+1, k \in \{D, R\},\ t \geq 0\}$ thus take values in the set

$$\left\{ (l_1^k, \ldots, l_{R+1}^k) \,|\, k \in \{D, R\}, \sum_{k \in \{D,R\}} \sum_{r=1}^{R+1} l_r^k \leq W, l_r^k \geq 0, l_{R+1}^k \leq 1 \right\},$$

since, due to the class switching behavior, and assuming $W \leq N$, the number of jobs of each class is only limited by the threading level W. It is easy to see that the cardinality of this set is large even for small values of W and R. In fact, this representation suffers from the well-known curse of dimensionality. To cope with this problem we make one observation and a further approximation. The observation is related to the fact that we are only interested in the states where $X_{R+1}^S = 1$, i.e., where the tagged job service is still ongoing. All other states can be removed from the MC. In spite of this observation, the set of possible states is still very large, so we consider the following approximation. Let the i-th sampled (tagged) job find $n_r(t_i)$ class-r jobs in service (including the arriving job itself) at the processing node. We assume that the population of class-r jobs inside the fixed capacity region (i.e. the total population in the approximate model) is equal to $n_r(t_i)$. As the total number of requests of each class is now limited, it is enough to keep track of the number of jobs of each class in the service node. Letting $k(t_i)$ be the class of the i-th arriving job, we can describe the system with the variables $\{X_r(t), r = 1, \ldots, R\ t \geq 0\}$, taking values in the set

$$\{(l_1, \ldots, l_R) |\ 0 \leq l_r \leq n_r(t_i) - \mathbb{1}\{r = k(t_i)\}\}, \qquad (3)$$

where $\mathbb{1}$ is the indicator function that equals one when the condition in brackets holds, and is equal zero otherwise. Here $X_r(t)$ is simply the number of class-r jobs in service, without considering the tagged job, and the state space is significantly smaller than the one considered before.

After defining the state space of the MC $\{X_r(t), r = 1, \ldots, R,\ t \geq 0\}$ we now define its transition rates. Figure 5 shows the non-absorbing rates at which the MC makes a transition from a given state $(l_1, \ldots, l_R)$. These transitions consider both service completions and new arrivals to the processing node. Additionally, absorption occurs in state $(l_1, \ldots, l_R)$ with rate $\mu_{k(t_i)}/l$, where $l = \sum_{i=1}^R l_i + 1$, which corresponds to the tagged job service completion. As the service and think times are exponentially distributed, the resulting process is in fact an absorbing MC with state space given by Eq. (3).

b) Initial vector $\boldsymbol{\alpha}$ definition: From the previous description, it is clear that we define a different absorbing MC for each sample $(\boldsymbol{n}(t_i), r_i)$, as the state variables of the MC, its state space, and its transitions rates depend explicitly on $\boldsymbol{n}(t_i)$. As the service rates $\boldsymbol{\mu}$ are also explicitly used in the definition of the MC transition rates, we can define the sub-generator matrix of the MC as a function of $\boldsymbol{n}(t_i)$ and $\boldsymbol{\mu}$, that is $\boldsymbol{T}(\boldsymbol{n}(t_i), \boldsymbol{\mu})$. Now, to define the initial probability we observe that the i-th sampled job finds the processor with $\boldsymbol{n}(t_i)$ jobs, thus the MC for this sample always starts in the state corresponding to $\boldsymbol{n}(t_i)$. Therefore, the initial probability vector $\boldsymbol{\alpha}(\boldsymbol{n}(t_i))$ has a one in the entry corresponding to the state $\boldsymbol{n}(t_i)$, and zero everywhere else. In this manner we have that the likelihood of obtaining a sample $(\boldsymbol{n}(t_i), r_i)$ when the service rates are $\boldsymbol{\mu}$ can be expressed as

```
State $(l_1, \ldots, l_R)$
Total number in service $l = \sum_{r=1}^R l_r + 1$
**for** $r = 1, \ldots, R$ **do**
    **if** $l_r \geq 1$ **then**
        Destination state: $(l_1, \ldots, l_{i-1}, l_i - 1, l_{i+1}, \ldots, l_R)$
        Rate: $\mu_i / l$
    **end if**
    **if** $l_r < n_r(t_i)$ **then**
        Destination state: $(l_1, \ldots, l_{i-1}, l_i + 1, l_{i+1}, \ldots, l_R)$
        Rate: $(n_r(t_i) - l_r)\bar{\lambda}_i$
    **end if**
**end for**
```

Fig. 5. MC non-absorbing transition rates

$$f(r_i|\boldsymbol{\mu}) = \boldsymbol{\alpha}(\boldsymbol{n}(t_i)) \exp(\boldsymbol{T}(\boldsymbol{n}(t_i), \boldsymbol{\mu})x)\boldsymbol{t}(\boldsymbol{n}(t_i), \boldsymbol{\mu}).$$

This is precisely the expression we will use as the likelihood of observing a sample $(\boldsymbol{n}(t_i), r_i)$, given that the service rates are $\boldsymbol{\mu}$.

D. Load dependent modification for multiple processors

We now consider the extension of the method to multiple processors. We notice that the previous description is valid for a single processor only, and an extension to multiple processors while keeping track of the number of busy worker threads in each processor would suffer from the curse of dimensionality. We tackle this issue by modifying the transition rates in the absorbing MC in the following manner. In a state $(l_1, \ldots, l_R)$ such that $\exists i | l_i \geq 1$, and $l = \sum_{i=1}^R l_i + 1 \leq V$, i.e., when the number of jobs in process is less than or equal to the number of processors, we assume that each job is being processed by a different processor, and therefore a transition occurs to state $(l_1, \ldots, n_{i-1}, l_i - 1, l_{i+1}, \ldots, l_R)$ with rate μ_i. Instead, when $l > V$, the service completion rate of a class-i job is $V\mu_i/l$, i.e., as if the processor was a single super-processor with capacity V times that of a single processor. A similar modification is performed for transitions related to a service completion of the tagged job.

E. Estimating the think rates

With respect to the estimation of the think rates $\bar{\lambda}_i$ for each class i, we assume that, during a given monitoring period, the arrival rate to the fixed capacity region (FCR), called β_r for class-r jobs, can be estimated. As this reduces to knowing the number of arrivals in a monitoring interval, this information can be extracted from server log files. Now, from the sample $\{(\boldsymbol{n}(t_i), r_i)\}_{i=1}^I$ we compute the average number of busy threads seen upon arrival, that is $\bar{W} = \frac{1}{I}\sum_{i=1}^I n(t_i)$. Since

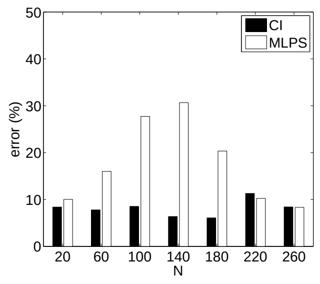


Fig. 6. CI - MLPS - Two classes - $V = 1$ - $W = 4$

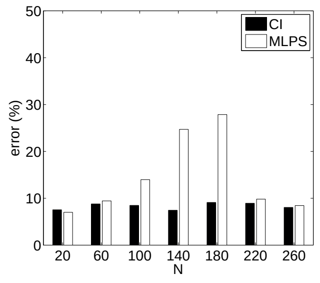


Fig. 7. CI - MLPS - Two classes - $V = 2$ - $W = 8$

$W - \bar{W}$ can be thought of an estimate of the mean number of threads undergoing a think time, we approximate the think rate as

$$\bar{\lambda}_r = \frac{\beta_r}{(W - \bar{W})/R}.$$

Here we divide the number of idle threads evenly among the different request classes. Notice that if the server is lowly loaded, the think rate λ_r is small both because β_r is small and $\bar{W}$ is close to 0. The opposite occurs under heavy loads.

F. *Results*

In this section we present some results for the MLPS method with the aim of illustrating its performance. As before, we use the mean relative absolute estimation error, as defined in Eq. (2), and consider a system with 2 job classes, $V = 1$ processor and $W = 4$ worker threads. Figure 6 shows the error rate for the CI and the MLPS methods, where a trend, that repeats among a very broad set of scenarios, arises: the MLPS provides estimates similar in accuracy to those of the CI method for low and high loads, but its accuracy diminishes for medium loads. This behavior is maintained for multiple processors, as illustrated in Figure 7, where we consider 2 processors and 8 worker threads. We have performed an exhaustive set of experiments, and found that this trend holds for a broad range of parameter values. We therefore have a method that performs well under low loads for the multi-processor case, and thus complements well the RPS introduced in the previous section. In the next section we introduce a simple method built on top of RPS and MLPS, and evaluate its behavior under a broad range of system setups.

VII. The MINPS method and Validation

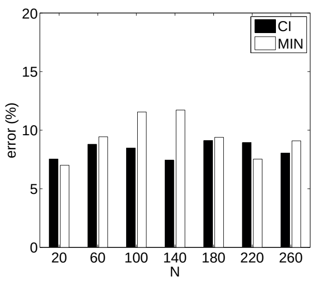
As we have shown, both RPS and MLPS are able to provide accurate estimates of the mean service demand, although only for specific regions of the load. In this section we present the MINPS method, which, although a simple extension built on top of RPS and MLPS, is able to perform similarly to CI for the whole load range considered.

After considering the results presented in the previous section, and relying on a set of training data, one could propose the following method: run both methods on the training data set and select a cut-off point for the *load* such that below this point the MLPS method is used, while above it the RPS method is selected instead. This presents two main problems: first, the term *load*, usually defined as a function of the arrival rate to service rate ratio, is not easy to define as the service rate is in fact unknown, and the arrival rate may also be unknown; second, the use of a training data set easily creates problems such as over-fitting, where the cut-off point is selected very well for the training data set, but fails for a different data set.

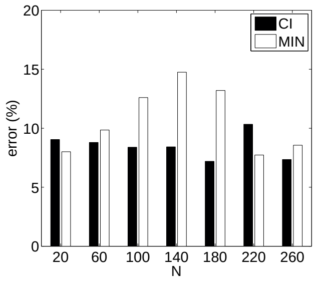
The MINPS method relies on two observations. First, as discussed in the previous section, the RPS method has a difficulty under low loads because it is incapable of treating correctly the situation where $n < V$ jobs are being processed. In these cases, not all processors are busy, and the approximation of them as a single super processor working at rate $V\mu$ becomes too rough. As a result, the RPS method will overestimate the mean service time, since the measured response times will appear too long for a system with service rate $V\mu$, while the actual service rate will be at most $n\mu$. Second, one of the main drawbacks of the MLPS method is its inability to capture arrivals that occupy the worker threads beyond the state observed upon arrival. As a result, under medium loads, there will be samples where the observed number of jobs in process is (significantly) lower than the maximum number reached before the tagged job service is completed. This will make the MLPS method over-estimate the service demand, since the response times observed have to be matched by a system where the service rate is shared among fewer jobs than in the actual system. Both observations tell us that, for the cases where the RPS and MLPS estimates have a large error, both methods are likely to fail by *over-estimating* the service demand. We therefore propose to run both methods and choose the one that offers the smaller estimated mean service demand. In the next section we show how this choice provides accurate estimates in practice.

A. *Validation*

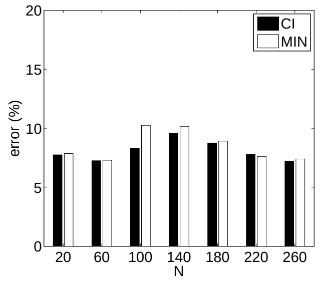
In this section we evaluate the MINPS method, built on top of MLPS and RPS, by exploring its behavior under different system setups. As summarized in Table I, we considered different values for the number of processors, the threading ratio W/V, and the number of classes. We also evaluate heterogeneous service demands, with significantly different values for the ratio μ_1/μ_2. Further, we consider different class switching behaviors, by means of the parameter α, which is the probability that a job switches class after leaving the worker thread. Finally, we evaluated scenarios with a number of users N ranging between 20 and 260. These values account for a broad range of load values, from light to heavy loads. As before, we make use of the mean absolute relative estimation error, as defined in Eq. (2), to evaluate the goodness of the estimates obtained with the proposed method.



(a) $V = 2$ - $W = 8$



(b) $V = 2$ - $W = 16$



(c) $V = 4$ - $W = 8$

Fig. 8. MINPS - Two classes

TABLE I. EXPERIMENTAL SETUP

Symbol	Parameter	Value
V	Number of processors	$\{1, 2, 4\}$
W/V	Threading ratio	$\{2, 4, 8\}$
R	Number of classes	$\{1, 2, 3\}$
μ_1/μ_2	Service rate ratio	$\{2, 10, 1000\}$
α	Class Switching Probability	$\{0.1, 0.001\}$
N	Number of users	$\{20, 60, \ldots, 220, 260\}$

We start by looking at the case considered in the previous sections, that is, with two job classes, 2 processors and 8 worker threads. As illustrated in Figure 8(a), the estimation error is similar to the one provided by the CI method, over the whole load range considered. Further, the MINPS method provides estimates that are statistically equivalent to those of the CI method for low and high loads, with a slight increase in the error under medium loads. A similar behavior holds under many different conditions, such as the others setups shown in Figure 8. Here we also observe that the method is more accurate when the threading level is small, as can be seen in Figure 8(c).

In the cases previously illustrated, the requests are allowed to switch class after leaving the service station. The switching probability α has been set to 0.1, which turns the switching probability matrix into a fast mixing one. We have also considered slow mixing cases, by letting $\alpha = 0.001$, but the estimation errors, although slightly higher than for the slow mixing case, are not significantly different from those already presented.

1) Heterogeneous service times: In the previous experiments with two user classes, the service rates were set at 20 and 40, respectively, such that the two user classes have service requirements of the same order of magnitude (ratio $\mu_2/\mu_1 = 2$). We have tested an increased differentiation in the magnitudes of the service rates, considering a ratio μ_2/μ_1 equal to 10 and 1000. Figure 9(a) shows the estimation errors for the MINPS and the CI methods where the first user class has an expected service time 1000 times as large as that of the second user class, that is $\mu_2/\mu_1 = 1000$. We observe that, despite the significant differentiation, the estimation error of the RPS method remains very similar to that of the CI method. We point out that the think times are modified such that the ratio λ_i/μ_i is fixed, and such that, in steady state, the server is loaded with many small jobs and few large jobs, with each class accounting for half the total server load.

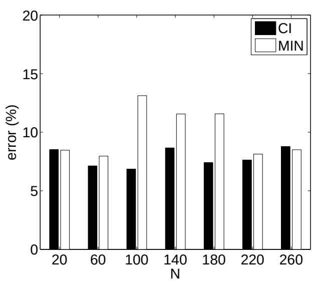
2) Non-exponential service times: As both the RPS and MLPS methods rely on the exponential assumption to estimate the mean service demands, we now consider non-exponential service times and evaluate their impact on the MINPS estimation procedure. In Figure 9(b) we present the estimation errors for the case where the service times follow an Erlang distribution with the same mean as before but with coefficient of variation equal to 0.2, instead of 1 of an exponential. Under this setup, both RPS and MLPS perform better, showing estimation errors close to those of the CI method.

On the other hand, Figure 9(c) illustrates the effect of hyper-exponential service times with coefficient of variation equal to 1.44. In this case we assume a distribution with two exponential phases, one of them with rate equal to half and the other to double the rate of the exponential distribution in the base case. While the results for both methods worsen, the estimation error of the MINPS method is similar to that of the CI method, verifying its good performance. Additional experiments have shown that further increasing the variability of the service time distribution affects the estimation errors provided by MLPS, and therefore the MINPS method. For those cases, alternative estimation methods are necessary, and can be the topic of future work.

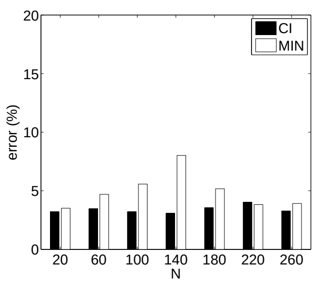
VIII. Case Study Application: SAP ERP

The software we have considered is the ERP application of SAP Business Suite [22]. The ERP application runs on top of a middleware, SAP NetWeaver [23], which defines the underlying architecture. In each experiment, we determine a fixed number of N users that issue requests for approximately 1 hour via a closed-loop workload generator with exponential think time with mean $Z_r = 10$s, for $r = 1, 2$, as the workload has two classes. Class 1 includes the following transactions: creation of a sales order document; automatic filling of sales order related fields; creation of outbound delivery with order reference; listing the sales order; changing outbound delivery related fields; saving the sales order. Class 2 includes: listing of sales orders with additional steps with the objective to fill up the cache before the loop; creation of outbound delivery; listing all sales orders for a given customer and date; creation and saving of billing document (invoice). SAP NetWeaver processes concurrently these requests, also known as *dialog steps*, which often require information retrieved from a database as depicted in Figure 10.

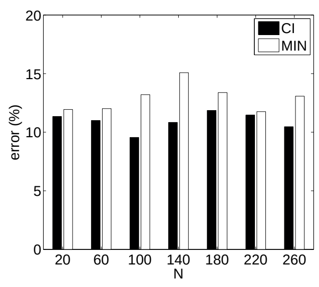
a) SAP NetWeaver: At the admission control level, SAP NetWeaver has a queue that receives the incoming requests



(a) $\mu_2/\mu_1 = 1000$



(b) Erlang service times



(c) Hyper-exponential service times

Fig. 9. MINPS - 2 classes - $V = 2$ - $W = 8$

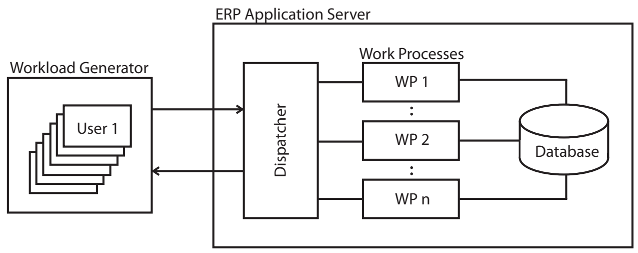


Fig. 10. *SAP NetWeaver architecture.*

which a dispatcher then forwards to software threads for processing, referred to as *work processes* (WPs). Admission occurs at the instant a WP becomes idle, based on a FCFS scheme. Dialog steps, i.e., requests, are served within a WP in a non-preemptive manner. As a result, the waiting time in the admission control queue tends to become the dominating component of the end-to-end response time as the number of active users becomes large with respect to the software threading level. In contrast, initialization activities for requests admitted to a WP imply a delay that remains constant, suggesting they are mainly due to memory-bound operations. These activities refer to the *load, generation, and roll-in* latencies, and the sum of them is denoted by Z^{lgr}. We have verified from measurements Z^{lgr} to be well approximated by an exponential distribution. The *time in work process* (D^{wp}) represents the total time required for computations in a WP, including blocking time waiting for database responses (D^{db}). Finally, the sum of the CPU consumption of each request, denoted by D^{cpu} and D^{db}, might provide an indication of the resource consumption of a request. We observe the coefficient of variation of both D^{wp} and $D^{cpu+db} = D^{cpu} + D^{db}$ to be close to unity as in an exponential distribution.

b) SAP ERP Performance Model: A performance model of the SAP ERP application is defined using the finite capacity region queueing network model shown in Figure 11. The model features an FCR that imposes a limit of W requests circulating in the stations inside the region. Thus, each job inside the FCR represents a request admitted into a WP in the real system. Arrivals to the FCR waiting buffer are regulated by the workload generator modeled as a delay station ($-/M/\infty$) with $Z_r = 10s$ for both $r = 1, 2$. Within the FCR, Z^{lgr} is modeled as a passage through a delay station, with the mean service time determined from measurements. The processing station is a multiserver $-/M/V/PS$ queue representing the WP usage of the V CPUs in a PS fashion, including the database activity.

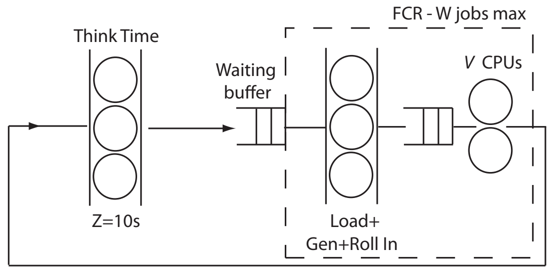


Fig. 11. *SAP ERP simulation model*

TABLE II. *SAP ERP PS model validation results. Response times are expressed in seconds.*

V	W	N_1	N_2	D_1	D_2	R_{model}	R_{meas}	U_{model}	U_{meas}
2	3	6	4	0.101	0.163	0.139	0.144	0.062	0.034
2	3	30	20	0.082	0.166	0.138	0.152	0.284	0.191
2	3	45	30	0.075	0.166	0.146	0.158	0.411	0.297
2	3	60	40	0.074	0.153	0.158	0.188	0.521	0.394
2	3	90	60	0.069	0.136	0.201	0.258	0.702	0.620
2	6	6	4	0.101	0.165	0.140	0.145	0.062	0.045
2	6	30	20	0.079	0.167	0.142	0.151	0.292	0.204
2	6	45	30	0.071	0.171	0.145	0.154	0.409	0.323
2	6	60	40	0.065	0.168	0.155	0.162	0.520	0.526
2	6	90	60	0.055	0.161	0.203	0.197	0.710	0.897

A. Results

We installed SAP ERP on a two-tier configuration composed of an application server and a database server residing on the same virtual machine, with no other virtual machines running on the same physical computer. The virtualization software installed is the VMware ESX server configured with 32 GB of memory, 230 GB of storage space, and $V = 2$ virtual CPUs each running with 2.2GHZ frequency. Each of the V servers is mapped to a separate physical CPU core. Experiments have been run with a number of work processes $W \in \{3, 6\}$ and with an increasing number of users in the range $N \in [10, 150]$, corresponding to an interval of CPU utilization $U_{meas} \in [0.03, 0.90]$. All measurements are obtained at $1ms$ resolution.

We present the end-to-end response time predictions, and compare them to the same measurements of 10 experiments on the real ERP application. For the most computational demanding estimations, we have taken the first 1000 data points for each of the datasets used; MLPS was computationally feasible in all cases up to the largest data set with 43705 points. The estimation time of MLPS grows from an average of about 1 minute for $N = 10$ to an average of 40 minutes for $N = 150$ and $W = 3$ and 87 minutes for $W = 6$. In light of this, the technique can be run periodically, but might not be practical to use to parameterize models in short periods of time, even though we have not explored the accuracy of estimates obtained by early stopping the ML optimization program. Note also that the code can be easily parallelized, thus it can leverage on multicore chipsets. Once the demand estimates are obtained, the models are solved by simulation using Java Modeling Tools.

We evaluate the estimation accuracy by the mean relative error of the estimated response times $E[R_{model}]$ with respect to the logged values $E[R_{meas}]$. Table II shows results for the simulation model, which delivers response time estimates that are in good agreement with experimental data. For $W = 3$, the best response time estimates suffer $< 9\%$ error for $U_{meas} < 30\%$. However, the last two user cases are harder to estimate, delivering errors of 16% and 22%, respectively. In contrast, for $W = 6$ all response time predictions suffer only $< 7\%$ error. Notice that the estimated decrease of service demand as the load increases is confirmed by direct measurement, but we do not have a verifiable explanation. We conjecture this to be due to fixed queue-independent delays in the demand value that get shared, and thus relatively smaller, as the number of jobs increases, e.g., overheads of software controller threads.

The utilization obtained with MLPS deviates significantly from measurement only at the highest load. We consider the latter as a minor issue since peak loads with 90% average utilization are not representative of real system behavior being applications unstable (e.g., timeouts, trashing).

IX. Conclusion

We have introduced two methods for demand estimation in multi-threaded applications: RPS and MLPS. While each of them is able to accurately provide estimates of the mean service demand for certain system setups, we combine them into MINPS, which provides accurate estimates for a broad range of parameter values. We have successfully tested the proposed methods using simulation data, as well as tested the performance of MLPS on the ERP application of SAP Business suite. The methods are well-suited for exponential and hypo-exponential demand distributions, as well as for hyper-exponential distributions with not excessively large variability. Estimating demands under high-variability is a challenge not addressed by any of the existing methods and therefore an opportunity for future work.

Acknowledgement

Sergio Pacheco-Sanchez has been co-funded by the InvestNI/SAP VIRTEX project. The research of Giuliano Casale and Juan F. Pérez leading to these results has received funding from the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement no. 318484.

References

[1] D. A. Menasce, L. W. Dowdy, V. A. F. Almeida, *Performance by Design: Computer Capacity Planning By Example*, Prentice Hall, 2004.

[2] B. Addis, D. Ardagna, B .Panicucci and L. Zhang, "Autonomic Management of Cloud Service Centers with Availability Guarantees," in *Proc. of IEEE CLOUD*. IEEE, 2010, 220–227.

[3] N. Roy, A. S. Gokhale, and L. W. Dowdy, "Impediments to analytical modeling of multi-tiered web applications," in *Proc. of MASCOTS*. IEEE, 2010, 441–443.

[4] A. Kalbasi, D. Krishnamurthy, J. Rolia, and S. Dawson, "Dec: Service demand estimation with confidence," *IEEE Trans. Software Eng.*, vol. 38, no. 3, 561–578, 2012.

[5] A. Kalbasi, D. Krishnamurthy, J. Rolia, and M. Richter, "MODE: Mix driven on-line resource demand estimation," in *Proc. of CNSM*. IEEE, 2011, 1–9.

[6] W. Wang, X. Huang, X. Qin, W. Zhang, J. Wei, and H. Zhong, "Application-level CPU consumption estimation: Towards performance isolation of multi-tenancy web applications," in *Proc. of CLOUD'12*, june 2012, 439–446.

[7] P. Cremonesi, K. Dhyani, and A. Sansottera, "Service Time Estimation with a Refinement Enhanced Hybrid Clustering Algorithm," in *Proc. of ASMTA*, LCNS vol. 6148, 291–305, Springer Berlin / Heidelberg, 2010.

[8] P. Cremonesi and A. Sansottera, "Indirect estimation of service demands in the presence of structural changes," in *Proc. of QEST*, 2012, 249–259.

[9] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson, "Estimating service resource consumption from response time measurements," in *Proc. of VALUETOOLS* , 1–10, 2009.

[10] D. Kumar, L. Zhang, and A. Tantawi, "Enhanced inferencing: estimation of a workload dependent performance model," in *Proc. of VALUETOOLS.* , 1–10, 2009.

[11] D. Menascé, "Computing missing service demand parameters for performance models," in *Proc. of CMG 2008*, 2008, 241–248.

[12] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "CPU demand for web serving: Measurement analysis and dynamic estimation," *Perf. Eval.*, 65(6-7):531–553, 2008.

[13] Q. Zhang, L. Cherkasova, and E. Smirni, "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications," in *Proc. of ICAC*, 27, 2007.

[14] T. Zheng, C. Woodside, and M. Litoiu, "Performance Model Estimation and Tracking Using Optimal Filters," *IEEE Trans. Sw. Eng.*, 34(3):391–406, May 2008.

[15] G. Casale, P. Cremonesi, and R. Turrin, "Robust workload estimation in queueing network performance models," in *PDP*. IEEE Computer Society, 2008, 183–187.

[16] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking time-varying parameters in software systems with extended Kalman filters," in *Proc. of CASCON*. IBM Press, 334–345, 2005.

[17] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload Analysis and Demand Prediction of Enterprise Data Center Applications," in *Proc. of IEEE IISWC*, 171–180, 2007.

[18] Z. Liu, L. Wynter, C. H. Xia, and F. Zhang, "Parameter inference of queueing models for IT systems using end-to-end measurements," *Perf. Eval.*, vol. 63, no. 1, 36–60, 2006.

[19] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queueing networks," *JACM*, vol. 27, 313–322, 1980.

[20] M. Bertoli, G. Casale, and G. Serazzi, "JMT: performance engineering tools for system modeling," *ACM PER*, vol. 36, 10–15, 2009.

[21] G. Latouche and V. Ramaswami, *Introduction to Matrix Analytic Methods in Stochastic Modeling*. SIAM, 1999.

[22] Enterprise resource planning (ERP). [Online]. Available: http://www.sap.com/solutions/business-suite/erp/index.epx

[23] T. Schneider, *SAP Performance Optimization Guide*. SAP Press, 2003.

[24] J. Zahorjan, *An Exact Solution Method for the General Class of Closed Separable Queueing Networks*. *Proc. of ACM SIGMETRICS*, 107-112, 1979.