Estimating Computational Requirements in Multi-Threaded Applications

Juan F. Pérez, *Member, IEEE,* Giuliano Casale, *Member, IEEE,*
and Sergio Pacheco-Sanchez, *Member, IEEE,*

Abstract—Performance models provide effective support for managing quality-of-service (QoS) and costs of enterprise applications. However, expensive high-resolution monitoring would be needed to obtain key model parameters, such as the CPU consumption of individual requests, which are thus more commonly estimated from other measures. However, current estimators are often inaccurate in accounting for scheduling in multi-threaded application servers. To cope with this problem, we propose novel linear regression and maximum likelihood estimators. Our algorithms take as inputs response time and resource queue measurements and return estimates of CPU consumption for individual request types. Results on simulated and real application datasets indicate that our algorithms provide accurate estimates and can scale effectively with the threading levels.

Index Terms—Demand estimation, Multi-threaded application servers, Application performance management

1 Introduction

Performance management of datacenter and cloud applications aims at guaranteeing quality-of-service (QoS) to end-users, while minimizing operational costs [1]. To achieve this goal, predictive models are enjoying a resurgence of interest as tools for automated decision-making at both design time and runtime [2], [3]. To successfully make use of predictive models, their parameterization is a key and non-trivial step, especially when estimating the resource requirements of requests. In performance models, this information is encoded in the *resource demand*, which is the total effective time a request seizes a resource, e.g., a CPU, to complete its execution. This parameter, or its inverse – the resource service rate, is fundamental to specify performance models such as Markov chains, queueing networks, or Petri nets, popular in software performance engineering. Estimating resource demands is also helpful to determine the maximum achievable throughput of each request type, for admission control algorithms, and to define a baseline for performance anomaly detection [4]. Further, these estimates can be used to parameterize performance models derived from high-level software specifications, such as UML MARTE [5], or PCM [6].

Although deep monitoring instrumentation could provide demand values, they typically pose unacceptably large overheads, especially when run at high-resolution. For this reason, a number of approaches [2], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16] have been proposed to obtain *mean* resource demand estimates by applying statistical methods on coarse-grained measurements. Averaging is also convenient to summarize the intrinsic uncertainty and variability that applications face due to file size distributions, variability in input data, and caching [17]. However, most of these estimation approaches rely on utilization data, which is not always available, as in, e.g., Platform-as-a-Service (PaaS) deployments where the resource layer is hidden to the application and thus protected from external monitoring.

We propose novel estimation methods that rely on *response time* and *thread-pool occupation* measurements at an application server. A key advantage of response data is that it can be efficiently obtained by active probing or by simple injection of timers in the application code. However, it is sensitive to the CPU scheduling policy and thus more challenging to analyze than utilization data. Response-based maximum likelihood formulations have been recently attempted only for simple first-come first-served queues [11], however these cannot cope with multi-threading and limited thread pools. These complex features are addressed effectively by the proposed estimation methods.

The main contributions of this paper are detailed as follows. First, we present MINPS, a scheduling-aware demand estimation method for multi-threaded applications with a *small* to *medium* threading level, e.g., below twenty threads. The MINPS method relies on two subsidiary methods: RPS, which is a regression-based method derived from mean-value analysis [18], and MLPS, a maximum-likelihood (ML) method that relies on a Markovian description of the resource consumption process. Our second contribution aims

- *J. F. Pérez and G. Casale are with the Department of Computing, Imperial College London, UK.*
  E-mail: {j.perez-bernal,g.casale}@imperial.ac.uk
- *S. Pacheco-Sanchez is with SAP HANA Cloud Computing, Systems Engineering, Belfast, UK.*
  E-mail: sergio.pacheco-sanchez@sap.com

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 (MODA-Clouds). Sergio Pacheco-Sanchez has been co-funded by the InvestNI/SAP VIRTEX and Colab projects.

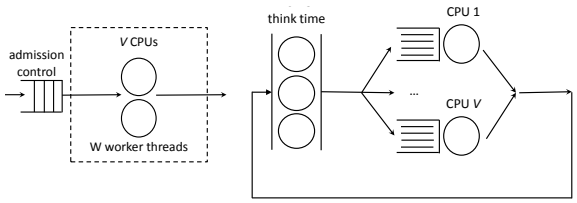
at applications with *medium* to *large* threading levels, in the order of several tens, hundreds or more, for which we introduce two alternative estimators: FMLPS, an ML approach based on fluid models, i.e., approximations of queueing models based on ordinary differential equations; and ERPS, a modified version of RPS that better accounts for parallelism in multi-core architectures. These methods complement each other, FMLPS being more accurate while ERPS being more efficient computationally. An implementation of these methods is available at https://github.com/imperial-modaclouds/modaclouds-fg-demand.

Validation results on simulation data show that our algorithms offer good performance under a broad range of scenarios, including heterogeneous and non-exponential service demands. Further, we have tested the methods using empirical datasets from two multi-threaded applications: the commercial application SAP ERP [19], which has a small threading level; and the open-source e-commerce application OFBiz, which has a large threading level.

The paper is organized as follows. Section 2 reviews related work on demand estimation. Section 3 provides background and a motivating example. Section 4 introduces an empirical estimation method that provides a baseline for the other methods. The regression-based methods RPS and ERPS are introduced in Section 5, while the ML approach MLPS, and MINPS, are presented in Section 6. The scalable ML method FMLPS is the topic of Section 7, and additional validation results for all the methods are provided in Section 8. We illustrate the use of these methods on empirical datasets in sections 9 and 10. Section 11 concludes the paper.

2 Related work

Resource demand estimation has received significant attention recently, in particular for resource management of self-adaptive systems [13]. In this context, different adaptation rules are undertaken for different request classes, where a class is a request type identified for example by a URL or by a clustering thereof. Estimation methods based on indirect measurements have gained wide acceptance, with the majority of them being based on regressing total CPU utilization and class throughputs to obtain class estimates of resource demands [7], [14], [15], [20]. Considering multiple classes is important since requests belonging to different classes may have very different resource demands. These methods, however, may suffer from multicollinearity, which can affect the estimates and their confidence intervals [2], [7]. To overcome this and other problems of regression-based methods, other approaches have been put forward, including Kalman filters [8], [16], [21], clustering [9], [10], and pattern recognition methods [22], [23]. Other works exploit CPU utilization measurements at the hypervisor level to estimate resource demands [24] and virtualization overheads [25].



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

Fig. 1. Layers in a multi-threaded server

Other estimation methods rely on a different set of measurements, i.e., response times, queue lengths, or equivalently arrival and departure time pairs for a set of requests. Previous work on inference on these metrics is limited. Queue-length measurements have been used for a Gibbs sampler [26] based on a closed multi-class product-form queueing network (QN). Queue-length data is also used in [27] to develop a maximum-likelihood (ML) method based on the diffusion limit of a single-class first-come-first-served (FCFS) multi-server queueing model. Arrival and departure times are used in [28] to estimate service times, with a Markov Chain Monte Carlo approach, for fairly general single-class QNs. In [13], unknown demands are estimated via an optimization problem where the difference between the measured and the predicted response times is minimized. Similar approaches [12], [29] model the resources as a product-form open QN, with explicit expressions for the mean response times. [30] uses utilization, response times, and arrival rates measurements to parameterize a regression model that predicts the overall average response time. Also based on response times, [11] proposes an ML demand estimation method assuming that requests are processed in an FCFS fashion. Our methods also make use of the measured response times, but we tackle the complications of estimation in multi-threaded, multi-core architectures, and in the presence of admission control and multi-class workloads. Further, our methods are able to consider applications with either small or large thread pools.

3 Background

3.1 System model

Our reference system is a multi-threaded application running on a multi-core architecture. The application server processes requests incoming from the external world, either in an open or closed-loop fashion. This reference system is conveniently modeled as the layered queueing network shown in Figure 1. Figure 1(a) details the server admission queue (a FIFO buffer), which we assume to be configured

with a pool of W worker threads. A worker thread is an independent software processing unit capable of serving requests. We assume that workers cannot remain idle if a request sits in the admission buffer. Also, we assume that the measured response times are sanitized of the time where the worker is blocked without consuming computational resources at the local server. In other words, we assume that the time consumed in calls to services deployed on external resources, where the worker is effectively blocked, has been subtracted from the sampled response time. This can be achieved by relying on application logs. In sections 9 and 10 we consider the SAP ERP and the OFBiz e-commerce applications, which have an architecture that can be mapped into this model. In both cases a set of services are exposed by the application, which is deployed on a single resource where all the services are executed. The case where the application is replicated on multiple servers, and a load balancing mechanism is applied to split the requests among them, may be handled by considering each resource separately. More complex applications, such as those that orchestrate a number of distributed services, fall beyond the reference model. However, the model may still be useful to characterize some of the individual services that compose the application.

Throughout the paper, we focus on the response times at the server, not on end-to-end delays, and restrict our attention to the dynamics inside the resource layer shown in Figure 1(b). This layer may represent a physical host or a virtual machine, with the queues modeling the V available CPUs. The queues are assumed to process jobs with a processor-sharing scheduling policy, capturing the time-sharing in the operating system. The resource layer may be abstracted by a closed network with a population of W workers placing resource demands to serve the requests, on the V identical queues. Thus, the jobs in the closed network represent workers, each one associated to an admitted request that belongs to one of R classes. The think-time server models the time that elapses between completion of a request and admission of the following one. If the admission control buffer is full at the time of a completion, this think time is zero; otherwise, it is the residual time before the arrival of the next request. 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 *user class-switching* behavior accounts for systems where users may change the request type they send, e.g., sending requests to different URLs.

Scheduling. We take the simplifying assumption that the operating system dispatches active workers to CPUs to maximize the available capacity. Thus, the V CPUs are assumed to be fully shared and the W workers are placed to exploit all the available capacity. Importantly, *we do not keep track of the specific CPU on which a worker executes*. This assumption is useful to limit the measurement complexity, since threads can frequently migrate among CPUs and high-resolution measurements would be expensive. By looking at an aggregate level, we are able to ignore these details without compromising estimation accuracy, as we show in Section 4. To simplify, we also ignore possible CPU affinities, where a subset of threads are bound to specific CPUs, although our fluid models are in principle generalizable to this case.

Demand estimation. We refer to the *expected* service demand posed by class-r requests as $E[D_r]$. The main goal of the analysis is to characterize $E[D_r]$, for each request class r, using measurements of response times inside the resource layer in Figure 1(b). This means that our response times require to measure the execution time, from request assignment to a worker thread, until completion, and excluding non-CPU related overheads such as network transmission latency. Monitoring modules simplify the task of filtering out these components from the response time measurements [31].

It is also important to remark that, by looking only at *mean* resource demands, we can equivalently map our problem into estimating the *service rates* μ_r at which an application serves class-r requests, thanks to the relation[1] $E[D_r] = \mu_r^{-1}$. However, since the application works in a multi-threaded fashion, at any time t there can be $n(t)$ concurrent requests being processed at the CPU. A class-r job will thus be processed at an effective rate of $\mu_r/n(t)$. Hence, the technical problem we need to address is how to filter out such $n(t)$ factor from the measurements. This is simple to do if all information is known, but becomes difficult when sampling, as we show in the next sections.

3.2 Estimation methodology

The demand estimation methodology we propose requires the ability to collect a 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

- $n_r(t_i)$ represents the number of busy workers serving requests of class r, $1 \leq r \leq R$, at time t_i
- $n(t_i) = \sum_{r=1}^{R} n_r(t_i)$ is the number of busy 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 consider two main alternatives for the collection of this information. First, we consider the *baseline* case, where

1. Here we assume that the ratio of the number of visits to the processing node to the number of visits to the delay node is one, which is in line with the architecture described. This however can be easily generalized to any ratio [4].

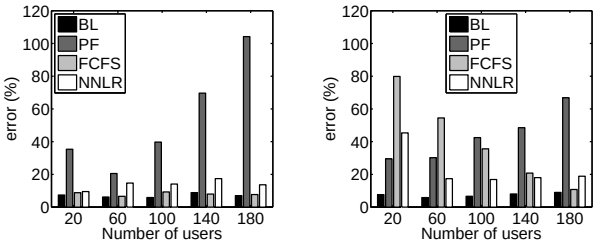
the system is observed during a monitoring period of length T, along which all samples $\{\boldsymbol{n}(t_i)\}_{0\leq t_i\leq T}$ are collected, i.e, every time a request enters or leaves a worker. While this is unrealistic for most production systems, due to the overhead necessary to collect this information, it serves us to set a lower bound on the estimation error attainable with a given sample size. It is important to stress that such estimation error will exist due to finite size of the sample, and more importantly, due to our assumption of not keeping track of the specific CPU on which a worker executes.

The second, more realistic, alternative is the one of *arrival sampling*, 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 $\boldsymbol{n}(t_i)$, we also collect the response time of the i-th sampled request. In other words, sample i is composed of the system state $\boldsymbol{n}(t_i)$ at the job arrival time, t_i, and the job response time r_i, which is recorded at the job departure time $t_i + r_i$. 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 through, e.g., server or application logs.

3.3 Motivating Example

In this section we consider the use of three existing methods, two of which require utilization measurements, and one based on response times and queue-length measurements. The method introduced in [29] relies on an *open* product-form QN model to represent the resource usage. We refer to this method as PF for product form. The system we consider, as shown in Figure 1(b), features a *closed* topology due to the thread pool and the admission control. The common method used in [15] and other works, is based on the utilization law and requires both utilization and throughput measurements to perform a non-negative linear regression. We refer to this method as NNLR for non-negative linear regression. Instead, the method in [11] relies on response time and queue length measurements, but models the resource as a single FCFS server, which may not be adequate for a multi-threaded multi-core deployment. We refer to this method as FCFS.

We consider two system setups: $V = 1$ processor and $W = 2$ working threads, and $V = 2$ processors and $W = 4$ threads. We assume a total of N external users that generate requests, and let this number vary between 20 and 180, which allows for load values between 10% and 90% approximately. The users may belong to one of two classes, each one with a different mean resource demand. With these parameters, we set up a simulation (the details of which are described later) and sample, after a warm-up period and using sampling windows of five seconds, 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 three methods mentioned above, and with the baseline (BL) method to be introduced in Section 4. For the single-processor case we observe that the NNLR and FCFS methods perform well, but increasing the number of servers degrades their performance severely. While these methods have been shown to be very effective under different setups, the multi-threaded multi-core deployment, together with the limited thread pool, pose additional challenges. Further, the utilization measurements required by two of these methods may not be available in some deployments.



(a) $V = 1$ - $W = 2$ (b) $V = 2$ - $W = 4$

Fig. 2. Various estimation methods - Two classes

4 BL: Baseline Estimation

We consider here the baseline case where 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. The estimation algorithm based on this dataset, named BL, will serve to test the performance of other methods that rely on less information.

4.1 BL Algorithm Description

In a single-processor system ($V = 1$), the baseline dataset 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 $\tau_{j,1} = t_l$ and departed at time $\tau_{j,L} = t_{l+L-1}$, where $L-2$ events occur between its arrival and departure at times $\tau_{j,i} = t_{l+i-1}$, $1 < i < L$. These events correspond to the arrival and departure of other jobs and are recorded in the baseline dataset. Then the demand placed by job j is

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

where $n(\tau_{j,i}^{+})$ refers to the state of the system just after time $\tau_{j,i}$. We therefore approximate the mean class-r

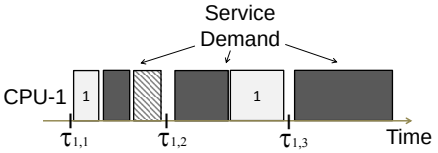


Fig. 3. Baseline estimation

demand $E[D_r]$ with the sample mean of the values $\{d_{j,r} \,|\, 1 \leq j \leq J\}$. Figure 3 illustrates (1), where a CPU starts serving three different jobs. We follow the light-gray job, marked with a one, which runs for two periods. The first period concludes when the job with the dashed background leaves the server, while the second begins at this instant and concludes when the light-gray job completes service. The length of the first period is $\tau_{1,2}-\tau_{1,1}$, while the second period lasts $\tau_{1,3}-\tau_{1,2}$. We have thus the two terms of the sum in (1), which give us the demand posed by the marked job as $(\tau_{1,2}-\tau_{1,1})/3+(\tau_{1,3}-\tau_{1,2})/2$. The BL method also has the following property, the proof of which is provided in the supplemental material.

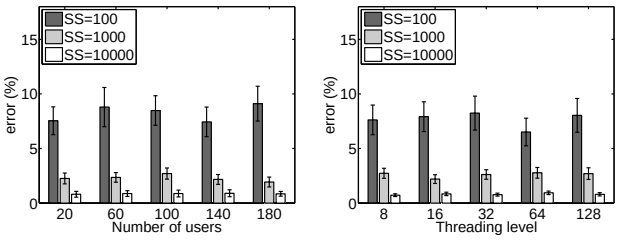
Proposition 1: When the processing times follow an exponential distribution, the BL estimator is efficient, i.e., it is unbiased and achieves the Cramer-Rao bound.

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

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

4.2 Results

Figure 4(a) reports the estimation error obtained with the BL method for a system with $V = 2$ processors and $W = 8$ threads, using different sample sizes (SS). In this and the upcoming experiments, we run each of the estimation methods with the same number of samples (100 per request class unless otherwise stated) and obtain an estimated mean service time for each class. Letting $E[D_r]$ and $\bar{D}_{r,k}$ be the actual and the estimated mean service time for class-r jobs in experiment k, respectively, we obtain the absolute



(a) $V = 2$ - $W = 8$ (b) $V = 4$ - $N = 100$

Fig. 4. BL - Two classes

relative error as

$$\text{error}_{r,k} = \frac{|E[D_r] - \bar{D}_{r,k}|}{E[D_r]}. \quad (2)$$

For each system setup, we run 30 experiments and present the mean estimation error and its 95% confidence interval using the samples of $error_{r,k}$ from all the request classes. In Figure 4 we observe that the error is on average 8.34% for a sample size of 100, with small variations depending on the number of users (load) in the system. The error diminishes to 2.45% on average for 1000 samples, and to 0.86% on average for 10000 samples. We have also observed that the estimates obtained with the BL method are not significantly affected by the threading level W, as depicted in Figure 4(b), nor by the number of processors V.

These results suggest the following considerations. First, since the samples 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 that rely on arrival sampling. It is however possible that other estimation methods can provide a smaller estimation error than BL for specific sample sets. Our results indicate that even with 100 samples the estimates are reasonably accurate, but better results are obtained with more samples. It is interesting to note that the BL estimation approach 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. Describing the state of each server is possible, but poses additional challenges in terms of both the instrumentation needed to collect the data, and the modeling required to track the state of each processor separately.

5 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. RPS is based on the mean-value analysis [18] theory for product-form closed queueing networks (QNs). Let $\boldsymbol{N} = (N_1, \ldots, N_R)$ be the population vector of a

closed QN, where N_r is the number of class-r jobs. For a processor-sharing (PS) station, let $E[R_r]$ be the expected response time of a class-r job, $E[D_r]$ its expected service demand, and $E[Q]$ the expected total queue length at the station. We add $\boldsymbol{N}$ as an argument to the previous expressions to make explicit that these are for a network with population $\boldsymbol{N}$. Also, let $\boldsymbol{e}_r$ be a zero vector with a 1 in the r-th position. For the single PS server case, the main result from mean-value analysis states that

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

i.e., the expected response time for a class-r job in this station, in a network with population $\boldsymbol{N}$, 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 [18] we also know that $E[Q](\boldsymbol{N}-\boldsymbol{e}_r)$ is equal to the expected number of jobs seen upon admission by a class-r job (excluding itself) in a network with population $\boldsymbol{N}$, referred to as $E[Q^A] \equiv E[Q^A](\boldsymbol{N})$. We thus obtain $E[R_r] = E[D_r]\left(1 + E[Q^A]\right)$. Letting $E[\bar{Q}^A] = 1+E[Q^A]$ be the expected number seen upon admission, including the admitted job, we have

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

Thus to estimate $E[D_r]$ we 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 3.2, as r_i is the response time and $n(t_i)$ the number of jobs seen upon admission by the i-th sample.

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

$$E[R_r] = E[D_r]E[\bar{Q}^A]/V, \tag{3}$$

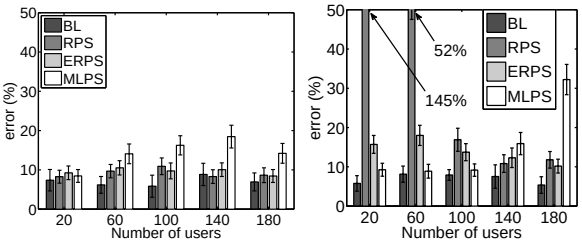
becomes the regression equation to estimate $E[D_r]$.

5.1 ERPS: An extended RPS

We now introduce a small modification to the RPS method that consists of replacing the term V in Eq. (3) by an estimate of the number of busy servers $\widetilde{V}$. The purpose of this is to attempt to solve the main drawback of the RPS method, which, as illustrated in the next section, is its inability to capture the behavior of the system under low loads. We estimate $\widetilde{V}$ as

$$\widetilde{V} = \min\left\{V, \frac{1}{I}\sum_{i=1}^{I} n(t_i)\right\}.$$

We refer to this method as Extended RPS (ERPS).



(a) $V = 1$ - $W = 2$ (b) $V = 4$ - $W = 16$

Fig. 5. BL - RPS - ERPS - Two classes

5.2 Results

We now illustrate the behavior of the RPS and ERPS methods under different system setups. We generate the data by means of a discrete-event simulation, implemented in Java Modelling Tools (JMT) [32], with the closed-loop topology described in Section 3.2, where the V CPUs are assumed to be fully shared among the workers.

We point out that the RPS method, as it relies on regression analysis, is computationally inexpensive and can be executed in less than one second in a standard desktop. Figure 5(a) shows the error rate for the BL, RPS and ERPS methods for the case with one processor and 2 threads. Both RPS and ERPS show a very good behavior, comparable with that of BL, for the whole range considered for the number of users. In Figure 5(b) we consider a larger number of processors $V = 4$, and $W = 16$ worker threads. The error for the RPS method increases significantly for low loads, reaching 145% and 52% error for 20 and 60 users respectively. This behavior, which worsens for a larger number of processors, is caused by the assumption 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. ERPS shows a better behavior, thanks to the correction introduced by splitting the queue length among the mean number of busy servers $\tilde{V}$. The error of ERPS is however twice that of BL under low loads, a trend sustained in a broad range of multi-processor scenarios. Thus, RPS and ERPS estimate the resource demand accurately for the single-processor case, and for a larger number of processors under medium to high loads.

6 MLPS: AN ML APPROACH

In this section we introduce MLPS, a maximum likelihood (ML) estimation method well-suited for the multi-processor case under low loads.

6.1 Maximum Likelihood

An ML estimation procedure is an optimization method that aims at finding the value of the parameters of a probabilistic model, such that the likelihood

$\mathcal{L}$ of obtaining a given set of samples is maximal. For MLPS, we assume the sample set $\{(\boldsymbol{n}(t_i), r_i)\}_{i=1}^{I}$, introduced in Section 3.2, is available. From this sample set, MLPS estimates the mean service demand $E[D_r]$ for each class r.

The MLPS method seeks a solution to the optimization problem with objective function

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

where the service rates μ_r are the decision variables. Recall that finding the rates μ_r is equivalent to finding the mean demands since $E[D_r] = \mu_r^{-1}$. We prefer in MLPS to operate with rates, instead of demands, because the likelihood function leverages on a Markov model parameterized by the rates $\boldsymbol{\mu} = [\mu_1,\ldots,\mu_R]$. For feasibility we add to this likelihood problem the constraints $\mu_r \geq 0$, for $r = 1,\ldots,R$.

Assuming independent samples and applying logarithms, the objective function can be written as

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

Notice that, although the response times are not independent, the effect of correlation is reduced if the samples are randomly selected or well spaced in time. The latter can be achieved, for instance, by collecting the response times of jobs whose execution do not overlap or that belong to different busy periods. For instance, considering the scenarios that will be later introduced in Section 8, and with sets of 200 samples, we observe a mean lag-1 auto-correlation of the response times of 0.091. If the samples are taken only from non-overlapping jobs, the mean auto-correlation reduces to 0.065, while randomly sampling the response times is also effective as it reduces the mean auto-correlation to 0.054. Further, if samples are taken from different busy periods the mean auto-correlation reduces to 0.053. The next sections define a Markov model to compute $\mathcal{L}\left(r_i|\boldsymbol{n}(t_i),\boldsymbol{\mu}\right)$.

6.2 The approximate model

To capture all the features of the multi-threaded application under consideration, the ideal model should consider the two layers depicted in Figure 1. A Markov model of this system, however, is infeasible since it would require a very large state space, limiting the scenarios that could be analyzed. To cope with this limitation, we introduce an approximate model that focuses only on the behavior after admission, as depicted in Figure 1(b). The model is a closed QN, with a total of W circulating jobs. Each job represents a worker thread contending for CPU capacity. 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 a think time with mean λ_r^{-1}, for $r = 1,\ldots,R$. This think time captures the time between a service completion and the admission of a new request. Both processing and think times are assumed to be exponentially distributed to obtain the Markov-chain (MC) representation described in the next section. In Section 8 we show that the MLPS method provides good estimation accuracy also under hypo-exponential processing times. The hyper-exponential case proves harder, though we have obtained good results for cases with limited variability.

6.3 The absorbing Markov chain representation

To compute the likelihood $\mathcal{L}\left((\boldsymbol{n}(t_i), r_i)|\boldsymbol{\mu}\right)$ of obtaining a given sample, we define an *absorbing* MC such that the time to absorption reflects the total processing time received by the i-th sampled job. Consider an MC with $m+1$ states such that its generator matrix can be written as

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{T} & \boldsymbol{t} \\ 0 & 0 \end{bmatrix},$$

where $\boldsymbol{T}$ holds the transition rates among the first m states, and is called a sub-generator matrix. The vector $\boldsymbol{t} = -\boldsymbol{T}\boldsymbol{e}$, with $\boldsymbol{e}$ a column vector of ones, holds the transition rates from the first m states to state $m+1$. Since the rates out of state $m+1$ are zero, once the chain visits this state it stays there forever, making it an absorbing state, and all the others transient. Similarly, the initial state in this MC is selected according to the *row* probability vector $[\boldsymbol{\alpha}\ \alpha_0]$, where the i-th entry of $\boldsymbol{\alpha}$ holds the probability that the MC starts in transient state i, and $\alpha_0 = 1 - \boldsymbol{\alpha}\boldsymbol{e}$. Starting from a transient state, the PDF of the time to absorption in state $m+1$ is [33]

$$f(x) = \boldsymbol{\alpha}\exp(\boldsymbol{T}x)\boldsymbol{t}. \tag{4}$$

Sub-generator $\boldsymbol{T}$: 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 r_i. To keep track of the tagged job, we extend the set of classes with a tagged class, and allow only the tagged job to belong to it. To describe this model, the state space of the MC needs to consider all possible combinations of W jobs in $R+1$ different classes and two nodes. This number is large even for small values of W, making infeasible the computation of the matrix-exponential in (4). To cope with this problem we introduce the following key assumption. We assume that the *total* population of class-r jobs (threads) is equal to the one observed by the tagged job upon admission, i.e., $n_r(t_i)$.

As the number of jobs in each class is now limited, it is enough to keep track of the number of jobs of each class in the service node. Let $k(t_i)$ be the class of the i-th admitted job, and let $X_r(t)$ be the number of class-r jobs in service, without considering the tagged job.

Require: State $\boldsymbol{l} = (l_1, \ldots, l_R)$, $l = \sum_{r=1}^{R} l_r + 1$
  for $r = 1, \ldots, R$ **do**
    Service completion
    $\boldsymbol{T}(\boldsymbol{l}, \boldsymbol{l} - \boldsymbol{e}_r) = \mu_r l_r / l$
    Think time completion
    $\boldsymbol{T}(\boldsymbol{l}, \boldsymbol{l} + \boldsymbol{e}_r) = (n_r(t_i) - l_r)\lambda_r$
  end for

Fig. 6. MC non-absorbing transition rates

We can thus describe the system with the variables $\{X_r(t), r = 1, \ldots, R,\ t \geq 0\}$, taking values in the set

$$\{(l_1, \ldots, l_R) |\ 0 \leq l_r \leq n_r(t_i) - \mathbb{1}\{r = k(t_i)\}\}, \quad (5)$$

where $\mathbb{1}$ is the indicator function.

The non-absorbing rates of the MC $\{X_r(t), r = 1, \ldots, R,\ t \geq 0\}$ are shown in Figure 6, including both arrivals and service completions at the processing node. Additionally, absorption occurs in state $(l_1, \ldots, l_R)$ with rate $\mu_{k(t_i)}/l$, $l = \sum_{i=1}^{R} l_i + 1$, corresponding to the tagged job service completion. The non-absorbing rates compose the sub-generator matrix $\boldsymbol{T}(\boldsymbol{n}(t_i), \boldsymbol{\mu})$ associated with sample $(\boldsymbol{n}(t_i), r_i)$.

Initial distribution $\boldsymbol{\alpha}$: To define the initial probability distribution $\boldsymbol{\alpha}$ we observe that the i-th sampled job finds the processor with $\boldsymbol{n}(t_i)$ jobs. Thus the initial probability vector $\boldsymbol{\alpha}(\boldsymbol{n}(t_i))$ has a one in the entry corresponding to state $\boldsymbol{n}(t_i)$, and zero everywhere else. In this manner the likelihood of obtaining a sample $(\boldsymbol{n}(t_i), r_i)$ when the service rates are $\boldsymbol{\mu}$ can be expressed as

$$\mathcal{L}\left(r_i | \boldsymbol{n}(t_i), \boldsymbol{\mu}\right) = \boldsymbol{\alpha}(\boldsymbol{n}(t_i)) \exp(\boldsymbol{T}(\boldsymbol{n}(t_i), \boldsymbol{\mu}) r_i) \boldsymbol{t}(\boldsymbol{n}(t_i), \boldsymbol{\mu}),$$

where $\boldsymbol{t}(\boldsymbol{n}(t_i), \boldsymbol{\mu}) = -\boldsymbol{T}(\boldsymbol{n}(t_i), \boldsymbol{\mu})\boldsymbol{e}$.

6.4 The case of multiple processors

We now extend MLPS to consider multiple processors. As keeping track of the number of busy worker threads in each processor would suffer from the curse of dimensionality, we modify the transition rates in the absorbing MC as follows. In state $\boldsymbol{l} = (l_1, \ldots, l_R)$ such that $\exists r$ with $l_r \geq 1$, and $l = \sum_{r=1}^{R} l_r + 1 \leq V$, 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 $\boldsymbol{l} - \boldsymbol{e}_r$ with rate $l_r \mu_r$. Instead, when $l > V$, the service completion rate of a class-r job is $V \mu_r l_r / l$, as if the processor was a single "super-processor" with V times the capacity of a single processor.

6.5 Estimating the think times

To estimate the mean think times λ_r^{-1} for each class r, we assume that, during a given monitoring period, the admission rate of requests to the worker threads, called β_r for class-r jobs, can be estimated. As this reduces to knowing the number of arrivals in the 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 admission, that is $\overline{W} = \frac{1}{I} \sum_{i=1}^{I} n(t_i)$. Since $W - \overline{W}$ can be thought of as an estimate of the mean number of threads undergoing a think time, we approximate the think rate as

$$\lambda_r = \frac{\beta_r}{(W - \overline{W})/R}. \quad (6)$$

This expression is a simple application of Little's law, as $(W - \overline{W})/R$ approximates the mean number of class-r jobs in the delay node, λ_r^{-1} is the mean time spent in this node, and β_r is the effective throughput. 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 $\overline{W}$ is close to zero. The opposite occurs under heavy loads.

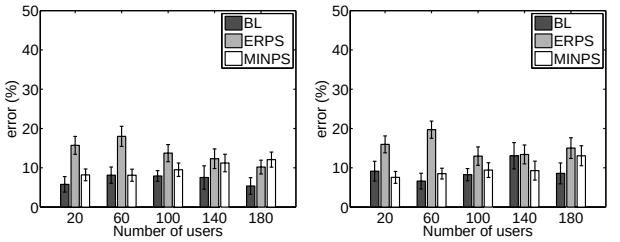
6.6 Results

We illustrate the performance of MLPS by considering, as in Section 5, a system with 2 job classes, 1 processor and 2 worker threads. We must highlight that the simulation is performed for the system as depicted in Figure 1, without considering the assumptions introduced in the derivation of MLPS. Figure 5(a) shows the error for MLPS, where a trend, repeated among a set of scenarios broader than the ones shown here, arises: MLPS provides estimates similar in accuracy to those of the BL method for low loads, but its accuracy diminishes for high loads. This behavior also holds for multiple processors, as illustrated in Figure 5(b), where we consider 4 processors and 16 worker threads. We have performed an exhaustive set of experiments, and found that this trend holds for a broad range of parameter values. MLPS thus performs well under low loads for the multi-processor case, complementing well the RPS method, which performs well under high loads but poorly under load loads.

6.7 The MINPS method

In this section we present the MINPS method, which is built on top of RPS and MLPS, and relies on two observations. First, RPS has a difficulty under low loads because it is incapable of treating correctly the situation where $n < V$ jobs are being processed. In this case, not all processors are busy, and approximating them as a single "super-processor" working at rate $V\mu$ becomes too coarse. As a result, RPS overestimates the mean service time, since the measured response times 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 drawbacks of MLPS is its inability to capture arrivals that occupy the worker threads



(a) $V = 4$ - $W = 16$ (b) $V = 4$ - $W = 32$

Fig. 7. MINPS - ERPS - Two classes

beyond the state observed upon admission. That is, the jobs observed upon admission can finish their service and return later on, but the number of jobs in each class is limited by the numbers observed upon admission. As a result, under high loads, there will be samples where the observed number of jobs in process is (much) lower than the maximum number reached before the tagged job service is completed. In this case, MLPS over-estimates the service demand, as the response times observed have to be matched by a system where the servers are shared among fewer jobs than in the actual system.

From these observations we conclude that, when the RPS and MLPS estimates have larger errors, both methods are likely to fail by *over-estimating* the service demand. In MINPS we propose to run both methods and choose the one that offers the *smaller* estimated mean service demand, thus preventing the use of an over-estimated value caused by the failures described above. Figure 7(a) shows the behavior of MINPS against BL and ERPS for a setup with 4 processors and 16 threads. While their performance is similar under high loads, MINPS has a much smaller error under low loads, similar to that of BL. This behavior is sustained for setups with a larger number of threads, as in Figure 7(b), and processors.

7 FMLPS: A SCALABLE ML APPROACH

The MLPS method described in the previous section relies on an approximation to limit the MC state-space size. However, if the number of threads or processors is large, the MC state-space size can turn the likelihood computation infeasible. In fact, the number of threads considered in the previous sections are compatible with memory-intensive applications, such as the ERP application analyzed in Section 9, where allowing a large number of parallel threads may cause memory bottlenecks. In this case, the number of parallel threads is limited and the state space of the MC underlying MLPS is tractable. In other types of applications, e.g., web sites, the number of worker threads can be significantly larger, as the memory access requirements are much lower.

To deal with applications with a large number of threads, we introduce an ML method that replaces the underlying MC in MLPS with a fluid approximation. The fluid model avoids the state-space explosion and is therefore suitable for models with a large number of threads and processors. With the fluid model, the matrix-exponential in (4) is replaced by the solution of a set of ordinary differential equations (ODEs), which is derived from the original system by a scaling argument. As with MLPS, for a given sample $(\boldsymbol{n}(t_i), r_i)$ we model the state after admission as a closed multi-class QN with two stations: a processing station and a delay station. A key difference with respect to MLPS is that, instead of assuming a total population of $n(t_i)$ workers, FMLPS assumes the correct number of W workers, where $W - n(t_i)$ are initially located at the delay station. This better captures the dynamics of the admitted jobs, but it is expensive to consider with the MC model of MLPS due to the state-space explosion.

7.1 The FMLPS QN model

We model the dynamics after admission as a QN with R classes and two stations. One station represents the super-processor obtained by aggregating the V CPUs; the other station is a think-time server to model the workers' idle time. We assume a total of W jobs and allow the jobs in the QN to change their class when transferring from the CPU to the think time server, making this model a class-switching QN[2].

We assign the admitted job to a tagged class, labeled $R{+}1$, as in MLPS. With this setup, the QN can be modeled as an MC, the state of which at time t is given by the vector $X(t) = \{X_{i,r}(t), i \in \{P, D\},\ 1 \leq r \leq R{+}1\}$, where $X_{i,r}(t)$ is the number of class-r jobs in resource i at time t, with $i = P$ (resp. $i = D$) standing for the processing (resp. delay) station. The system state is modified by events, which are limited to admissions (think time departures) and service completions (CPU departures). Once the service of a class-r job terminates in station i, a job proceeds to station j as a class-s job with probability $P_{i,j}^{r,s}$, allowing the *worker* class-switching behavior. In FMLPS, this feature is used to estimate the response time distribution, and thus the likelihood, of each sample. A service completion triggers a transition from state $\boldsymbol{x}$ to state $\boldsymbol{x}+\boldsymbol{e}_{j,s}-\boldsymbol{e}_{i,r}$, where $\boldsymbol{e}_{i,r}$ is a vector of zeros with a one in entry (i, r). Thus, the number of class-r jobs in station i decreases by one, while the number of class-s jobs in station j increases by one.

The transition rate from state $\boldsymbol{x}$ to state $\boldsymbol{x}+\boldsymbol{e}_{j,s}-\boldsymbol{e}_{i,r}$ is denoted with $f_{i,j}^{r,s}(\boldsymbol{x})$, and its definition is given in Figure 8. Notice that the transition rates associated to service completions are adjusted to consider the PS multi-core case. Thus, when at most V jobs are

2. We refer to this as *worker* class-switching, to differentiate it from the *user* class-switching described in Section 3.1. Worker class switching will be used *only* to estimate the likelihood of the samples, not to capture the user class-switching behavior. This would require a layered model and additional monitoring information to parameterize the model. This generalization is left for future work.

Require: State $\boldsymbol{x} = (x_{i,r})$, $x_i = \sum_{r=1}^{R+1} x_{i,r},\ i \in \{P, D\}$
  for $r, s = 1, \ldots, R+1$ **do**
    Service completion
    if $x_P \leq V$ **then**
      $f_{P,D}^{r,s}(\boldsymbol{x}) = \mu_r P_{P,D}^{r,s} x_{P,r}$
    else
      $f_{P,D}^{r,s}(\boldsymbol{x}) = \mu_r P_{P,D}^{r,s} V \frac{x_{P,r}}{x_P}$
    end if
    Think time completion
    $f_{D,P}^{r,s}(\boldsymbol{x}) = \lambda_r P_{D,P}^{r,s} x_{D,r}$
  end for

Fig. 8. MC transition rates with worker class switching

present at the processing station, each of the jobs is assumed to be assigned to a different processor, without contention. When more than V jobs are present, the service completion rate, as in MLPS, is that of a single super-processor with processing rate V times that of an individual processor. The transition rates associated to think time completions use the rates λ_r, estimated as in Eq. (6). As in MLPS, both processing and think times are assumed to be exponentially distributed. However, this assumption can be relaxed to consider Phase-Type (PH) [33] processing and think times, as the fluid model can better accommodate the required extended state description without suffering from the state-space explosion that affects MLPS.

7.1.1 The fluid model

As mentioned above, the MC $\{X(t), t \geq 0\}$ cannot be analyzed directly as in MLPS, due to the state space explosion. To overcome this issue we introduce a fluid approximation, i.e., an ODE formulation of the queueing model. This approximation can be shown to become exact when the number of jobs W grows asymptotically large. Such asymptotic limit is obtained as follows. We consider a sequence of QN models, indexed by k, such that when $k \to +\infty$, the sample paths of the QN models tend to the solution of a deterministic ODE system. Let $\{X_k(t)\}_{k \in \mathbb{N}_+}$ be the sequence of QN models such that $X_1(t) = X(t)$ (the QN model defined in the previous subsection), and $X_k(t)$ for $k \geq 2$ is defined as $X_1(t)$ with a total population of kW jobs and kV servers. The state space of $X_k(t)$ is thus $\{\boldsymbol{x} \in \mathbb{N}^{MR} : \sum_{i \in \{P,D\}} \sum_{r=1}^{R+1} x_{i,r} = kW\}$.

As we show in the technical report [34], the sequence $\{X_k(t)\}_{k \in \mathbb{N}_+}$ verifies the conditions of [35, Theorem 3.1], such that the sample paths of the *normalized* sequence $\{X_v(t)/v\}_{v \in \mathbb{N}_+}$ converge in probability to a deterministic ODE system. Informally, this means that the solution of an ODE system can be used to approximate the original QN, with the approximation becoming tighter as v grows large. Notice that the fluid model is derived using a scaling argument, where we consider a certain limit for increasing number of servers and jobs, but the resulting asymptotic ODE system is re-scaled to approximate the behavior of the *original* system $X_1(t)$. We now introduce the ODE system for the evaluation of the FMLPS queueing model.

7.1.2 The ODE system

The state of the ODE system $\boldsymbol{x}(t) \in \mathbb{R}^{MR}$ that describes the FMLPS QN model evolves according to

$$\frac{d\boldsymbol{x}(t)}{dt} = F(\boldsymbol{x}(t)), t \geq 0, \tag{7}$$

where, for any $\boldsymbol{x} \in \mathbb{R}^{MR}$, $F(\boldsymbol{x})$ is the drift of $X(t)$ in state $\boldsymbol{x}$, defined as

$$F(\boldsymbol{x}) = \sum_{i \in \{P,D\}} \sum_{j \in \{P,D\}} \sum_{r=1}^{R} \sum_{s=1}^{R} (\boldsymbol{e}_{j,s} - \boldsymbol{e}_{i,r}) f_{i,j}^{r,s}(\boldsymbol{x}). \tag{8}$$

The ODE initial state is partitioned as $\boldsymbol{x}_0 = [\boldsymbol{x}_0^D,\ \boldsymbol{x}_0^P]$, where $\boldsymbol{x}_0^D$ (resp. $\boldsymbol{x}_0^P$), refers to the initial number of jobs of each class undergoing a think time (resp. processing at a CPU). For sample $(\boldsymbol{n}(t_i), r_i)$, $\boldsymbol{x}_0^p$ is set as

$$\boldsymbol{x}_0^p = [n_1, \ldots, n_{k-1}, n_k - 1, n_{k+1}, \ldots, n_R, 1],$$

where k is the class of the admitted job, and we removed the parameter t_i for readability. Notice that the number of class-k jobs is reduced by one, and this job is assigned to the tagged class $R+1$. Additionally, the initial state of the delay station is set by taking the total number of jobs not present at the processing station, $W - \sum_{r=1}^{R} n_r(t_i)$, and assigning them proportionally to the effective admission rate of each class (β_r). The tagged class has zero jobs in the delay node. The initial state of the delay station is thus set to

$$\boldsymbol{x}_0^d = \left[\left(W - \sum_{r=1}^{R} n_r(t_i) \right) \frac{\boldsymbol{\beta}}{\sum_{r=1}^{R} \beta_r},\ 0 \right], \tag{9}$$

where $\boldsymbol{\beta}$ is the $R \times 1$ vector with β_r as entries. The routing matrix P is defined in the next section.

7.1.3 Response time distribution

To assess the likelihood of the observed response time r_i with the fluid model we follow a similar approach as in [36]. This approach consists of defining a transient class, and letting the jobs (workers) in this class switch to a non-transient class after processing. As in MLPS, for a class-k sample we define a tagged class $R+1$ and assign the admitted (tagged) job to this class. Next, we set $P_{P,D}^{R+1,k} = 1$, such that, when the tagged job (worker) finishes, it switches from class $R+1$ to its original class, making the class $R+1$ transient. For all the other classes, we set $P_{D,P}^{r,r} = P_{P,D}^{r,r} = 1$, thus routing the workers without any class switching.

As in MLPS, we require an expression for the likelihood $\mathcal{L}$ of obtaining a sample $(\boldsymbol{n}(t_i), r_i)$ given a set of processing rates $\boldsymbol{\mu}$, which for the fluid model we express as the response time PDF $\tilde{f}_i(\cdot)$ evaluated at r_i. To approximate this function we introduce the

variable $y(t)$, which keeps track of the station visited by the tagged job at time t, thus $y(t) \in \{D, P\}$ for $t \geq 0$. Let R_i be the random variable describing the response time of the tagged job associated with sample i, and let Φ_i be its cumulative distribution function. Since, for any $t \geq 0$, the event $\{y(t) = P\}$ is equivalent to $\{R_i > t\}$, we can state

$$\Phi_i(t) = \mathbb{P}\left(R_i \leq t\right) = 1 - \mathbb{P}\left(y(t) = P\right).$$

Further, $\mathbb{P}\{y(t) = P\}$ can be written as the expected value of the indicator function $\mathbb{1}\{y(t) = P\}$ since

$$\mathbb{E}[\mathbb{1}\{y(t) = P\}] = 1 \cdot \mathbb{P}\left(y(t) = P\right) + 0 \cdot \mathbb{P}\left(y(t) = D\right).$$

Notice that $\mathbb{1}\{y(t) = P\}$ is the number of tagged jobs in the processing station, thus being equal to $X_{P,R+1}(t)$. We can therefore rewrite $\Phi_i(t)$ as

$$\Phi_i(t) = 1 - \mathbb{E}\left[X_{P,R+1}(t)\right] \approx 1 - x_{P,R+1}(t),$$

where, as in [36], we make use of the fluid solution $\boldsymbol{x}(t)$ to approximate the expected value of $X(t)$. Using this approximation, the response time PDF $\tilde{f}_i(\cdot)$ is obtained as the derivative of $\Phi_i(\cdot)$,

$$\tilde{f}_i(t) = \frac{d\Phi_i(t)}{dt} \approx -\frac{dx_{P,R+1}(t)}{dt} = -F_{P,R+1}(\boldsymbol{x}(t)),$$

since, from (7), $F(\cdot)$ is precisely the derivative of $\boldsymbol{x}(t)$. The likelihood of sample $(\boldsymbol{n}(t_i), r_i)$ is thus given by $-F_{P,R+1}(\boldsymbol{x}(r_i))$, obtained by solving the ODE in (7) between $t = 0$ and $t = r_i$ with initial state $\boldsymbol{x}_0$.

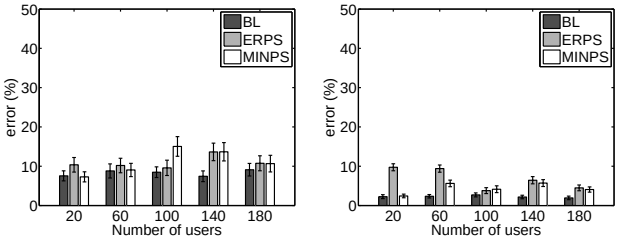
8 Validation

In this section we explore the behavior of the different methods introduced in the paper. As summarized in Table 1, we consider a broad range for the number of processors and the threading ratio W/V. We also evaluate heterogeneous service demands, by modifying the ratio μ_1/μ_2. The number of users N is set between 20 and 180. From simulations we know that for 20, 100, and 180 users, the server load is about 10%, 50%, and 90%, respectively. Recall that the simulation assumes that the V CPUs are fully shared among the workers, although we will also consider a Join the Shortest Queue (JSQ) policy for worker allocation. As before, we evaluate the estimates with the absolute relative error, defined in Eq. (2). For the cases with small (resp. large) number of servers and threads we consider MLPS (resp. FMLPS), since MLPS suffers from the state-space explosion while FMLPS is expected to be more accurate when these numbers are large due to the fluid approximation.

We also consider different user class-switching behaviors, by means of the parameter α, which is the probability that a job switches class after leaving the worker thread. In all the results to be presented we set $\alpha = 0.1$, making the switching probability matrix fast mixing. We also considered slow mixing cases, $\alpha = 0.001$, but the estimation errors, although slightly higher, are very similar to those shown here.

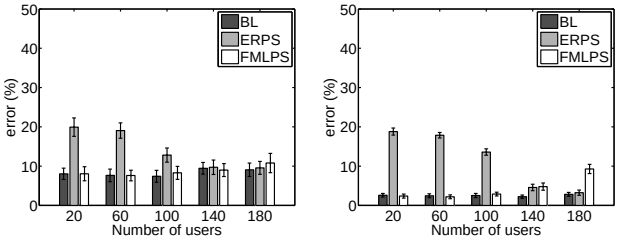
TABLE 1
Experimental setup

Symbol	Parameter	Value
V	Number of processors	$\{1, 2, 4, 8, 16\}$
W/V	Threading ratio	$\{2, 4, 8, 16, 32\}$
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	$\{20, 60, \ldots, 180\}$



(a) $SS = 100$ (b) $SS = 1000$

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



(a) $SS = 100$ (b) $SS = 1000$

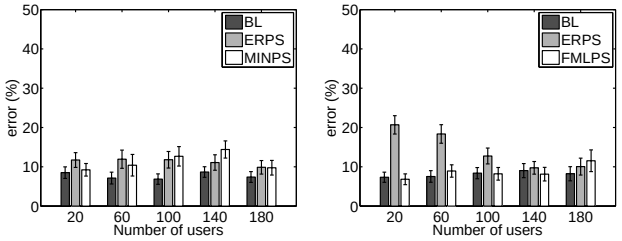
Fig. 10. FMLPS - ERPS - $V = 8$ - $W = 64$

8.1 Larger sample size

Figure 9(a) shows how MINPS and ERPS provide similar estimation errors for the case of 2 processors and 100 samples. However, increasing the sample size to 1000, as in Figure 9(b), MINPS performs similar to BL, and much better than ERPS, under low loads. This is a direct effect of the better performance of MLPS under low loads, which is inherited by MINPS. Figure 10(a) illustrates the case of $V = 8$ and $W = 64$, with a sample size of 100. FMLPS performs remarkably well, similar to BL, over the whole load range considered, while ERPS shows larger errors under low loads. This effect is amplified when increasing the sample size to 1000, as in Figure 10(b), where FMLPS performs significantly better than ERPS, particularly under low to medium loads. Under high loads the performance of FMLPS diminishes, probably due to the approximation introduced to set the number of jobs of each class initially located at the delay station, cf. Eq. (9).

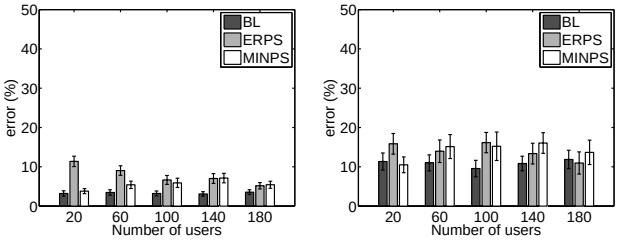
8.2 Heterogeneous service times

We now consider highly differentiated service times, setting the ratio $\mu_2/\mu_1 = 1000$, which was equal to 2 in



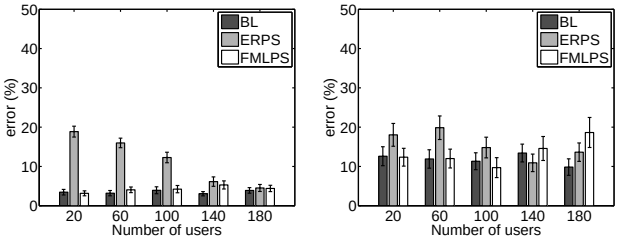
(a) $V = 2$ - $W = 8$ (b) $V = 8$ - $W = 64$

Fig. 11. $\mu_2/\mu_1 = 1000$



(a) Erlang (b) Hyper-exponential

Fig. 12. MINPS - ERPS - $V = 2$ - $W = 8$



(a) Erlang (b) Hyper-exponential

Fig. 13. FMLPS - ERPS - $V = 8$ - $W = 64$

the previous experiments. We also modify the think times, keeping the ratio λ_i/μ_i fixed, such that each class accounts for half the server load. This prevents one class from becoming insignificant with respect to the other. In Figure 11, we observe that, despite the significant differentiation, the estimation errors of all the methods remain very similar to the case $\mu_2/\mu_1 = 2$. Comparing Figures 9(a) and 11(a), we observe a minor increase for MINPS under medium loads, but the difference is not statistically significant. In the case of a large number of processors and threads, in Figures 10(a) and 11(b), we observe a similar behavior for both ERPS and FMLPS. In this and other experiments we have observed that the estimation errors are similar for the different request classes, provided a similar number of samples for each class is available.

8.3 Non-exponential service times

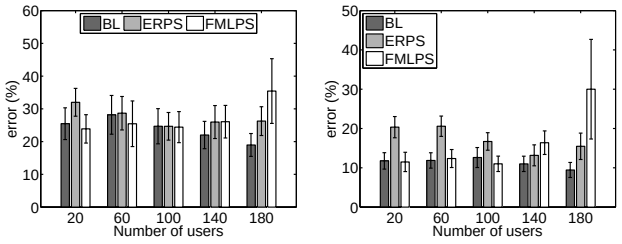
We now consider the impact of non-exponential service times on the proposed estimation methods, since both MLPS and FMLPS assume exponentially-distributed processing times. This is particularly relevant since the exponential assumption does not necessarily hold in real applications. To this end we apply the estimation methods, which assume exponential processing times, while in the simulation the processing times are generated from non-exponential distributions. We consider two main cases for the processing times: an Erlang distribution made of five consecutive exponential phases, resulting in a square coefficient of variation (SCV) of 0.2; and a hyper-exponential (HE) distribution made of the mixture of two exponential phases, which can match any SCV greater than one [37]. In both cases we keep the same mean as in the exponential experiments to maintain the same offered load on the servers.

Figure 12(a) presents the estimation errors under Erlang service times for the case $V = 2$ and $W = 8$. Under this setup MINPS performs better than in the exponential case, depicted in Figure 9(a), with estimation errors close to those of BL. ERPS, however, shows similar results than in the exponential case under low loads, and improves only under high loads. Figure 13(a) also considers Erlang service times, for the case $V = 8$ and $W = 64$. We highlight the performance of FMLPS, which is as accurate as BL. ERPS instead presents larger errors under low loads, similar to the exponential case, depicted in Figure 10(a), while in high loads it performs as well as BL. The methods thus handle well hypo-exponential processing times.

Figure 12(b) illustrates the effect of HE service times with SCV$= 2$. While the results for both MINPS and ERPS worsen, their estimation errors are similar to those of BL, verifying their good performance. Similar results hold for a large number of processors and threads, shown in Figure 13(b), where all the methods show a similar behavior, with a slight advantage for FMLPS (resp. ERPS) under low (resp. high) loads. Notice that the performance of the BL method is affected by the scatter of the data around its mean, improving in the hypo-exponential case and worsening in the hyper-exponential case.

8.3.1 *Extending FMLPS to HE processing times*

We have observed that further increasing the variability of the processing times, beyond an SCV of 2, affects the accuracy of the ML methods, particularly under high loads. To handle this case we extend the FMLPS method to allow for non-exponential, specifically 2-phase hyper-exponential (HE), processing times. To this end, we exploit the class-switching assumption, slightly modifying the ML methods as follows. We assume 2 classes (k_1,k_2) for each class k in the model. For a class-k sample, we assume that it joins the server as a class-k_i request with probability α_i^k. A class-k_i request poses a mean demand $(\mu_i^k)^{-1}$ on the server, and upon completion it leaves the server as a class-k request. The likelihood of sample $(\boldsymbol{n}(t_j), r_j)$



(a) $SS = 100$ (b) $SS = 500$

Fig. 14. HE (SCV=10) - $V = 8$ - $W = 64$

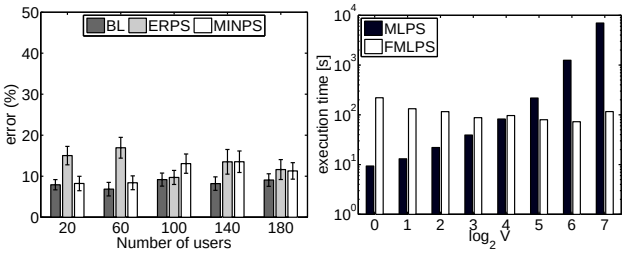


Fig. 15. JSQ Scheduling Fig. 16. Execution times

is obtained by separately computing the likelihood as if it was a class-k_i request and combining the results with weights α_i^k. The method thus needs to estimate three parameters for each class k: μ_1^k, μ_2^k, and α_1^k, as $\alpha_2^k = 1 - \alpha_1^k$.

We illustrate this strategy in Figure 14 by considering highly-variable processing times with SCV = 10. The effect of the high variability is evident as all the methods, including BL, show larger estimation errors. Increasing the sample size to 500, as in Figure 14(b), results in smaller errors and a smaller variability in the estimates. We observe that the modified FMLPS behaves very well under low to medium loads, but shows a large error under high loads. ERPS instead behaves similarly to previous cases, with errors larger than BL and FMLPS under low loads, and similar to BL under high loads. The case of high variability and high load is thus difficult for FMLPS, and in these cases ERPS is preferable. Another option to handle high variability is to extend FMLPS to consider PH distributions, which are more flexible and can approximate long-tailed processing times [38]. The downside is the additional computational effort needed to estimate a larger number of parameters. This will be the topic of future work.

8.4 A different scheduling policy

In the previous scenarios the samples were collected from a simulation where the V CPUs are fully shared, to represent time-sharing in the operating system scheduler. For illustration, we now consider a different scheduling policy, namely Join the Shortest Queue (JSQ), to allocate the jobs to the CPUs. Figure 15 depicts the estimation errors under this policy, for the case $V = 4$, $W = 16$, which are only slightly higher than those obtained under full sharing, depicted in Figure 7(a). The good performance of our methods is related to the ability of the JSQ policy to balance the number of jobs assigned to each processor, although it is blind to the job size. Simpler allocation alternatives, such as random allocation, do not have this balancing property, and the methods presented in this paper are not well suited for their analysis.

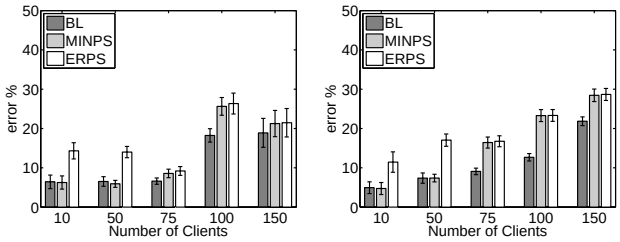
8.5 Computational cost

The regression-based methods, RPS and ERPS, scale very well with the number of processors and threads, since the size of the linear problem that needs to be solved for the regression depends on the number of samples only. The ML procedures are instead computationally more expensive. In fact, the key approximation in MLPS was introduced to deal with the state-space explosion and its effect on the computation times. FMLPS overcomes this problem with the fluid approximation, and can therefore tackle problems with a large number of threads and processors. Figure 16 compares the execution times of MLPS and FMLPS for an increasing number of processors, while keeping the thread-to-processor ratio $W/V = 4$ and the number of users $N = 140$ fixed. This experiment was performed in a 3.4 GHz 4-core Intel Core i7 machine, with 4GB RAM, running Linux Ubuntu 12.04. We observe how, for a small number of processors, MLPS is about one order of magnitude faster than FMLPS. However, its execution times increase rapidly with the number of processors and surpass those of FMLPS. Instead, FMLPS shows a more stable behavior, with execution times between 1 and 4 *min*.

9 Case Study I: SAP ERP Application

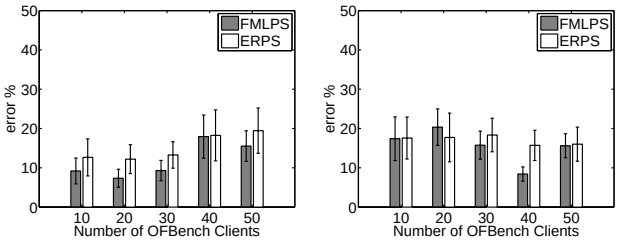
In this section we consider a multi-threaded application with a small threading level, the ERP application of SAP Business Suite [19], which runs on top of SAP NetWeaver [39], a middleware that defines the underlying architecture. Similar to the model in Section 3.1, SAP NetWeaver has an admission control queue that receives the incoming requests, and dispatches them to software threads, referred to as *work processes* (WPs), for processing. Admission occurs when a WP becomes idle, based on an FCFS scheme. Thus, the waiting time in the admission queue tends to become the dominating component of the response time as the number of active users becomes large with respect to the threading level. The operations in the WPs are CPU-intensive and include calls to a database.

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 is the VMware



(a) $W = 3$ (b) $W = 6$

Fig. 17. SAP - MINPS - ERPS - $V = 2$



(a) $R = 4$ (b) $R = 8$

Fig. 18. OFBiz - Estimation errors relative to BL

ESX server configured with 32 GB of memory, 230 GB of storage, and 2 virtual CPUs, running at 2.2GHz and mapped to separate physical CPU cores. We consider different scenarios varying the number of WPs $W \in \{3, 6\}$, and the number of users $N \in [10, 150]$, which correspond to a CPU utilization in the range $[0.03, 0.90]$. The users issue requests via a closed-loop workload generator with exponential think time with mean $10\ s$. The workload we consider consists of six transaction types, related to the creation and display of sales orders, the creation of billing documents, and the generation of outbound deliveries.

We use the data collected to estimate the mean demand with BL, ERPS, and opting for MINPS instead of FMLPS since the threading level W is small. The threading level W is small due to the high memory requirements of the requests. We compare the results against measurements obtained from the internal profiler of the ERP application (STAD). The measurements of the profiler collect the CPU consumption per request type, including initialization steps and database calls. In this case, these execution times are considered part of the request demand since the WP is not freed during the database call, and the database itself is located on the same physical host. Discriminating these times would require additional monitoring, measuring the times consumed in database calls.

Figure 17 depicts the estimation errors obtained with 30 experiments, each based on 600 samples. We observe a small estimation error, particularly for BL and MINPS, when the number of clients is between 10 and 75. For a larger number of clients, the error increases to around 20% for BL, and between 20% and 30% for MINPS and ERPS, which show similar results. Even in this case, both EPRS and MINPS remain very close to the best achievable error (BL). A first source of error is in the processing times' variability, reflected in an SCV in the range $[1.01,\ 1.72]$, which is above that of the exponential, although not very far from it. Another source of error may be related to the database calls, which, as mentioned above, are not explicitly modeled. The proposed methods, in spite of the simplifying assumptions, are able to provide useful estimates, especially under low to medium load scenarios. In these experiments MINPS shows execution times between 1 and 10 *min*, with an average under 3 *min*, while ERPS requires less than 10 *ms*.

10 Case Study II: OFBiz

In this section we turn to an application with a large threading level featuring several tens of concurrently executing requests. We consider the open-source Apache OFBiz e-commerce application[3], which has a three-layer architecture, including presentation, application, and data layers, and uses scripting languages to generate dynamic content. We generate the workload with OFBench [40], a benchmarking tool designed to recreate the behavior of a set of OFBiz users, generating semantically-correct requests.

We have deployed OFBiz and the OFBench client on separate Amazon EC2 instances. Different from the case study in the previous section, the deployment on a public cloud implies that the VMs used can be co-located with other VMs on the same physical host, posing additional challenges to the estimation methods. The traces collected are used to estimate the mean resource demand with the BL, ERPS, and FMLPS methods. In this case, the default maximum number of threads is 200, and we therefore opt for FMLPS instead of MINPS. The OFBiz workload consists of 35 request types [40], a number that significantly increases the execution times of FMLPS. We have therefore clustered these classes into four and eight sets. This approach reflects the practice in performance analysis of aggregating several requests types into a few groups with homogeneous business meaning.

Figure 18 depicts the estimation error obtained with ERPS and FMLPS, when comparing their estimates with those obtained with BL. Different from the ERP application considered in the previous section, no profiler is available for OFBiz, as is the case with many applications, restricting the possibility of obtaining the values of the actual demands. Further, in the previous section the deployment was performed in a private environment, where we could control that only the application VM was deployed on the physical host, and the virtual cores were mapped to

3. Apache OFBiz: http://ofbiz.apache.org/

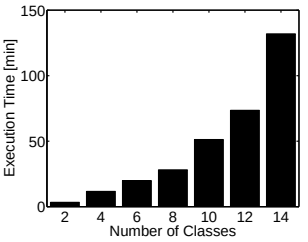


Fig. 19. OFBiz - FMLPS - Execution times

the physical cores. In this section instead, we consider a cloud deployment, where no control is provided over the physical resources, and the actual processing capacity depends on the current resource usage of the physical host. As a result, we make use of the estimates obtained with BL as the basis to compare the other two methods. These results are for a server deployed on a *c1.medium* VM instance, which has 2 virtual CPUs, but we have obtained similar results for other instance types. The scenarios considered, with the number of users between 10 and 50, generate an average utilization between 25% and 65%. We observe similar estimation errors for both methods, with a maximum mean error around 20%. In the case with $R = 4$ classes, the error is around 10% when the number of users is between 10 and 30. For 40 and 50 users, the error is around 20%, which is also the prevalent value for the case with $R = 8$ classes.

We also observe higher variability in the results, which is to be expected given the application characteristics and the cloud deployment. As with the SAP ERP application, we observe that the methods with limited information, FMLPS and ERPS, provide estimates that are in generally good agreement with those provided by BL, which has full information about the requests' arrival and departure times. In the $R = 4$ case, using 400 samples, FMLPS experiences running times between 2 and 30 *min*, with an average close to 9 *min*. It is thus appropriate for offline analysis or online use at coarse, e.g. every hour, timescales. Figure 19 shows how the the FMLPS execution times increase with the number of classes. Although feasible, assuming a large number of classes implies significantly larger computation times, which can be appropriate for offline analysis only. ERPS instead executes in under 10 *ms*, thus being well-suited for online estimation.

11 Conclusion

We have introduced demand estimation methods for multi-threaded applications with either large or small threading levels. An implementation of these methods is available at https://github.com/imperial-modaclouds/modaclouds-fg-demand. The methods provide accurate results under different server setups, including different numbers of processors, threads, and different server loads. The MINPS method is well-suited for low threading levels, while for large threading levels FMLPS (resp. ERPS) offers better results under low (resp. large) loads. MINPS is partly based on MLPS, which assumes a fixed population during the request execution time. This assumption is removed in FMLPS as the underlying fluid model is able to model systems with a large number of requests. In both of the maximum-likelihood methods we proposed, MLPS and FMLPS, we make the simplifying assumption that the response times are independent. Although the response times are not independent, their dependence can be reduced by selecting the samples randomly or well-spaced in time, avoiding response times of jobs whose execution overlap or that correspond to the same busy period.

We evaluate the estimation methods proposed by means of two case studies: an enterprise application deployed on a local host, and an e-commerce application deployed on a public cloud. Cloud deployments pose additional challenges that require further study, particularly regarding the variability in the CPU resources effectively available to the application when other VMs are deployed on the same physical host. Demand estimation approaches that account for this variability, without requiring access to hyper-visor information, could rely on metrics such as the *CPU Steal*, which records the time the virtual CPU has to wait for the physical CPU when the hyper-visor is busy with other virtual CPU [41]. This will be the topic of future work.

References

[1] B. Addis, D. Ardagna, B. Panicucci, and L. Zhang, "Autonomic management of cloud service centers with availability guarantees," in *Proc. of IEEE CLOUD*, 2010.

[2] A. Kalbasi, D. Krishnamurthy, J. Rolia, and S. Dawson, "Dec: Service demand estimation with confidence," *IEEE Trans. Software Eng.*, vol. 38, pp. 561–578, 2012.

[3] L. Zhang, X. Meng, S. Meng, and J. Tan, "K-scope: Online performance tracking for dynamic cloud applications," in *Proceedings of the 10th ICAC*, 2013.

[4] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[5] M. Tribastone, P. Mayer, and M. Wirsing, "Performance prediction of service-oriented systems with layered queueing networks," in *Proc. of the 4th ISoLA'10*, 2010.

[6] S. Becker, H. Koziolek, and R. Reussner, "Model-based performance prediction with the palladio component model," in *Proc. of the 6th WOSP*, 2007.

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

[8] 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 the 5th IEEE CLOUD*, 2012.

[9] P. Cremonesi, K. Dhyani, and A. Sansottera, "Service Time Estimation with a Refinement Enhanced Hybrid Clustering Algorithm," in *Proc. of ASMTA*, 2010.

[10] P. Cremonesi and A. Sansottera, "Indirect estimation of service demands in the presence of structural changes," in *QEST*, 2012.

[11] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson, "Estimating service resource consumption from response time measurements," in *Proc. of the 4th VALUETOOLS*, 2009.

[12] D. Kumar, L. Zhang, and A. Tantawi, "Enhanced inferencing: estimation of a workload dependent performance model," in *Proc. of the 4th VALUETOOLS*, 2009.

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

[14] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "CPU demand for web serving: Measurement analysis and dynamic estimation," *Performance Evaluation*, vol. 65, pp. 531–553, 2008.

[15] Q. Zhang, L. Cherkasova, and E. Smirni, "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications," in *Proceedings of the 4th ICAC*, 2007.

[16] T. Zheng, C. Woodside, and M. Litoiu, "Performance Model Estimation and Tracking Using Optimal Filters," *Software Engineering, IEEE Transactions on*, vol. 34, pp. 391–406, 2008.

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

[18] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queueing networks," *Journal of the ACM*, vol. 27, pp. 313–322, 1980.

[19] (2011) Enterprise resource planning (ERP). [Online]. Available: http://www.sap.com/solutions/business-suite/erp/index.epx

[20] G. Casale, P. Cremonesi, and R. Turrin, "Robust workload estimation in queueing network performance models," in *Proc. of 17th PDP*, 2008.

[21] 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 the CASCON '05*, 2005.

[22] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *Proc. of IEEE NOMS*, 2012, pp. 1287–1294.

[23] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload Analysis and Demand Prediction of Enterprise Data Center Applications," in *Proc. of the 10th IEEE IISWC*, 2007.

[24] C. Isci, J. Hanson, I. Whalley, M. Steinder, and J. Kephart, "Runtime demand estimation for effective dynamic resource management," in *Proc. of IEEE NOMS*, 2010.

[25] F. Brosig, F. Gorsler, N. Huber, and S. Kounev, "Evaluating approaches for performance prediction in virtualized environments," in *Proceedings of MASCOTS*, 2013.

[26] W. Wang and G. Casale, "Bayesian service demand estimation with Gibbs sampling," in *Proc. of IEEE MASCOTS*, 2013.

[27] J. V. Ross, T. Taimre, and P. K. Pollett, "Estimation for queues from queue length data," *Queueing Systems*, vol. 55, pp. 131–138, 2007.

[28] C. Sutton and M. I. Jordan, "Bayesian inference for queueing networks and modeling of internet services," *The Annals of Applied Statistics*, vol. 5, pp. 254–282, 2011.

[29] 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, pp. 36–60, 2006.

[30] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *Proceedings of EuroSys'07*, 2007.

[31] "Apache module log-firstbyte." [Online]. Available: https://code.google.com/p/mod-log-firstbyte/

[32] M. Bertoli, G. Casale, and G. Serazzi, "JMT: performance engineering tools for system modeling," *ACM Perf. Eval. Rev.*, vol. 36, pp. 10–15, 2009.

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

[34] J. F. Pérez and G. Casale, "A fluid model for closed queueing networks with PS stations," Imperial College London, Tech. Rep. DTR13-8, 2013. [Online]. Available: https://www.doc.ic.ac.uk/research/technicalreports/2013/DTR13-8.pdf

[35] T. G. Kurtz, "Solutions of ordinary differential equations as limits of pure jump Markov processes," *Journal of Applied Probability*, vol. 7, pp. 49–58, 1970.

[36] R. A. Hayden, A. Stefanek, and J. T. Bradley, "Fluid computation of passage-time distributions in large Markov models," *Theor. Comput. Sci.*, vol. 413, no. 1, pp. 106–141, Jan. 2012.

[37] W. Whitt, "Approximating a point process by a renewal process, I: two basic methods," *Operations Research*, vol. 30, pp. 125–147, 1982.

[38] A. Riska, V. Diev, and E. Smirni, "Efficient fitting of long-tailed data sets into hyperexponential distributions," in *IEEE GLOBECOM'02*, 2002.

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

[40] J. Moschetta and G.Casale., "OFBench: an Enterprise Application Benchmark for Cloud Resource Management Studies," in *Proc. of MICAS*, 2012.

[41] G. Casale, C. Ragusa, and P. Parpas, "A feasibility study of host-level contention detection by guest virtual machines," in *Proc. of CloudCom*, 2013.

Juan F. Pérez is a Research Associate in performance analysis at Imperial College London, Department of Computing. He obtained a PhD in Computer Science from the University of Antwerp, Belgium, in 2010. His research interest center around the performance analysis of computer systems, especially on cloud computing and optical networking.

Giuliano Casale is a Lecturer in performance analysis and operations research at Imperial College London, Department of Computing, since January 2013. Prior to this, he was a full-time researcher at SAP Research (UK), a postdoctoral research associate at the College of William & Mary (US). He obtained a PhD from Politecnico di Milano (Italy) in 2006. He has served as program co-chair for ACM SIGMETRICS/Performance 2012 QEST 2012, and ICAC 2014, and as general co-chair for ACM/SPEC ICPE 2013, and as technical program committee member for more than 50 conferences and workshops.

Sergio Pacheco-Sanchez is a Research Specialist in performance analysis at the SAP HANA Cloud Computing, Systems Engineering division. Sergio obtained a PhD from the University of Ulster, UK, in 2012. Previously he was a Research Associate at SAP Research (UK) where he investigated and developed statistical inference methods to effectively parameterize queueing models for IT systems. He is currently investigating the impact of the latest hardware and software technologies on the performance of in-memory distributed database systems.