Upstream: Exposing Performance Information from Cloud Providers to Tenants

Adriano Lange, Marcos Sfair Sunye, Luis Carlos Erpen de Bona

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)

{alange,sunye,bona}@inf.ufpr.br

Abstract. *Infrastructure-as-a-Service (IaaS) is a widely adopted cloud computing paradigm due to its flexibility and competitive prices. To improve resource efficiency, most IaaS providers consolidate several tenants in the same virtualization server, which usually incurs variable performance experiences. In this paper, we evaluate the CPU time received by tenants' virtual machines (VMs). We present a model that estimates the probability of a VM to receive, at least, a determined fraction of CPU time using limited information about the host and VMs running on it. We constructed this model using a series of experiments with different numbers of CPU cores and co-located VMs.*

1. Introduction

Infrastructure-as-a-Service (IaaS) is a widely adopted cloud computing paradigm, enabling both flexibility and low costs for computing resources. In order to improve efficiency, and consequently achieve competitive prices, IaaS providers typically consolidate several virtual machines (VMs or guests) from different customers (tenants) in the same physical server (host).

Despite the apparent advantages, sharing computing resources has also brought unwanted hurdles for both providers and tenants. The dualism between reservation and proportional sharing of resources [14] has severe effects on the density of VMs per host and, consequently, on the infrastructure costs. From the tenant perspective, on the other hand, sharing resources with other unknown tenants with their respective unknown workloads in an unknown host has been synonymous of unpredictable performance.

Estimating the performance of applications in shared environments is challenging due to a large number of possible interactions between them and the shared resources. In recent years, substantial work has addressed this problem, analyzing the effects of concurrent applications and proposing different mechanisms for monitoring the consumed resources in order to support better scheduling and allocation decisions, e.g., [11, 9, 8]. A comprehensive survey of this theme is presented by Aceto et al. [3]. Although significant efforts have been addressed to support provider decisions, these authors recognize the lack of mechanisms that permit an effective exchange of performance information between IaaS providers and their respective tenants.

In this paper, we address the above problem of exposing more performance information about the shared resources to the tenants. In virtualized environments, each VM receives one or more virtual CPUs (VCPUs), which are scheduled by the host among the physical CPU cores. The time received by each VCPU depends on the total number of

VCPUs, their demand for CPU, and the physical cores available. Using limited information from the host and the co-located VMs, we implemented a model that estimates the probability of a particular VM to receive a certain fraction of CPU time from the host.

CPU time is a critical resource for many computing-intensive applications, including scientific high-performance computing (HPC) models, machine learning, compression, encryption, and analytical databases. Its lack of predictability is a hindrance for those applications when deployed in shared IaaS environments, compromising the accuracy of internal optimization mechanisms, the quality of service (QoS), and the scheduling of tasks among horizontally scaled computing nodes [5, 13, 15]. Although reserving resources and lowering the density of VMs per host are the most trivial solutions to improve predictability, they certainly incur additional costs for both providers and tenants. By exposing more information to the tenants, our model can improve application decisions without having to pay for dedicated resources.

This paper is organized as follows. In Section 2, we describe the problem of predicting the CPU time inside the guests for scenarios with multiple CPU cores and multiple VMs. The setup and experiments used by our prediction model are presented in Section 3, whereas its construction process is presented in Section 4. In Section 5, we discuss the possible applications for this model, and we conclude this paper in Section 6.

2. Problem Statement

The main objective of this study is to estimate the probability of a particular VM_i to receive a specific fraction (x) of CPU time form the host by using general information about the host itself and the other VMs running concurrently. We call this fraction as potential VCPU time (P_i) for the VM_i. In these terms, the probability (Pr) of achieving $P_i \geq x$ is defined as:

$$Pr(P_i \geq x | I_{\text{host}}, I_{\text{VMs}}) \tag{1}$$

, where I_{host} and I_{VMs} represent sensitive information about the host and the running VMs.

In this paper, we restrict our analysis to one VCPU per VM. We let the evaluation of any number of VCPUs for future work. In our analysis, x may assume values between 0 (no CPU time) and 1 (all CPU time). Also note that, in the Function 1, we use $P_i \geq x$ instead of $P_i = x$. With this formulation, we consider the probability of getting **at least** x fraction of CPU time rather than getting precisely this value. Additionally, $P_i \geq x$ implies that its probability has the form of a survival function, which is equivalent to one minus the cumulative distribution of $P_i = x$. We approximated this function using a normal distribution, which has the form of a sigmoid and can be calculated by Function 2.

$$f(x) = \frac{c}{1 + e^{-k(x - x_0)}} + y_0 \tag{2}$$

From the Functions 1 and 2, this problem can be modeled by extracting the parameters x_0, y_0, c, and k from the information about the host (I_{host}) and the VMs concurrently running in the system (I_{VMs}). In order to determine I_{host} and I_{VMs}, we evaluated many aspects of the Linux kernel, and a tool responsible for extracting information from the

virtualization infrastructure [7, 8]. Table 1 summarizes the parameters and counters considered relevant for this problem formulation. Most of them can be either retrieved or calculated from /sys and /proc pseudo filesystems. Additionally, we also used the libvirt library for Python 3 to obtain information about the VMs running in the host.

Table 1. Main Parameters of the Performance Model

Parameter	Description
M	number of CPU cores in the host available for the VMs
N	number of VMs running in the host
V	number of VCPUs of all VMs ($N = V$)
V_i	number of VCPUs of VM$_i$ ($V_i = 1$)
V_i'	$V_i' = V - V_i$
T_i	time of CPU consumed by the VM$_i$
T	total CPU time of all VMs ($T = \sum_{i=1}^{N} T_i$)
S_i	steal time of VM$_i$
S	total steal time of all VMs ($S = \sum_{i=1}^{N} S_i$)
D_i	demand for VCPU of VM$_i$ ($D_i = T_i + S_i$)
D	demand of all VMs in the host ($D = \sum_{i=1}^{N} D_i$)
D_i'	$D_i' = D - D_i$

The parameters M and N are the number of CPU cores available and the number of VMs in the host, respectively. V represents the total VCPUs in the system, whereas V_i denotes the number of VCPUs in a VM$_i$. As mentioned previously, we assume $V = N$ and $V_i = 1$. The parameters M and N only refer to the resources available and potentially consumed in the host by the VMs, but not their real use. The parameter T_i, in turn, represents the portion of CPU time received by the VM$_i$. The host only grants this time to the VM if requested. For the host's kernel, idle time is not accounted to any VM. On the other hand, if the host can not provide all the time requested by a VM, its VCPU is put in the ready queue of the host's scheduler until it is possible to provide more CPU time. This waiting time is computed as steal time (S_i) for the VCPU.

We define D_i the demand for CPU of a VM$_i$ and use the sum between the service time (T_i) and the waiting time (S_i) to calculate it. Although this sum may not be realistic, once the time in the ready queue of a VCPU may be higher than the computing time needed by the VM, we consider this difference negligible and, therefore, a good approximation of this demand. Finally, we respectively denote D_i' and V_i' as the demand and the number of VCPUs of all VMs in the host, except VM$_i$.

Although all information described in Table 1 is accessible by the host, only V_i, T_i, and S_i can be retrieved inside the VM$_i$. For example, in recent versions of the Linux kernel, the host exposes this information to the guests by using model-specific registers (MSRs) [6]. Inside the VMs, many performance tools (e.g., top and vmstat) read this information from the /proc filesystem and display it to the users.

Without more information about the current availability of CPU in the host, the most trivial alternative for a VM is to probe the host by demanding 100% of its VCPU and measuring S_i. However, this is a resource-consuming approach, which may incur in performance degradation if several VMs perform this probe regularly in the same host. Once one probe in a particular VM may potentially affect the others, the arbitrary use

of this approach in production environments may reduce the performance predictability instead of increasing it.

In order to minimize the above problem, a better alternative is to provide guests with more information about the usage of host resources or even provide more accurate performance models about them. In this paper, we have followed this approach by implementing two main programs in Python 3 [10]. The first one, named **system**, runs inside the host and collects statistics about the physical CPUs and running VMs. Such information is then compiled and shared with the VMs by using an in-memory filesystem. Each VM received a separate directory in this filesystem to share information. The 9P protocol performed the connection between these directories in the host and the respective VMs. To avoid overexposing information, each VM can only see statistics about itself (including T_i, S_i, and V_i), the counters and a summary per CPU core, in addition to the totals V, T, and S.

The second program, named **guestmodel**, runs inside the VMs and reads the information produced by the first program to predict P_i. In order to determine the values x_0, y_0, c, and k, we used $I_{\text{host}} = M$, and $I_{\text{VMs}} = (N, D'_i)$. In this paper, we omit V_i and V from I_{VMs} once V_i is always equal to 1 and $V = N$. The model $\mathcal{M}$ represents a mapping between the above parameters and the quadruple x_0, y_0, c, k:

$$(x_0, y_0, c, k) = \mathcal{M}(M, N, D'_i) \tag{3}$$

We defined $\mathcal{M}$ by using a series of experiments with different values for M and N. The data obtained by these experiments were used to construct different regression models necessary to calculate the quadruple x_0, y_0, c, k. In the next section, we describe these experiments, whereas the construction of $\mathcal{M}$ is presented in Section 4.

3. Setup and Experiments

The experiments conducted in this study were performed on a computer with one Intel i5-2400, containing four CPU cores, and 16GB of RAM DDR3. The operating system (OS) used by the host was Ubuntu 16.04 LTS running with the version 4.4 of the Linux kernel. We also used KVM as the virtualization infrastructure in addition to Qemu 2.5 and Libvirt 1.3.1.

To evaluate the performance interference between VMs, we used a set of 11 experiments with different numbers of CPU cores and VMs. Considering the pair (M, N) an experiment with M CPUs and N VMs, this set of experiments is $\{(2, 4)$, $(2, 5)$, $(2, 6)$, $(2, 7)$, $(2, 8)$, $(3, 4)$, $(3, 5)$, $(3, 6)$, $(3, 7)$, $(3, 8)$, $(3, 9)\}$. For $M = 2$, the experiments cover a VM density varying from 2 to 4 times the number of CPU cores. For $M = 3$, this density is between 1.33 and 3. We used a cpuset cgroup of the Linux kernel to control the set of physical cores available to the VMs. Within this cpuset, we let the host schedule VCPUs among physical cores without any prioritization of VMs. In order to reduce possible noises coming from outside the experiments, the other processes running in the host were bound to the remaining CPU cores. The only exception was IRQ processing, which remained in its default configuration.

In this work, we did not conduct experiments using one CPU core, once it is not possible to migrate VCPUs between cores, nor using all CPU cores, once we reserved at

least one core to control the experiments. We leave the evaluation of more CPU cores, especially the new manycore processors, for future work. All VMs used in the experiments are clones of the same OS installation; in this case, Debian 9. The virtual hardware assigned to each VM was 1 VCPU and 1GB of RAM.

In each experiment, our primary objective was assessing the CPU time (T_i) and steal time (S_i) of a particular VM$_i$ while the demand of the other VMs (D'_i) varied. This scenario requires a benchmark that can coordinate the CPU load of several VMs running in the same host. We initially analyzed several commonly used benchmarks [1, 12, 2]. However, most of them were designed for assessing the performance of different CPU features and other hardware characteristics, which do not adequately match with the objectives of this study.

Instead of using or extending any existing benchmark, we implemented two small programs to generate and control the VM load [10]. The first one, named **load**, runs inside each VM and consists of a closed-loop that reads a set of files and compresses them using the zlib library. Between each compression, the load program sleeps for a period determined by the second program, named **loadcontrol**, which resides on the host and coordinates the experiments. In order to inform the sleep interval for each VM, this program used the same shared folder described in the previous section.

Inside the VMs, each instance of the load program used a separated thread to read the sleep interval. This second thread prevents the main thread from being unnecessarily blocked while reading information from the shared folder. In each loop of the main thread, the load program compressed a set of forty-four files with 5MB each. We produced these files by extracting 220MB from Linux source code (.c and .h files). The extracted lines of code were shuffled before splitting them into the 5MB files in order to homogenize the compression time. In order to prevent unnecessary I/O operations, we used in-memory filesystems to store these files in each VM.

In each experiment, we fixed the sleep time of the first VM to zero ($D_1 \approx 1$) and varied the sleep time of the remaining VMs in order to produce different levels of D'_1. We progressively increase D'_1 from 0 to V'_1 by setting their sleep time to -1 ($D_i \approx 0$) and between 0.8 and 0. During the increase of D'_1, different sleep times were randomly assigned to these VMs.

Figure 1 depicts the measurements of the experiment with 3 CPUs and 6 VMs. In this figure, the top-most graph represents the first VM ($i = 1$), whereas the other VMs are sorted below. The horizontal axis of the figure represents the elapsed time in seconds. In order to produce consistent performance behavior, we varied the elapsed time according to the number of VMs in each experiment.

All the experiments performed in this study follow the same pattern depicted in Fig. 1. While D'_1 stayed very low, we observed $T_1 \approx D_1$ and $S_1 \approx 0$. As D'_1 increased, the CPU demand of the other VMs forced the host's scheduler to decrease T_1, and, consequently, increase S_1. In order to prevent VM$_1$ from dominating the host's CPU, we periodically set D_1 to 0 for one second and again to 1 This behavior can be observed by the vertical lines in the graph of VM$_1$. In the next section, we analyze the pattern produced by these experiments in more details.

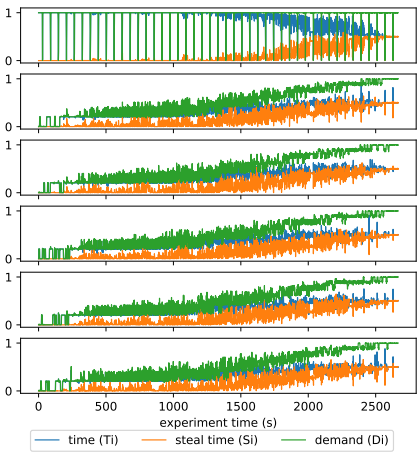
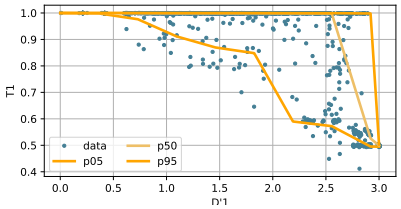


Figure 1. Measurements of the experiment with 3 CPUs and 6 VMs.

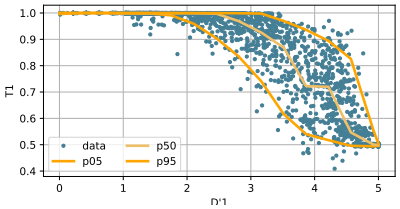
4. Model Construction

In this section, we analyze the data obtained by the experiments described in Section 3. From now on, we will only consider the behavior of VM_1 as a function of D'_1, i.e., the CPU demand of all VMs except VM_1. Due to space restrictions, we will not show all the graphs produced from these experiments. These graphs are available in the project repository [10].

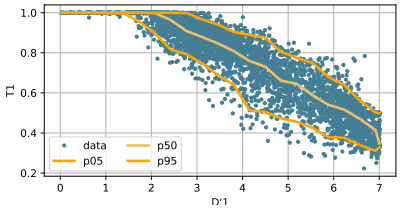
The graphs in Figure 2 represent the data obtained by the experiments $(2, 4)$, $(3, 6)$, and $(3, 8)$, where (M, N) are the numbers of CPUs and VMs, respectively. These graphs demonstrate how T_1 is affected by D'_1 in different scenarios. In all experiments, we considered only the data where $D_1 \geq 0.98$, which means that S_1 is always close to $1 - T_1$. Therefore, we will not show S_1 in these graphs.



(a) M=2, N=4



(b) M=3, N=6



(c) M=3, N=8

Figure 2. Experiments Data.

In addition to the data, we also show in Fig. 2 the 5th (p05), 50th (p50), and 95th (p95) percentiles. The graphs 2a and 2b represent two low-density scenarios of VMs per

CPU. In such conditions, VM_1 can receive good performance most the time. On the other hand, graph 2c depicts a high-density scenario, where VM_1 may experience severe performance degradation. We used the 5th and 95th percentiles as baselines to construct the model $\mathcal{M}(M, N, D'_i)$. We define the **upper function** $u(M, N, D'_i)$ as an approximation of the 95th percentile. Similarly, the **lower function** $l(M, N, D'_i)$ represents an approximation of the 5th percentile. Finally, we constructed $\mathcal{M}$ as a normal distribution between l and u.

We also used sigmoids as the form of the functions l and u. Although other function types were also evaluated in this study, including polynomials, exponentials, and inverses, sigmoids presented the best tradeoff between low- and high-density scenarios. Both l and u were created by using reference points. These points are coordinates in the space $T_i \times D'_i$ and represent the main characteristics of each curve. For the function l, we used four points: the initial ($\mathbf{l}^{\text{ini}}$) and final ($\mathbf{l}^{\text{fin}}$) coordinates of the sigmoid, in addition to two intermediate points ($\mathbf{l}^{\text{midT}}$ and $\mathbf{l}^{\text{midB}}$) used to indicate the location of the inflection point and the angle of the sigmoid at that point. We calculated these points from the parameters M and N by combining both analytical and linear regression models[1].

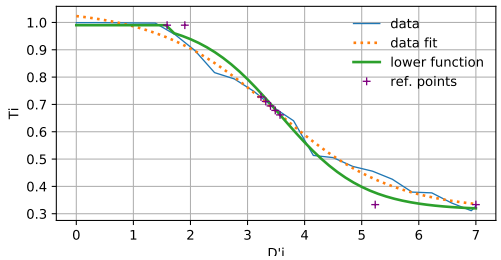


Figure 3. Construction of the lower function l (M=3, N=8).

Figure 3 depicts the 5th percentile, a sigmoid generated by the curve_fit[2] function using the 5th percentile, and the lower function with its reference points for the experiment $(3, 8)$. We divided l into two parts. The first one stays in the left of the initial point ($\mathbf{l}^{\text{ini}}$), where T_i is constant and close to 1 (0.99, to be more precisely, once we consider $l < u$). The second part of l is between $\mathbf{l}^{\text{ini}}$ and $\mathbf{l}^{\text{fin}}$, which is represented by the sigmoid that characterizes the decline of T_i.

Both the initial and final points of l have one trivial component each. For the initial point, its T_i component is always equal to 0.99 (i.e., $\mathbf{l}^{\text{ini}} = (?, 0.99)$), whereas $\mathbf{l}^{\text{fin}}$ has the component D'_i always equal to V'_i (i.e., $\mathbf{l}^{\text{fin}} = (V'_i, ?)$). Therefore, we only had to estimate D'_i for $\mathbf{l}^{\text{ini}}$ and T_i for $\mathbf{l}^{\text{fin}}$. For this later, we defined the component T_i as $\lceil N/M \rceil^{-1}$. When D is equal to N, the CPU time tends to be equally distributed among the VCPUs (i.e., M/N). However, if the number of VCPUs is not a multiple of M, some CPU cores may receive more VCPUs than others.

In order to estimate the D'_i component of $\mathbf{l}^{\text{ini}}$, we evaluated several possible linear and non-linear relationships between the initial decline of the 5th percentile of T_1 and the parameters of Table 1. As depicted in Figure 4, the best model obtained was a linear

[1]We used the sklearn package for linear regressions: www.scikit-learn.org.

[2]From scipy package: www.scipy.org.

regression between M, N, and $V'i - D'i$. We observed the same pattern for the 50th and 95th percentiles. In Fig. 4, we also plot a projection of this model for $M = 4$, but we leave the confirmation for future work.

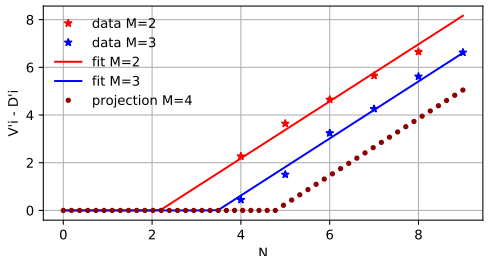


Figure 4. Initial reference point of the lower function l (5th percentile).

The intermediate points $\mathbf{l}^{\text{midT}}$ and $\mathbf{l}^{\text{midB}}$ are used to determine the position of the inflection point and the angle of l at that point. In Fig. 4, the point $\mathbf{l}^{\text{midT}}$ is to the right of $\mathbf{l}^{\text{ini}}$, whereas $\mathbf{l}^{\text{midB}}$ is to the left of $\mathbf{l}^{\text{fin}}$. Respectively, these points have the same T_i components of $\mathbf{l}^{\text{ini}}$ and $\mathbf{l}^{\text{fin}}$, i.e., $\mathbf{l}^{\text{midT}} = (?, 0.99)$ and $\mathbf{l}^{\text{midB}} = (?, \lceil N/M \rceil^{-1})$. In order to calculate the D'_i components, we used the line tangent to the inflection point of the sigmoid generated by the 5th percentile of the experiments ("data fit" in Fig. 4). The intersections of this line with $T_i = 0.99$ and $T_i = \lceil N/M \rceil^{-1}$ were used to fit linear regressions for $\mathbf{l}^{\text{midT}}$ and $\mathbf{l}^{\text{midB}}$. For $\mathbf{l}^{\text{midT}}$, the best fit was using M, N, the D'_i component of $\mathbf{l}^{\text{ini}}$, and V'_i. For $\mathbf{l}^{\text{midB}}$, we used M, N, and the distance between $\mathbf{l}^{\text{ini}}$ and $\mathbf{l}^{\text{fin}}$ in each component.

Figure 5 illustrates the construction of the upper function (u) for the experiment $(3, 8)$. This process was more straightforward than constructing l. For the function u, we used the upper half of a sigmoid, which required only three reference points: $\mathbf{u}^{\text{ini}}$, $\mathbf{u}^{\text{fin}}$, and $\mathbf{u}^{\text{midT}}$. Similarly to $\mathbf{l}^{\text{ini}}$, $\mathbf{u}^{\text{ini}}$ separates the regular part of u on the left ($T_i = 1$) from the part where T_i begins to decline. Once $\mathbf{u}^{\text{fin}}$ represents the inflection point of the sigmoid, only one intermediate point (i.e., $\mathbf{u}^{\text{midT}}$) was necessary to calculate the angle of u at $\mathbf{u}^{\text{fin}}$.

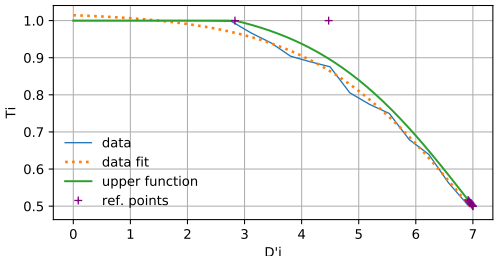


Figure 5. Construction of the upper function u (M=3, N=8).

All the reference points used to construct u have one predefined component: $\mathbf{u}^{\text{ini}} = (?, 1)$, $\mathbf{u}^{\text{midT}} = (?, 1)$, and $\mathbf{u}^{\text{fin}} = (V'_i, ?)$. We applied the same method used in $\mathbf{l}^{\text{ini}}$ to define the D'_i component of $\mathbf{u}^{\text{ini}}$. Figure 6 presents the linear regression between M, N, and $V'i - D'i$ for the initial decline of T_i in the 95th percentile. As in Fig. 4, we also plot a probable projection for $M = 4$.

Similarly to $\mathbf{l}^{\text{fin}}$, we analytically defined the T_i component of $\mathbf{u}^{\text{fin}}$ as $\lfloor N/M \rfloor^{-1}$. For the function u, we use floor instead of ceil to indicate that, in case of imbalance,

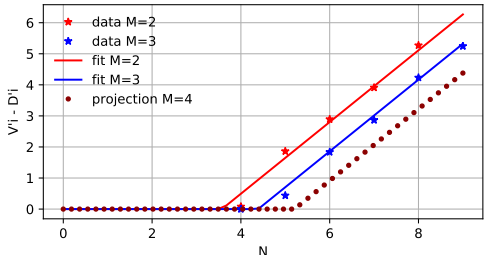
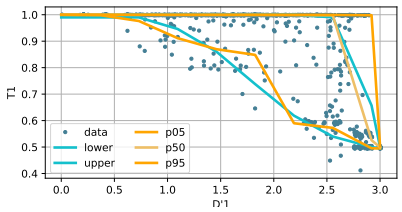


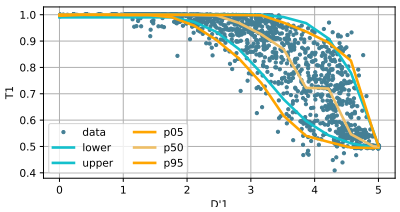
Figure 6. Initial reference point of the upper function u (95th percentile).

the VCPU of a VM_i may be scheduled for a CPU core with fewer VCPUs. Once $\mathbf{u}^{\text{ini}}$ and $\mathbf{u}^{\text{fin}}$ were established, we defined $\mathbf{u}^{\text{midT}}$ by using a linear regression between: (a) the D'_i component of $\mathbf{u}^{\text{ini}}$; (b,c) both components of $\mathbf{u}^{\text{fin}}$; and (d) the D'_i component of the intersection between $T_i = 1$ and the tangent line at the inflection point of the sigmoid produced by the curve_fit function using the experiment data ("data fit" in Fig. 6).

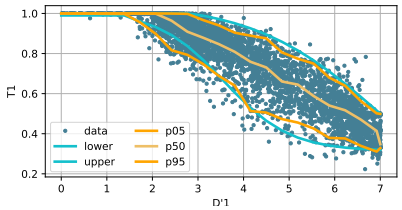
The graphs in Figure 7 show the upper and lower functions described above for the same experiments as in Fig. 2. In most cases, l and u achieved good approximations of the 5th and 95th percentiles, especially for high-density scenarios. The graph 7a also shows an error in the initial decline of the upper function for the experiment $(2, 4)$. We can also observe this error in Fig. 6.



(a) M=2, N=4



(b) M=3, N=6



(c) M=3, N=8

Figure 7. Lower (l) and Upper (u) Functions.

Using the functions l and u as approximations of the 5th and 95th percentiles, we constructed a normal distribution between them and calculated the values for the model $\mathcal{M}$. Once both l and u are dependent on D'_i, this model is also subject to the intervals produced these functions. We used the norm class of the scipy library to generate the normal distribution between l and u. Figure 8 illustrates the probability function $Pr(P_i \geq x | M = 3, N = 8, D'_i = 5.5)$ generated by the sigmoid of Function 2 and the parameters produced by $\mathcal{M}(3, 8, 5.5)$. In this figure, we also plot the survival function obtained from the data of experiment $(3, 8)$ with the same D'_i. The intersections between $D'_i = 5.5$ and the functions l and u are also shown as reference (ref) points.

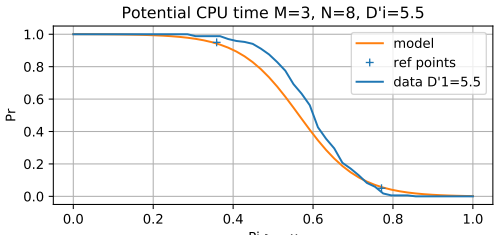


Figure 8. Potential CPU time.

There are two exceptions to the use of normal distribution between l and u in model $\mathcal{M}$. The first and most trivial case is when $D'_i \leq \mathbf{l}^{\text{ini}}$. Once l and u are approximately 1, in this case, we return a constant function (x^0). The second exception is when $\mathbf{l}^{\text{ini}} < D'_i < \mathbf{u}^{\text{ini}}$. Once u is limited to $T_i \leq 1$, this function does not represent the 95th percentile when $D'_i < \mathbf{u}^{\text{ini}}$. In this case, we projected u above this limit and then calculated the normal distribution. Figure 9 illustrates this projection of u for $M = 3$, $N = 8$, and $D'_i = 2.5$.

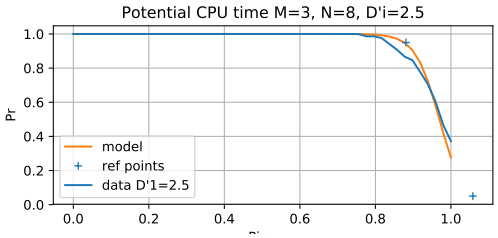


Figure 9. Projection of function u outside its limit.

5. Discussion and Application

In this section, we discuss some characteristics and potential applications of our prediction model. In order to exemplify its use, we implemented a simple performance monitor that runs inside the guest and combines our model with a histogram of D'_i. The following textbox presents the output of this monitor:

```
Guest Usage            :  50.4%    Guest Steal :   49.5%    Guest Demand : 100.0%
Total CPUs:            :  3        Total VCPUs :   7        Other Demand :  99.9%
Scale (%) ------------ :   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
Other Demand Histogram :   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  45  55
Potential VCPU Time    : 100 100 100 100 100  99  97  90  68  38  17   7   2   1   0   0   0   0   0   0   0
```

In the above example, the histogram represents historical information about D'_i normalized between 0 and 100%. From our prediction model, we generated a probability matrix with the same number of rows and columns as the buckets of the histogram. Each row corresponds to a probability function for each normalized D'_i. In each iteration of the performance monitor, it multiplies the histogram and the probability matrix in order to estimate the potential CPU time for the VM.

Our prediction model is orthogonal to both the method used to determine D'_i and the performance model implemented by each application. Instead of using historical information about D'_i, a possible improvement to the above example might be to use predictive methods to provide short- and long-term forecasts. These methods could be implemented either in the host or in the guests. In any case, this implementation is beyond the scope of this work.

Each application may exhibit distinct resource requirements and react differently to their contentions. Another possible use for our model is combining it with domain-specific performance models and orchestration mechanisms. For example, most distributed HPC applications need to coordinate their tasks among several computing nodes. Lack of predictability in computing resources is one of the critical factors that prevent many HPC applications from being deployed in shared environments. Although our model was not designed to deliver more resources to these nodes, it can be used to improve the accuracy of their coordination process.

Another relevant example of combining our model with domain-specific models is in database systems. Although most of these systems implement their specific cost models for optimization, performance prediction is still a problematic matter in this field, even for dedicated hardware [4, 16, 17]. Especially for CPU-intensive queries and in-memory databases, sharing CPU time with other tenants may further impair the performance predictability in addition to compromise several administrative and tuning decisions. These systems could use the estimations about the potential CPU time to improve the accuracy of their models in shared environments.

6. Conclusion

In this paper, we have presented a model that estimates the probability of a particular VM to receive at least a determined fraction of CPU time from the host. This model uses limited information about the host and running VMs, which is forwarded to each VM by using shared directories. We constructed this model based on a series of experiments with different numbers of CPU cores and VMs. In each experiment, we evaluated the CPU time received by a particular VM while the other VMs varied their CPU demand. From these experiments, we applied different linear and non-linear regression models to produce probability functions of CPU time.

In our experiments, we evaluated only two and three CPU cores and only one VCPU per VM. A future direction of this work is to consider more CPU cores in the host and allocate more VCPUs per VM. However, one potential concern with increasing the number of VCPUs per VM is the increased complexity of our model if the host's scheduler does not consider these VCPUs independently. Besides the number of CPU cores, hyperthreading and NUMA architectures are other relevant hardware scenarios for further investigations.

Another future research direction is to automate the construction of this model. Although we have evaluated different combinations of CPU cores and VMs, the accuracy of our model may vary according to the virtualization infrastructure and the version of the Linux kernel running on the host. Therefore, one possible solution is to implement mechanisms that validate the accuracy of the existent model and recreate it if necessary. Furthermore, many cloud datacentres have a large number of identical hosts. In such cases, the automated mechanism could recreate the prediction model only once and replicate it to all equivalent machines.

References

[1] SPEC - Standard Performance Evaluation Corporation, . URL `http://www.spec.org/`.

[2] Stress-ng, . URL http://kernel.ubuntu.com/{~}cking/stress-ng.

[3] G. Aceto, A. Botta, W. de Donato, and A. Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013. doi: 10.1016/j.comnet.2013.04.001.

[4] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *International Conference on Extending Database Technology (EDBT/ICDT)*, page 449, Uppsala, Sweden, 2011. ACM Press. doi: 10.1145/1951365.1951419.

[5] M. Armbrust. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[6] G. Costa. KVM-specific MSRs. URL https://github.com/torvalds/linux/blob/master/Documentation/virtual/kvm/msr.txt.

[7] C. B. Hauser. Kvmtop. URL https://github.com/cha87de/kvmtop.

[8] C. B. Hauser and S. Wesner. Reviewing Cloud Monitoring: Towards Cloud Resource Profiling. In *IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 678–685, 2018. doi: 10.1109/CLOUD.2018.00093.

[9] C. B. Hauser, J. Domaschka, and S. Wesner. Predictability of Resource Intensive Big Data and HPC Jobs in Cloud Data Centres. In *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 358–365, 2018. doi: 10.1109/QRS-C.2018.00069.

[10] A. Lange. Upstream Repository. URL https://github.com/alange0001/upstream.

[11] F. Licht, B. Schulze, L. C. E. Bona, and A. R. Mury. Analysis of parallelized libraries and interference effects in concurrent environments. *Computers & Electrical Engineering*, 66:435–453, 2018. doi: 10.1016/j.compeleceng.2017.08.028.

[12] NASA. NAS Parallel Benchmarks. URL https://www.nas.nasa.gov/publications/npb.html.

[13] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.*, 3(1-2):460–471, 2010. doi: 10.14778/1920841.1920902.

[14] I. Stoica, H. Abdel-wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *In Proc. of Multimedia Computing and Networking*, pages 207–214, 1997.

[15] F. Wu, Q. Wu, and Y. Tan. Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, 71(9):3373–3418, 2015. doi: 10.1007/s11227-015-1438-4.

[16] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *International Conference on Data Engineering (ICDE)*, pages 1081–1092, 2013. doi: 10.1109/ICDE.2013.6544899.

[17] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton. Uncertainty Aware Query Execution Time Prediction. *Proc. VLDB Endow.*, 7(14):1857–1868, 2014. doi: 10.14778/2733085.2733092.