A classifier for the latency-CPU behaviors of serving jobs in distributed environments

Christophe Restif   Natalia Ponomareva   Krzysztof Ostrowski

Research at Google, New York

crestif, nponomareva, ostrowki@google.com

Abstract

End-to-end latency of serving jobs in distributed and shared environments, such as a Cloud, is an important metric for jobs' owners and infrastructure providers. Yet it is notoriously challenging to model precisely, since it is affected by a large collection of unrelated moving pieces, from the software design to the job schedulers strategies. In this work we present a novel approach to modeling latency, by tracking how it varies with CPU usage. We train a classifier to automatically assign the latency behavior of methods in three classes: constant latency regardless of CPU, uncorrelated latency and CPU, and predictable latency as a function of CPU. We use our model on a random sample of serving jobs running on the Google infrastructure. We illustrate unexpected and insightful patterns of latency variations with CPU. The visualization of latency-CPU variations and the corresponding class may be used by both jobs' owners and infrastructure providers, for a variety of applications, such as smarter latency alerting, latency-aware configuration of jobs, and automated detection of changes in behavior, either over time, during pre-release testing, or across data centers.

Categories and Subject Descriptors D.4.7 [*Operating systems*]: Organization and Design—Distributed sytems

Keywords Machine learning, weighted scatterplot modeling, resource management, performance, intelligent alert management.

1. Introduction

Latency of serving jobs is an important factor to keep under control, and often to minimize. For user-facing products, even small variations of end-to-end latency can make a difference in user experience; and for jobs offering contractual services to other jobs, increases of latency leading to violations of Service Level Agreement (SLA) are often directly charged as monetary fines. Yet, since latency is not a resource being shared and managed directly (unlike memory, CPU, disk space, etc.), it often is not included in state-of-the-art multi-resource modeling [1, 12]. Alternatively, latency is modeled independently within scheduling frameworks [7, 11, 17], since the scheduling order of jobs in distributed and shared environments may have a direct influence on latency. Yet latency may also be affected by factors beyond the control of a job scheduler - it may simply be proportional to the size of the job's input being processed. Successful examples of latency-focused modeling have been reported for specific classes of jobs (database services [16], straggler jobs [18]) or specific configurations (addressing the handling of unused resources [4]). In this work we consider all types of serving jobs, running in distributed and shared environment, responding to any type of remote procedure calls (RPC) with measurable end-to-end latency.

The Cloud paradigm brings additional constraints to traditional distributed and shared computing [3, 17]. Depending on the depth of the Cloud service, customers may only see the underlying infrastructure as a black box running virtual machines, with limited information about their inner performance; and the schedulers may only see the running jobs as black boxes too, with little control over their configuration [9]. This necessary inner separation makes it more difficult to track and improve end-to-end latency.

In this work we model the variations of end-to-end latency correlated to CPU consumption for any type of serving job running in production. These two standard metrics may generally be readily available to Cloud customers, as part of billing or SLA monitoring. The combination of these two metrics provides a novel insight into the behavior of methods, and has a wide variety of applications. For our model training and all our experiments, we use data from serving jobs running on the Google Borg infrastructure [15, 19]. We identify 3 classes of behaviors: constant latency at all CPU levels; unpredictable latency at all CPU levels; predictable latency as a function of the CPU. Smarter alerting rules may

Submission to SoCC '14,   October, 2014, Seattle, WA, USA.

	Hierarchical organization			*Aggregated in*
DC	*job*	*task*	*method*	*scatterplot*
data center DC1	frontend-prod	task1	HandleGetRequest	S1
			HandlePostRequest	S2
		task2	HandleGetRequest	S1
			HandlePostRequest	S2
		task3	HandleGetRequest	S1
			HandlePostRequest	S2
	helper	task1	HandleGetRequest	S3
		task2	HandleGetRequest	S3
		task3	HandleGetRequest	S3
	...	...	...	...
DC2	helper	task1	HandleGetRequest	S4
		task2	HandleGetRequest	S4
		task3	HandleGetRequest	S4

Figure 1: Hierarchical organization of methods, and the corresponding scatterplots where their data will be aggregated (names are for illustration purposes only).

be set up for methods depending on which class they fall, and depending on the CPU consumption at the time of a high latency. For jobs whose latency increases with CPU consumption, higher number of replicas can reduce latency at a small cost. Also, for infrastructure providers, these combined metrics provide information on what latency to expect at different loads and usages, which can be included in an automated configuration framework. Finally, we find that job behaviors do change, either over time, over new releases, or over data center, and therefore we suggest that this classifier may be part of a daily monitoring console, both for infrastructure providers and for jobs' owners.

2. Classifier for latency-CPU behavior

2.1 Aggregating data from comparable methods

Serving jobs running on Google data centers have been described in [15, 19]. In short, a job is duplicated over several data centers, for redundancy, reliability (in case a data center goes offline), and for geographical proximity to callers and to backends. In each data center, a job is a collection of multiple tasks or replicas; they are the units of work that handle the incoming RPC calls and consume resources. The number of replicas may be set by the user, or may be highly dynamic [9] and controlled by the scheduler. Each task responds to RPCs through one or many methods (*e.g.* the two methods HandleGetRequest and HandlePostRequest may well be served by the same task). The latency is measured for each method independently. In a given data center, all tasks of a job run the same binary, and use the same methods for RPCs. However, on two different data centers, a job may be running different binaries, in particular during the roll-out of a new release, and may even be exposing different methods and responding to different types of RPCs.

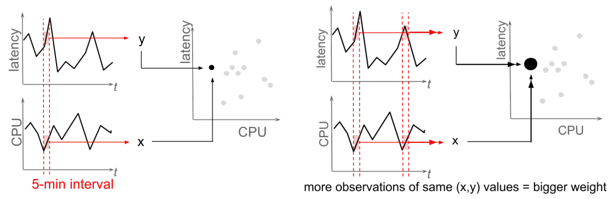

Figure 2: Computation of a weighted scatterplot representing the correlation between two timeseries. Left: the coordinates of a point in the scatterplot represent the CPU and latency from the same 5-min interval. Right: the weight of a point represents the duration when the two values were observed together.

In this architecture, there are several levels of duplication, but not all duplicates are actual clones that can be aggregated as single units with identical behavior. In this work we consider the following as the unit of behavior: one method from one job in one data center, combining the data from all the corresponding tasks. We argue that doing the aggregation at a different level would be detrimental to latency-CPU analysis. Aggregating the latency of different methods served by the same job, which may well be within different orders of magnitude from each other, would lose the functional interpretation of latency, since latency is method-specific. Also, aggregating jobs across data centers would hide the specificities of each data center (whose location may have a strong impact on latency), and would potentially fail during the roll-out of new releases of existing jobs with new methods. On the other hand, all tasks of a job in a data center are guaranteed by design to have identical behavior (running the same binary, sharing the same configuration, and serving the same requests), and are treated as interchangeable by the job scheduler and the load balancer. Aggregating them gives more data on which to fit our models. We illustrate this in Fig. 1: the left part of the table shows the hierarchical organization of methods; the right side shows which methods are actual clones with the same behavior, and whose data are aggregated into single scatterplots.

To ensure the generality of our approach, we selected 500 serving jobs, with a total of $1,882$ methods, from 8 randomly-selected data centers, among the jobs that met the following minimum replication requirements: each job was running in at least 2 other data centers, and each job had at least 3 tasks in each data center.

2.2 Computing weighted scatterplots from timeseries

The CPU consumed by each task is averaged every five minutes. For the latency of a given method, we measure the 90th percentile of all values during the same five-minute interval. These two timeseries, CPU and latency, are computed continuously, and stored over periods of 3 days. The process is repeated every 1.5 days, which ensures there is always some overlap between two consecutive scatterplots of a given method. The 3-day interval is chosen to be large enough to contain enough data for modeling purposes, and short enough to detect temporal changes of behavior of methods over time.

The two timeseries for a target over 3 days are then combined into a single weighted scatterplot (see Fig. 2). Each point in the plot is a triplet (x=CPU, y=latency, w=duration), where w (for weight) is the total duration over which this CPU/latency pair is observed during the 3 day period of measurements. This data is displayed here as weighted scatter plots, where x is the CPU value (in cores), y the latency (in ms), and the size of a point is proportional to the weight. The longer the CPU/latency values are observed together, the higher the weight in the data, and the bigger the point in the plot. Overall, we use a total of $3,379$ weighted scatterplots.

2.3 Classes of latency-CPU behaviors

The end-to-end latency of serving jobs is challenging to model precisely, especially when considering a general population of serving jobs (some being user-facing and latency-sensitive, others being background processes with little concern for latency). Additionally, as explained in the Introduction, in a Cloud context, the breakdown of a latency chain is not publicly available to jobs' owners.

In this context, with no prior information on jobs and no inner knowledge of infrastructure, we identified three classes of latency behaviors for serving jobs (illustrated in Fig. 3). The first class is for plots with *constant* latency at all CPU values. Methods in this class show little to no variations of latency (as a fraction of the absolute latency value), over the complete range of CPU. The corresponding jobs offer straightforward rules for latency monitoring and alerting, and are typically robust to resource allocation.
The second class behavior is *uncorrelated* latency and CPU values. While both latency and CPU vary significantly, their variations are uncorrelated. The latency may depend on external factors (such as the complexity of the input values, or the latency of backend calls).
The third class is *predictable* latency as a function of CPU. Scatterplots of methods in this category exhibit a wide variety of patterns, as illustrated in Fig. 4. Patterns include: latency increasing with CPU (mostly non-linearly), latency decreasing with CPU, and V-shaped variations of latency with CPU (high latency at small CPU values, low latency at medium CPU range, and high latency at high CPU values). We think the *predictable* class has the most promising applications. While it is certainly interesting to further model the variations of latency with CPU for methods in that class, we found that keeping them in the same class was more reliable, as a classifier task, than adding more classes for each specific pattern. Since the first two classes represent the majority of classes, dividing the third class further would cause a strong imbalance in the number of representatives for each class, which notably leads to lower classification performance. A straightforward extension of this work would be to model the variations of latency only in this third class.

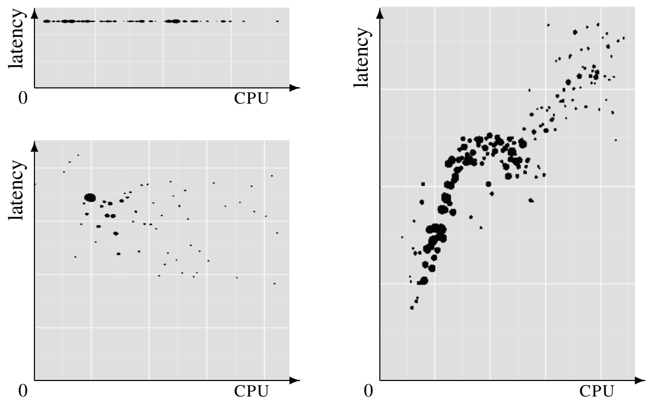

Figure 3: Three classes of latency behavior. Top let: constant. Bottom left: uncorrelated. Right: predictable. As described in section 2, the the size of a point represents the duration when its coordinates were observed together.

2.4 Data preprocessing and feature extraction

Because outliers both can change the scale of the plots (which will make labeling the plots by humans harder) and significantly change some of the statistics (like mean), we first remove outliers from the data. We use the widely popular Tukey method for outliers detection [13], which filters out data points that fall outside of interval $(Q1-1.5\times IQR, Q3+1.5\times IQR)$, where IQR is the interquartile range [14], $Q1$ and $Q3$ are the first and third quartiles respectively. This filtering is done for each axis (CPU and latency), independently of each other. Note that here and everywhere when we mention some statistic, like IQR, we mean a weighted statistic, since each point on the scatter plot has a weight assigned to it.

Once outliers are filtered out, we compute a total of 34 features for each data plot. We briefly describe these features below and provide the rationale for adding them to the model.

First, we break the CPU axis into n intervals and for each interval $i \in [2,n]$ we add as features the relative change of weighted mean and relative change of weighted standard deviation of latency. More formally, if the latency in some bucket i has a weighted mean of μ_i and a standard deviation of σ_i, then for a bucket $i+1$ we add two features: $(\mu_{i+1}-\mu_i)/\mu_i$ and $(\sigma_{i+1}-\sigma_i)/\sigma_i$. For buckets with a previous bucket's mean or standard deviation of 0, these features are undefined, due to division by 0. We replace such values with some constant value, which is outside of the range of the values observed for this feature, at the post processing step before model is trained. For the first bucket, since we can't calculate relative change, we simply include the weighted mean and weighted standard deviation of the latency for the points in this bucket.

Additionally, for each bucket we add a feature representing the relative total weight concentrated in this bucket (*i.e.* the proportion of observations that this bucket contains). It is calculated as follows for a bucket i:

$$\frac{\sum_{\text{point}_j \in \text{Bucket}_i} \text{weight}_j}{\sum_{k=1}^{N} \text{weight}_k}$$

where N is the number of points in the plot. Empirically we chose n=5 as the number of buckets.

The rationale behind these $3 \times n$ features is that in constant plots, we would expect to see a very small relative change in both mean and standard deviation of latency, whereas in uncorrelated plots, the changes might be potentially large. In the predictable class, we do expect to see the changes between the buckets, but for each bucket the changes should be on a smaller scale than for uncorrelated (points are more tightly coupled in the predictable case). The weight in the bucket helps the model to understand whether the relative change in values is only due to a small number of observations. This is useful in particular for cases where the data in concentrated in few points in the bucket, and thus the mean and the standard deviation calculated are not reliable estimates.

The next set of features is also weight-related. Intuitively, having few relatively heavy data points containing the majority of the weight and some light outliers is a good signal for a constant class; whereas having all points, including outliers, with approximately the same weight would indicate an uncorrelated plot. Therefore, as a signal, we include the relative weight of largest, smallest and median points, in relation to the total weight of all points in the plot. We also include the number of observations as additional signal, since a larger number of observations allows us to distinguish between uncorrelated versus constant or predictable plots: few points are unlikely to form a pattern and are most likely either uncorrelated or constant (that distinction being a function of their relative weights and coordinates).

We also include features that describe the CPU and latency behavior globally: weighted index of dispersion [8], weighted *IQR* [14], weighted variance, and logarithms of maximum, minimum and median latency. The purpose of the first three features is to quantify the dispersion of the data points: the more spread out the points on a scatter plot are, the more likely this scatter plot is to be classified as uncorrelated; on the contrary, "concentrated" points would most likely represent either constant plots or some sort of pattern. The last three features help estimate the change in latency on the scatter plot. Since all plots from the training dataset have very different latency ranges, from plots with all points at a few *ms* to plots with points with latency of several hundred thousands *ms*, we use a log of the last three features. The CPU related features include weighted *IQR* of CPU values and weighted variance.

To estimate the strength of a predictable relationship between latency and CPU, we fit a linear regression model and include its slope, intercept, standard errors of coefficients and sigma (estimate of standard deviation of noise in the model). The slope of the regression line is a measure of correlation between CPU and latency (adjusted by the standard deviations), which helps distinguish uncorrelated and constant classes from predictable classes. Standard errors and sigma are included to understand how accurate the fit is. Finally, we also fit a 3-piece spline, which we experimentally decided was good enough to describe the patterns visible in our dataset, and include its *adjusted r-squared* metric, which is a goodness of fit measure.

Table 1: Confusion matrix for the random forest classifier detailed in Section 2.

		Predicted		
		Constant	Uncorrel.	Predictable
Actual	Constant	**139**	11	1
	Uncorrelated	13	**91**	11
	Predictable	5	9	**43**

2.5 Classifier and evaluation

During our experiments we tried a number of different classifiers, but since the decision boundary is highly non linear, only ν-SVM with RBF kernel [5, 6] and random forests [2] were finally shortlisted. Rule-based classifiers had the advantage to be straightforward to implement and interpret, but we found that they were systematically out-performed by SVM and random forest. Upon tuning parameters for both of the models, SVM exhibited 75.6% cross validation accuracy (with variance of up to 3% depending on the split), whereas random forests achieved 15.48% out of the bag error rate. Given the study of out-of-bag estimates for bagging predictors [10], which provides empirical evidence that such estimates are close to optimal generalization accuracy values, we consider it equivalent of 84.2% generalization accuracy and therefore we choose random forests model as the winning model. Since the majority class (constant) in a training dataset represented approximately 47% of all training instances, with predictable class approximately 18%, and 10-15% level of disagreement among human experts, we consider the achieved cross validation accuracy to be acceptable for preliminary analysis. The confusion matrix for random forests is shown in Table 1.

Using the random forest model, we classified 3,379 available scatter plots. Our analysis of the results showed that the most common misclassification was between the classes *constant* and *predictable* (on which human experts also tend to disagree, when the range of latency is small).

Interestingly enough, the model actually classified correctly, as *predictable*, plots with patterns of latency-CPU variations that were not present in the training data. Some examples are provided in Fig. 4. This illustrates that our classifier is not actively looking for particular patterns (such as linear increase of latency), but to predictability in general.

The trained model was used for preliminary analysis, and can be improved by both collecting more labeled data and

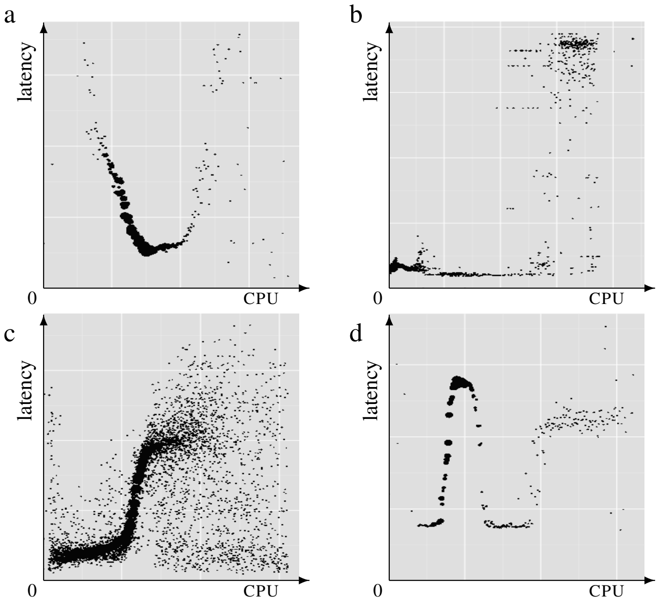

Figure 4: Interesting patterns discovered by the model, which were not present in the training set

agreeing on more formal rules of handling contentious plots, affected by intra- and inter-user variability (different experts, or one expert at different times, may label a borderline case as different classes). With these rules, it will be easy to add additional features into the model to improve the accuracy on bordeline plots.

Overall, the trained random forest model classified 54.48% plots as uncorrelated, 30.61% as constant and 14.91% as predictable. The second best trained model (SVM) when applied to the same data, produced similar proportions (44.5% plots classified as uncorrelated, 38.5% as constant and 17% as predictable).

3. Applications

A typical use of latency values is to set alarms responding to threshold crossing. The scatterplots presented in this work show that there might be smarter rules of latency depending on the concurrent CPU values. As shown in Fig. 4c, a high latency may be acceptable at a high CPU usage (*e.g.* for a task processing a large input), but may be unacceptable at a lower range (which may result from a backend spike in latency). Another application of these scatterplots is to assist the determination of acceptable CPU ranges for tasks. If a maximum latency is provided as a Service Level Objective (SLO), jobs in classes *constant* and *uncorrelated* would be unaffected by any level of CPU allocation, whereas jobs in classes *predictable* may need to cap the CPU usage if the scatterplots goes beyond the SLO.

For infrastructure providers, having access to the latency variations with CPU may be used to automatically adjust the resources allocated to jobs when the data shows a risk of exceeding an SLO or SLA. Even when no such service

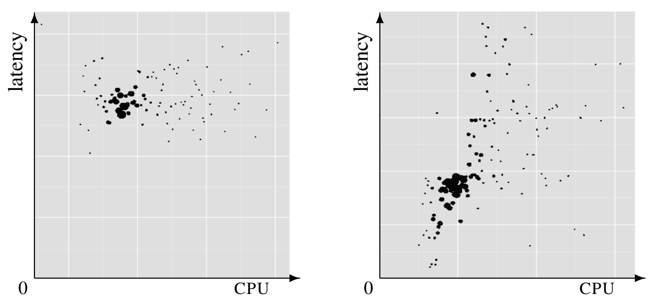

Figure 5: Different latency behaviors of the same method over time running on the same data center. Left: uncorrelated. Right: predictable.

level exists for the latency of a job, the scatterplots provide a way to determine regions of lower latency and regions of exploding latency. This may be used internally as part of the scheduler algorithms.

3.1 Variations of behavior over time

To test how consistent methods' behaviors are with time, we compare the outcome of the classifier for the same methods separated by at least 3 days. Out of $1,881$ such methods, we find that 300, or 15.9%, were classified as different classes. Examples are presented in Fig. 5. From a job owner's perspective, this means that approximately 1 in 6 jobs may exhibit a different a different latency behavior, with the same binary running in the same data center. Such changes may be due to a change of behavior in a dependency or a backend, or due to a different configuration for the job.

3.2 Variations of behavior in testing setups

Another application of our classifier is to determine whether a new binary release will affect the latency behavior. We compare the outcomes of 25 new binary releases which have the same methods as their existing counterpart and run in the same data centers. That is a necessary constraint for this test, which limits the number of candidates (a common pattern is to run a new release on a different data center). We find that 14 new releases (56%) exhibit different behaviors (an example is provided in Fig. 6). While this may be due to different software design, it may impact the configuration of the job in production (from its resource quotas to its alerting rules).

3.3 Variations of behavior over data centers

A third application is to compare the behavior of a given method running on different data centers. This may be more of interest for the infrastructure management than for a job's owner, especially in a Cloud context where users may not always control the data center where their virtual machines are running. We compare the outcome of $2,542$ methods running on at least two different data centers over the same time period. We find that 179 (7.04%) are assigned to different classes in at least two data centers (see Fig. 7). Different hardware configuration in data centers may explain some

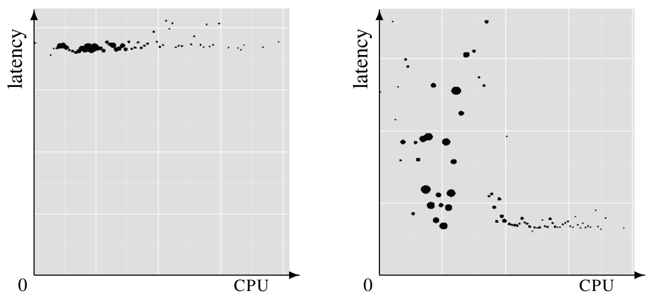

Figure 6: Different latency behaviors over different releases for the same method running on the same data center. Left: constant. Right: predictable.

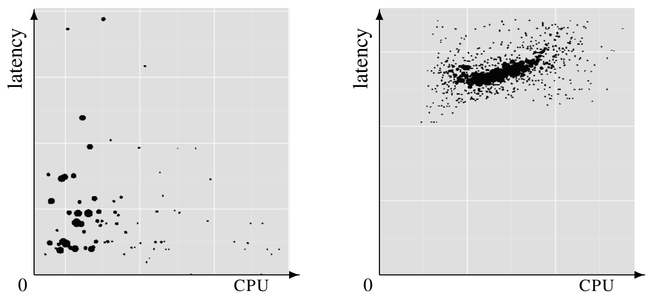

Figure 7: Different latency behaviors for the same method running on two distinct data centers. Left: uncorrelated. Right: predictable.

variations, as well as different geographic locations from backends and from callers.

4. Conclusion

In this work we presented a novel approach to modeling latency of serving jobs in distributed and shared environments, by tracking the correlation variations of latency and CPU. We used a random-forest classifier to assign methods' behaviors in three pre-defined classes: constant latency, uncorrelated latency, and predictable latency as a function of CPU. Our classifier was applied to a random collection of scatterplots of serving jobs running in production at Google. We illustrated how the behavior of a given method may change over time, with new releases, and depending on the underlying infrastructure. It is our hope that this work will motivate further analysis of latency as a function of more infrastructure resources, which tend to be considered separately in the literature. We envision that the scatterplots and the job behavior will provide a richer insight for jobs' owners as well as infrastructure providers, allowing them to set smarter alerts and more appropriate configuration.

Acknowledgments

We thank Abdullah Al-Nayeem, Daniel Bundala, Eran Gabor, and Sebastian Pueblas, for providing the tools collecting the data used in this work, and Boris Dadachev and Afshin Rostamizadeh for their comments on the article.

References

[1] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SOCC '13*, pages 4:1–4:15, 2013.

[2] L. Breiman. Random forests. *Mach Learn*, 45(1):5–32, 2001.

[3] S. Butt, V. Ganapathy, and A. Srivastava. On the control plane of a self-service cloud platform. In *SOCC '14*, pages 10:1–10:13, 2014.

[4] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term slos for reclaimed cloud computing resources. In *SOCC '14*, pages 20:1–20:13, 2014.

[5] P.-H. Chen, C.-J. Lin, and B. Schölkopf. A tutorial on ν-support vector machines: Research articles. *Appl. Stoch. Model. Bus. Ind.*, 21(2):111–136, 2005.

[6] C. Cortes and V. Vapnik. Support-vector networks. *Maching Learning*, 20(3):273–297, 1995.

[7] C. Curino, D. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *SOCC '14*, pages 2:1–2:14, 2014.

[8] E. J. Hannan. Spectral analysis for geophysical data. *Geophysical Journal International*, 11(1):225–236, 1966.

[9] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[10] L. B. Statistics and L. Breiman. Out-of-bag estimation.

[11] P. Stuedi, B. Metzler, and A. Trivedi. jVerbs: Ultra-low latency for data center applications. In *SOCC '13*, pages 10:1–10:14, 2013.

[12] P. Tembey, A. Gavrilovska, and K. Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *SOCC '14*, pages 14:1–14:14, 2014.

[13] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley Publishing Company Reading, Mass., 1977.

[14] G. Upton and I. Cook. *Understanding Statistics*. Oxford University Press, 1996.

[15] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

[16] P. Xiong, Y. Chi, S. Zhu, H. Moon, C. Pu, and H. Hacigumus. Smartsla: Cost-sensitive management of virtualized resources for cpu-bound database services. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1441 – 1451, April 2014.

[17] Y. Xu, M. Bailey, B. Noble, and F. Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *SOCC '13*, pages 7:1–7:16, 2013.

[18] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *SOCC '14*, pages 26:1–26:14, 2014.

[19] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 379–391, 2013.