2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

Adaptive Performance Modeling and Prediction of Applications in Multi-Tenant Clouds

Hamidreza Moradi, Wei Wang and Dakai Zhu
The University of Texas at San Antonio
San Antonio, Texas, 78249
{hamidreza.moradi, wei.wang, dakai.zhu}@utsa.edu

***Abstract*—Clouds have been adopted by many organizations as their computing infrastructure due to the support for flexible resource demands and low cost, which is normally achieved through sharing the underlying hardware among multiple cloud tenants. However, such sharing can result in large variations for the performance of applications running in virtual machines (VMs) on same hosts. In this paper, we propose *User-APMP*, a user-level application performance modeling and prediction framework, based on micro-benchmarks and regression techniques for applications that run repetitively in clouds (such as on-line data analytics). Specifically, a few micro-benchmarks are devised to probe the in-situ perceivable performance of CPU, memory and I/O components of the target VM. Then, based on such probe information and in-place measured performance of applications, the performance model can be adaptively developed and refined at runtime with regression techniques. Moreover, sliding-windows are exploited to control the number of historical data items to retrain the model. We evaluate the prediction accuracy for the considered benchmark applications. The evaluation results show that, the prediction error of User-APMP generally decreases with higher adaptation frequencies and more historical data points, which however leads to higher runtime overhead. With only 100 data points, the average prediction errors can reach 25%.**

I. Introduction

Due to the low cost-of-ownership, clouds have been increasingly adopted by many organizations to serve as their main computing infrastructures. However, this low cost is generally achieved through sharing hardware resources by multiple virtual machines (VMs) on the same host with *multi-tenancy* of different users. Such sharing of hardware may lead to resource contention, which in turn will negatively affect the performance of the applications running in the VMs [1]. The impacts of such contention on the performance can be application specific with their resource requirements. Moreover, as the number of VMs on the same host and applications running in different VMs change over time, the severity of such contention can vary significantly, causing the performance of an application fluctuates in a quite large range.

Such performance variation has made it very challenging to predict the performance of an application running in clouds. However, it is critical to obtain the accurate knowledge on given applications' performance for cloud users to select the correct type and number of VMs and design efficient auto-scaling policies to meet their performance and cost objectives.

This work was supported in part by US National Science Foundation award CCF-1617390.

There have been several studies on the prediction of the application's performance under hardware resource contention. However, many of these studies focus on system level techniques that normally require extensive knowledge of the applications (including their resource demands and detailed memory behaviors) that share the hardware resources [2, 3, 4]. However, in public cloud environment, users typically do not have control over which groups of applications and their associated VMs will share the hardware, nor do they have a-prior knowledge of the behaviors of these applications. Although Bubble-flux and ESP are two predictive techniques that do not require prior knowledge of the co-running applications [5, 6], they would require access to low level hardware performance monitoring units (PMU), which are not always available to the ordinary users in public clouds.

To predict the performance of applications in cloud environments, PARIS has been designed as a predictive model that exploits resource profiling information obtained from the OS on different public cloud services [7]. Similar, Scheuner and Leitner used micro-benchmark profiles to predict application's performance on various VMs in different clouds [8]. However, both approaches can only predict an application's average performance on various clouds. In particular, they cannot be utilized to predict the *instantaneous* performance of an application running on clouds. Such runtime performance information can be important for cloud users to make proper decisions on when to trigger auto-scaling operations [9] and for time-sensitive applications to satisfy their timeliness requirements [10, 11].

Therefore, for ordinary cloud users who usually do not have prior knowledge of colocated VMs with the contending applications and have no access to special hardware registers or counters about the underlying host machines, a new *user-level* prediction framework is desired. In this paper, we propose **User-APMP**: *a User-level Adaptive Performance Modeling and Prediction framework*. User-APMP only utilizes user-perceivable information in the cloud environment to build the predictive models and hence to predict the instantaneous performance of applications running in VMs at runtime.

Specifically, based on the profiling technique, User-APMP exploits several micro-benchmarks to assess the level (severity) of the resource contention for CPU, memory and I/O components, respectively, caused by colocated VMs on the same host as well as their running applications. Combining such

978-1-7281-2058-4/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2019.00097

contention information with the in-situ measured performance of a give application, a regression based predictive model can be trained at runtime to learn the application's sensitivity to the levels of resource contention. Once the application-specific model is obtained, it can be utilized to predict the application's instantaneous performance based on the in-place profiling information of the micro-benchmarks.

For such an online adaptive predictive model in User-APMP, there are two major affecting factors: the *adaptation frequency* (i.e., how often the model is refined/re-trained) and *the length of historical data* (i.e., the number of data points regarding to the measured application performance with the correlated contention level utilized to train the model). Here, the adaptation frequency is controlled by a *batch size*, which indicates the number of correlated data points to be predicted/collected before the model is refined/re-trained at runtime. Moreover, based on the concept of sliding-window, the length of historical data will be determined by a *window-size*, which denotes the number of batches (of data points) to be utilized to re-train the model.

The proposed User-APMP has been evaluated by running PARSEC benchmark applications [12] in a target VM on a cluster with the OpenStack environment. Here, to emulate the different levels of resource contention from the colocated VMs on the same host, different number of background VMs are randomly introduced at runtime periodically to execute either CPU or I/O intensive applications. For different batch and window sizes, both the accuracy of the predictive model in User-APMP and their associated runtime overheads are evaluated. The results show that larger window sizes (i.e., more historical data points) can generally reduce the prediction errors of the predictive model for all the considered benchmarks. Moreover, smaller batch sizes (i.e., higher adaptation frequencies) may lead to more accurate model. However, smaller batch sizes and large window sizes can result in higher runtime overheads. For the considered benchmarks, the batch size of 10 with the window size of 10 (i.e., the total of 100 data points) enables User-APMP achieve 25% average prediction error.

The reminder of this paper is organized as follows. Section II presents the proposed User-APMP framework and the detailed online adaptation approach to obtain the predictive models based on regression techniques. The evaluation methodology, experiment setups and evaluation results are discussed in Section III. Section IV concludes the paper.

II. User-APMP: User-level Adaptive Performance Modeling and Prediction in Clouds

In this work, we focus on applications (such as data and graph analytics that deployed in online machine learning applications [3, 13]) that run repeatedly on a given VM at the request of users in the cloud environment. In general, the computation demand of such applications is linearly related to the size of data to be processed. Hence, for simplicity, we assume that their execution times are affected only by the interference (i.e., resource contention) from the collocated VMs on the same host machine (i.e., the data to be processed in each iteration is assumed to have a fixed size). However, we would like to point out that, the developed predictive model can be easily extended to incorporate the data size, especially when such a size has a known (e.g., linear) relation with the application's execution time.

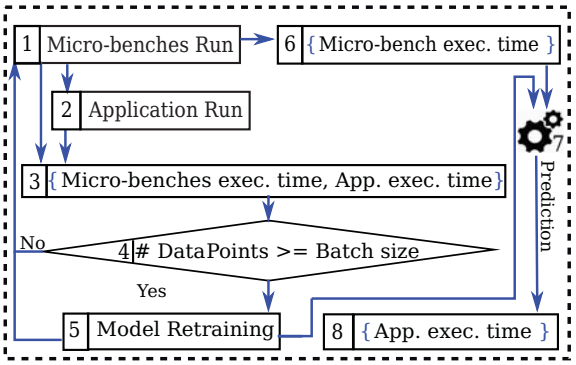


Fig. 1: Overview and workflow of User-APMP.

In what follows, we first present the overview of User-APMP and the workflow for the proposed framework. The micro-benchmarks that are used to assess the level of resource contention in CPU, memory and I/O are discussed next. Then, based on the correlated performance of these micro-benchmarks and a target application, the performance predictive model of the application in a given VM is developed. At last, to address the tradeoff between model accuracy and overhead, the adaptation of the predictive model at runtime is discussed based on the sliding-window technique.

A. Overview of User-APMP

The overview of the proposed User-APMP framework is shown in Fig. 1. Here, as the first step, a few specially designed micro-benchmarks (see Section II-B) will run in the target VM to assess the current level of resource contentions in CPU, memory and I/O components due to other applications running in VMs collocated on the same host machine. Then, the application will run in the target VM in step 2. The measured execution time of the application will be correlated with the obtained performance of the micro-benchmarks in step 1 to form a data tuple. Note that, the interference experienced by the application may change during its execution due to changes in collocated VMs and/or their applications. However, exploring such changes is beyond the scope of this paper and will be left for our future work.

Once enough (new) number of data tuples are collected, the performance predictive model for the application can be developed (or re-trained adaptively) based on regression techniques (step 5; see Section II-C). As shown in the figure, we use the *batch size* to control how frequently the model is re-trained (see Section II-D for more details). Once the model is obtained, based on the profiling information from running the micro-benchmarks at the beginning of each iteration, the application's performance can be predicted at runtime (step 7).

B. Resource-Contention Profiling: Micro-benchmarks

As a user-level framework, we assume that User-APMP does not have access to performance counters in the hypervisor and special registers in the underlying hardware of the host machine. Instead, User-APMP interacts only with the target VM in which the user's application will run. Note that, the key components that affect the performance of a VM are its (virtual) CPUs, memory and disks[1]. To perceive the actual performance of these components for the user application at runtime, we designed a few micro-benchmarks to estimate their contention levels due to interference from the collocated VMs and their applications on the same host machine. Through experiments, we found that the prolonged (shortened) execution of a micro-benchmark can be a indicator of increased (decreased) contention level for the related resource, respectively. In what follows, we explain the detailed design for each micro-benchmark.

CPUs: For resource contention in CPUs, a multi-threaded micro-benchmark is designed to stress the performance of the virtual CPUs of a given VM. Here, each thread will loop through and decrement an *in-register counter* that is initiated with a given value. These in-register operations ensure that this microbenchmark's performance is not affected by memory at runtime and thus examine the contention in CPUs to the maximum extent. The amount of time for each thread to reach zero for the in-register counter is recorded. The number of created threads at runtime for this micro-benchmark will be equal to the number of virtual CPUs of the target VM. In the end, the longest execution time (t_{cpu}) from all threads will be used as the indicator of the contention level for the virtual CPUs in the target VM.

Memory: Similarly, to stress the memory bandwidth of the target VM, the memory micro-benchmark will access a 2GB array with the stride size of 128 sequentially. The objective of such a memory access pattern is to ensure that each data access needs to go to the off-core memory rather than the on-chip caches. Again, the number of threads in this micro-benchmark is the same as the number of virtual CPUs in the VM and each of them accesses the equal portion of the array. The execution time of this micro-benchmark will provide us the insight into the performance impact of the memory contention experienced by the target VM in the system.

Disk I/Os: For the I/O performance of the target VM, we design the disk micro-benchmark that reads 256MB data from the disk with the page size of 4KB. During the execution of this micro-benchmark, the OS file caching will be disabled to prevent the data file being cached by the OS. The micro-benchmark adopts four threads, which will introduce enough I/O operations to stress the disk's bandwidth while avoiding too much inter-thread communication. Again, the execution time for this micro-benchmark to access the required amount of data in the file will be utilized to indicate the contention level of the disk.

In each iteration, these micro-benchmarks will be first invoked sequentially, followed by the execution of the user's application. In this work, we assume that the changes in the contention level of the resources measured by the micro-benchmarks will affect the execution of the application and be reflected by corresponding changes in its execution time. Here, the measured execution times of the micro-benchmarks and the application in each iteration form a data tuple $\{t_{CPU}, t_{mem}, t_{disk}, t_{app}\}$, which will be used to train or retrain the performance predictive model for the application, as discussed below.

C. Regression-based Performance Predictive Model

For a given set of available data tuples, the relationship between the execution times of the target application and the levels of resource contention measured by the micro-benchmarks can be derived (or learned) with regression techniques [14]. Then, based on such derived relationship, the performance predictive model for the target application can be developed. For our measured data tuples, we have conducted extensive experiments with different degrees of polynomial regression and found that the 2-degree polynomial regression fit the relationship for the considered applications the best, while the linear regression and 3-degree polynomial regression can result in either under-fitting or over-fitting problems.

With the 2-degree polynomial regression being adopted in this paper, the relationship between the execution times of the application and the micro-benchmarks can be represented as:

$$\begin{aligned} t_{app} =& f(t_{CPU}, t_{mem}, t_{disk}) \\ =& \alpha_1 \cdot t_{CPU}^2 + \alpha_2 \cdot t_{mem}^2 + \alpha_3 \cdot t_{disk}^2 + \\ & \alpha_4 \cdot t_{CPU} \cdot t_{mem} + \alpha_5 \cdot t_{CPU} \cdot t_{disk} + \alpha_6 \cdot t_{mem} \cdot t_{disk} \\ & + \alpha_7 \cdot t_{cpu} + \alpha_8 \cdot t_{mem} + \alpha_9 \cdot t_{disk} + \alpha_{10} \end{aligned} \tag{1}$$

There are many existing packages can be exploited to find out the values of the coefficients in the above equation and train the predictive model, such as Lasso, Elastic Net, Ridge and Stochastic Gradient Decent regression techniques [15, 16, 17, 18], for a given set of data tuples. We have evaluated the methodology with these different algorithms and similar results have been obtained. In what follows, due to space limitation, we focus on reporting the model training and prediction results with the Lasso regression algorithm only.

D. Online Adaptation of the Predictive Model

Once the predictive model is developed by determining the coefficients in Equation (1) with regression techniques, it can be utilized to predict the performance of the given application based on the profiling information obtained from running the micro-benchmarks. However, due to the limitation of the predictive model, the predicted execution time can differ from the application's actual execution time and result

[1]As the initial study of this problem, we consider applications running only on a single VM. Hence, we do not consider network issues, which will be studied in our future work where applications may run on multiple VMs.

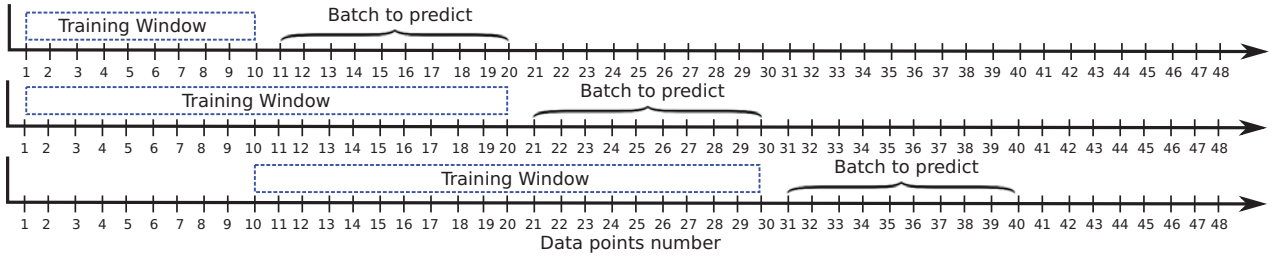


Fig. 2: Sliding-window based adaptations for the predictive model with batch size of 10 and window size of 2.

in prediction errors. As the resource contention caused by other collocated VMs and their applications can change over time, which may not be captured in the predictive model, the prediction errors may increase at runtime. To address this problem, we can adaptively retrain the predictive model by incorporating new data tuples that contain information related to the recent resource contention levels.

For the online adaptation of the predictive model, two major issues have to be considered. First is the *adaptation frequency* (i.e., *how often* and *when* the model should be retrained). The second issue is regarding to the number of historical data tuples should be exploited to retrain the model. Intuitively, more historical data tuples can improve the accuracy of the predictive model, which however will introduce higher runtime overheads. In this work, we consider a *sliding-window* based adaptation approach, which is explained in details below to address the aforementioned two issues.

Batch Size (i.e., Adaptation Frequency): After each iteration of running the micro-benchmarks and an application in a given VM, a new data tuple can be obtained from the in-place measured performance. In theory, the predictive model could be retrained after each new data tuple is obtained to incorporate the new information and, hopefully, to get a better model for more accurate prediction of the application in the next iteration. However, this would introduce significant runtime overheads as shown in our evaluations (see Section III).

Therefore, to control the adaptation frequency and regulate how often (and when) the predictive model should be adaptively retrained at runtime, a *batch size* is utilized. It denotes the number of new data tuples accumulated in a batch before the predictive model is retrained. Fig. 2 shows an example with the batch size of 10. That is, once a predictive model is obtained (after the 10^{th} data point), it will be utilized to predict the performance of the target application for its execution in the next 10 iterations. At the meantime, 10 new data tuples will be generated from the profiling information of the micro-benchmarks and the measured performance of the target application. Before the 21^{th} iteration, the predictive model will be retrained by utilizing the recent data tuples in the training window.

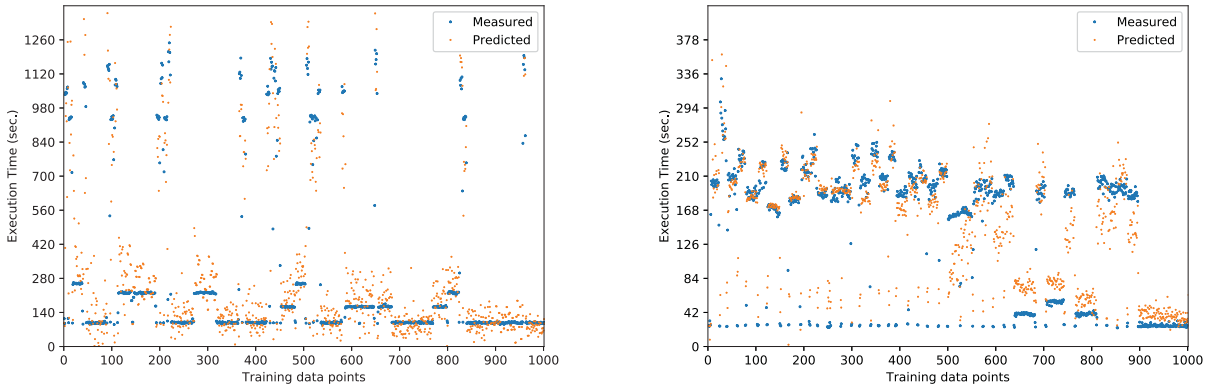
Clearly, having smaller batch sizes will enforce the predictive model be retrained more frequently at runtime, which may improve the accuracy of the prediction results. On the other hand, it will in turn lead to higher runtime overheads. For the extreme case where the batch size equals 1, the predictive model will be retrained after every iteration of running the target application. The tradeoffs between the prediction accuracy and runtime overheads have been extensively evaluated as reported in Section III.

Window Size (Historical Data Tuples): When it is the time to retrain the predictive model, we have to decide *which part* and *how many* historical data tuples should be exploited. Intuitively, the most recent data tuples should be utilized as they contain recent resource contention information that can help the predictive model to get better prediction results for the target application's execution in future iterations. Moreover, instead of utilizing all historical data tuples, which can lead to prohibitive runtime overhead as well as excessive memory space demand for model retraining, we adopt the *sliding-window* technique to control the number of historical data tuples to be utilized for retraining the model.

Specifically, a sliding-window contains a certain number of the most recent batches (where the number is denoted as *window size*). Only the data tuples in these batches will be utilized to retrain the predictive model. At the beginning of the execution, the first few sliding-windows may not have enough batches and contain fewer number of data tuples for training. For the example shown in Fig. 2, it has the window size of 2. Here, the first sliding-window has only one batch of 10 data tuples.

Once enough number of batches are accumulated, the predictive model will be retrained with a certain number of most recent data tuples that are determined by both batch and window sizes. For the example in Fig. 2, it has the batch size of 10 and window size of 2. Therefore, (up to) 20 most recent data tuples in the training window will be utilized to train/retrain the predictive model.

When the window size is set as ∞, this extreme case will reduce to where all historical data tuples are needed for retraining the predictive model. Moreover, when a given number (e.g., 100) of historical data tuples are desired to retrain the model, various combinations of batch and window sizes can be adopted (e.g., batch and window sizes of being 20 and 5 vs. 10 and 10). Apparently, these settings will affect the overall prediction accuracy and the runtime overheads (see Section III for the detailed evaluation results).



a. Streamcluster b. Canneal

Fig. 3: Measured and predicted execution times for Streamcluster and Canneal with batch size of 10 and window size of ∞.

III. Evaluations and Discussions

The proposed User-APMP has been evaluated extensively regarding to its prediction accuracy and runtime overheads. In this section, we first discuss the evaluation methodology and experimental setups. Then, the evaluation results are presented and discussed by considering varying batch and window sizes.

A. Evaluation Methodology and Setups

We consider eight (8) benchmark applications in PARSEC suite [12], including *streamcluster, blackscholes, bodytrack, canneal, facesim, ferret, swaptions* and *dedup*. These benchmarks were chosen with the consideration to represent a wide range of applications (such as CPU vs. memory intensive to be the representative workloads of graph and data analytic applications [3, 13]) and the duration of experiments.

These benchmark applications were compiled and run with the Ubuntu 16.04 environment on a virtual machine (VM) with 16 vCPUs and 16GB memory. The VM was created under the OpenStack installed on a server with dual Intel Xeon E5-2630 16-core processors and 128GB memory. Before the benchmark applications run on the VM, each of the three designed micro-benchmarks runs for around 3 seconds sequentially to probe the resource contention levels of the VM on the host machine. Then, the eight benchmark applications were executed on the VM with 16 worker-threads and their native input sets.

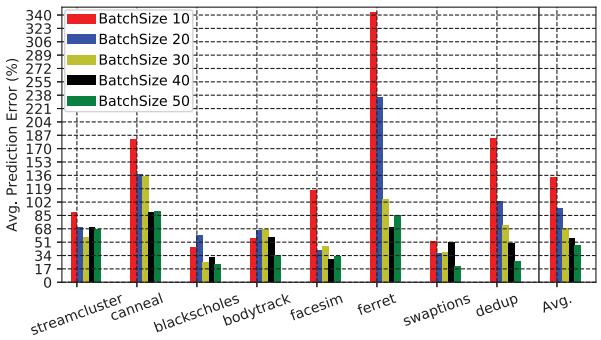
To introduce interference and emulate resource contentions, varying number of background VMs have been created on the same host machine during the executions of the benchmark applications. Specifically, after each 2-hour interval, a random number (up to 7) is generated to indicate the number of background VMs should be created for the next interval of 2 hours. Moreover, each background VM will randomly choose either a CPU or memory intensive synthetic application from iBench suite [19] to introduce the different levels of interference for the key components of the VM that runs the target benchmark applications.

With the randomly introduced resource contention from the background VMs and their applications, the micro-benchmarks and the selected PARSEC benchmark applications run repeatedly until 1,000 data tuples are collected for each benchmark. These data tuples will be utilized to train and evaluate User-APMP for each benchmark application, respectively, on its prediction accuracy and runtime overheads.

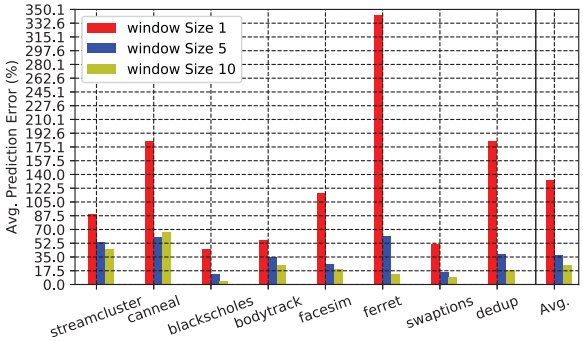
B. Exemplary Measured vs. Predicted Results

Before we present the evaluation results with varying batch and window sizes, Fig. 3 shows the measured execution times (the blue points) for two representative benchmarks, *Streamcluster* and *Cannel*, respectively. From the figure, we can see that their execution performance does change radically at runtime due to varying levels of resource contention on the host machine, where their execution times can vary up to 10 times! Therefore, to support cloud users for proper planning of their operations, it is crucial to get reasonably accurate performance prediction for the execution of their applications. Note that, for *Cannel*, the execution times for some iterations can be as low as around 30 seconds. This comes from the fact that, at the beginning of each 2-hour interval when the background VMs change, we stop the executions of the interfering applications for the first 5 minutes.

The figure also shows the predicted execution times of User-APMP with Lasso regression techniques for these two benchmark applications with the batch size of 10 and window size of ∞ (i.e., the predictive model is retrained after every 10 data points and each time all historical data tuples are utilized for model adaptations). Although it is hard to associate a predicated execution time of the benchmark applications with its corresponding measured one in the figure, we can clearly see that the predicted results have the same pattern (or trend) as that of the measured execution times. This also validates our hypothesis that the devised micro-benchmarks can properly assess the level of resource contention at runtime.

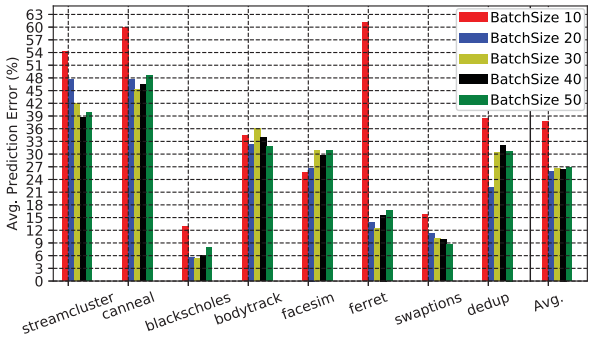


a. Varying batch sizes with window size of 1.

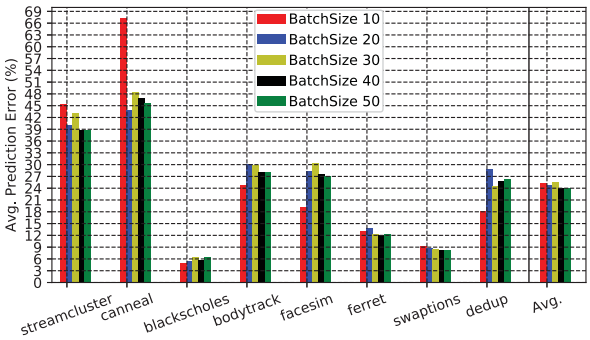


b. Varying window sizes for batch size of 10.

Fig. 4: Prediction errors of User-APMP for the benchmark applications with varying batch and window sizes.



a. Window size of 5



b. Window size of 10.

Fig. 5: Prediction errors of User-APMP with different batch sizes and larger window sizes.

C. Accuracy of User-APMP vs. Varying Batch/Window Sizes

As discussed in Section II, we adopted 2-degree polynomial regression technique for User-APMP. In particular, we train the predictive model in User-APMP with the Lasso regression [16] and have implemented it using the Scikit-learn library (version 0.19.2) [20]. We tuned the regression-based model with an alpha of 1 and a tolerance value of 0.001. To quantify the accuracy of the predicted results from the model, we define the *prediction error* for each data point of an application as:

$$Pred^{err} = \frac{|time_{measured} - time_{predicted}|}{time_{measured}} \quad (2)$$

where $time_{predicted}$ is the predicted execution time of the application using User-APMP with the current profiling data from the three micro-benchmarks and $time_{measured}$ denotes the measured execution time. We report the *average* prediction error all the data points that have a prediction result for each benchmark application.

Fig. 4a first shows the average prediction errors of User-PAMP for the benchmark applications with different batch sizes (i.e., 10, 20, 30, 40 and 50). Here, the window size is set as 1; that is, only the data tuples in the last batch are utilized to retrain the predictive model at runtime to predict the data tuples in the next batch. From the figure, we can see that, although having larger batch sizes reduces the adaptation frequency of the predictive model, more accurate model can be re/trained with more historical data tuples being used in the last batch, which generally results in smaller prediction errors. In particular, for *ferret*, the average prediction error is reduced from about 341% to 82% for the batch sizes of 10 and 50, respectively. The overall average prediction errors for all the benchmarks are reduced from 135% to 45% when the batch size increases from 10 to 50.

When batch size is set as 10, Fig. 4b shows the effects of varying window sizes on the accuracy of User-APMP for the benchmark applications. Clearly, when the window size increases from 1 to 5 (i.e., the number of historical data tuples for retraining the model increases from 10 to 50), the average

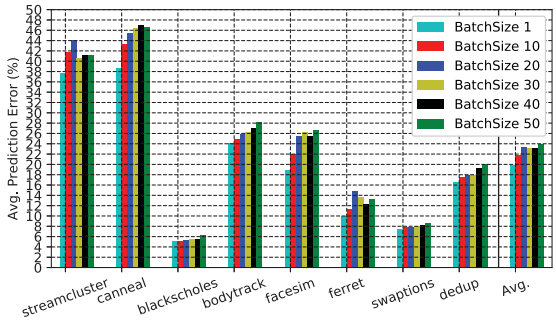


Fig. 6: Prediction errors of User-APMP: window size of ∞.

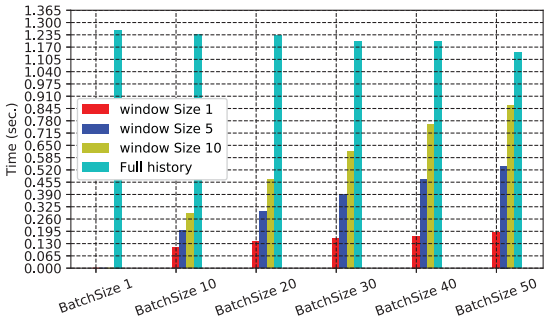


Fig. 7: Overhead for modeling/prediction per batch

prediction errors can be significantly reduced where the overall average for all applications can reduce from 135% to 39%. When the window size increases to 10, the prediction errors can be further reduced, but with relatively less magnitude. From the results, we can see that having more historical data tuples for re/training can generally improve the accuracy of the predictive model in User-APMP. However, such benefits normally reduce as more data tuples are included with larger window sizes.

Fig. 5 further shows the average prediction errors of User-APMP with different batch sizes for the cases of window sizes of 5 and 10, respectively. The results are inline with our previous observations. That is, the benefits of increasing window sizes diminish for larger batch sizes. In particular, when the window size is 10, we can see that the overall average prediction errors are almost the same (about 25%) for all different batch sizes. That is, once the number of data tuples in the training window reaches a certain number (e.g., 100), the benefit of having even more data tuples is very limited.

For the case of window size being ∞ (i.e., where all historical data tuples are used for re/training the predictive model), Fig. 6 shows the average prediction errors of User-APMP for the benchmark applications with different batch sizes. In particular, we include the case with the batch size of 1 to illustrate the benefit limit for increasing the frequency of adaptations. From the results, we can see that the prediction errors do decrease compared to the cases of window sizes being 5 or 10, but with very limited improvements. Moreover, even when the model is retrained after each iteration of executing the applications (for the case of batch size being 1), the overall prediction errors reduce less than 5%. However, as shown below, the overhead of User-APMP for the batch size of 1 can be very prohibitive and prevent it from being deployed at runtime.

D. Modeling and Prediction Overheads of User-APMP

In addition to its prediction accuracy, we report the runtime overhead of User-APMP, which is also an important evaluation metric for it being designed as an online predictor. First, Table I shows the overall time required for User-PAMP to train or retrain the predictive models and to predict all 1000 data points for the different settings of batch/window sizes at runtime.

TABLE I: Overall time (sec.) to model/predict all data.

Batch Size	Win. of 1	Win. of 5	Win. of 10	Win of ∞
1	-	-	-	1325.7
10	12.2	21.8	31.1	129.8
20	7.6	15.8	24.8	64.0
30	5.5	13.4	21.3	40.8
40	4.5	12.2	19.9	31.3
50	3.8	10.8	17.2	22.9

Not surprisingly, with increased batch sizes, the adaptation frequency decreases and it takes less overhead. On the other hand, when window size increases, more historical data tuples are utilized for model adaptation, which leads to higher overheads. In particular, for the case of batch size being 1 and window size of ∞, it can take more than 1325 seconds to process these 1000 data points. In comparison, for the case of window size being 5 and batch size of 50, it takes only 10.8 seconds (a reduction of more than $120X$), which however can result in almost the same level of prediction errors (with only up to 5% difference).

Fig. 7 further shows the average time used by User-APMP to retrain the model and predict the data points within each batch for different batch sizes. As the figure shows, the average time to retrain the model using a new batch and predict the execution time for the benchmark runs within a batch is every small. Even with a windows size of 10 and batch size of 50, the time to retrain and predict a new batch is less than 0.9 seconds. In the case of batch size 20 and window size 10, the time to retrain and predict a batch is only 0.5 seconds. This time is much smaller than the execution time of a PARSEC benchmark, which is at least 20 seconds on our VM. This low overhead demonstrates that User-APMP is efficient for online prediction.

Furthermore, Fig. 7 also shows that the overhead does increase with larger batch sizes and window sizes. In particular, when all past data are used to make predictions, the

overhead to retrain and predict a new batch increased to more than 1 seconds. This overhead can be significant for the PARSEC benchmarks with execution times less than 50 seconds. Consequently, the batch- and window-based approach employed by User-APMP is important for this prediction to be both accuracy and efficiency at runtime.

IV. Conclusions

Applications running in cloud environment may have performance variations due to contention caused by collocated VMs on the same host machine. Hence, it is important for cloud users to have accurate predictions on the executions of their application for them to make better planning of their operations and expenditure. In this paper, *User-APMP*, a user-level application performance modeling and prediction framework, based on micro-benchmarks and regression techniques is proposed. In User-APMP, a few micro-benchmarks are devised to probe the perceivable performance of CPU, memory and I/O components of a target VM. The predictive performance model of User-APMP is adaptively retrained at runtime with regression techniques based on such probe information from micro-benchmarks and in-place measured performance of an applications. In addition, the sliding-window technique is utilized to control the adaptation frequency and the number of historical data items to retrain the model. Our evaluation results show that, the micro-benchmarks can properly assess the resource contention levels of a VM in multi-tenant clouds. The prediction error of User-APMP generally decreases with higher adaptation frequencies and more historical data points, which however leads to higher runtime overhead. With only 100 data points, the average prediction errors can reach 25%.

References

[1] P. Leitner and J. Cito, "Patterns in the Chaos&Mdash;A Study of Performance Variation and Predictability in Public IaaS Clouds," *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 3, 2016.

[2] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," in *Proc. of Annual IEEE/ACM Int'l Symposium on Microarchitecture*, 2011.

[3] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," in *Proc. of Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[4] F. Romero and C. Delimitrou, "Mage: Online and Interference-Aware Scheduling for Multi-Scale Heterogeneous Systems," in *Proc. of Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[5] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proc. of the 40th Annual Int'l Symposium on Computer Architecture*, 2013.

[6] N. Mishra, J. D. Lafferty, and H. Hoffmann, "ESP: A Machine Learning Approach to Predicting Application Interference," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2017.

[7] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach," in *Proc. of ACM Symposium on Cloud Computing (SoCC)*, 2017.

[8] J. Scheuner and P. Leitner, "Estimating Cloud Application Performance Based on Micro-Benchmark Profiling," in *IEEE International Conference on Cloud Computing*, 2018.

[9] M. Mao and M. Humphrey, "Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows," in *Proc. of Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.

[10] R. Begam, W. Wang, and D. Zhu, "Virtual machine provisioning for applications with multiple deadlines in resource-constrained clouds," in *Proc. of the IEEE Int'l Conference on High Performance Computing and Communications (HPCC)*, 2017.

[11] R. Begam, H. Moradi, W. Wang, and D. Zhu, "Flexible vm provisioning for time-sensitive applications with multiple execution options," in *Proc. of the IEEE Int'l Conference on Cloud Computing (CLOUD)*, 2018.

[12] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[13] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 89–100, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2490301.2451126

[14] J. Kiefer, J.; Wolfowitz, "Stochastic Estimation of the Maximum of a Regression Function," *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952.

[15] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

[16] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[17] A. E. Hoerl and R. W. Kennard, "Ridge regression: applications to nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 69–82, 1970.

[18] T. Zhang, "Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms," in *Proc. of Int'l Conference on Machine Learning*, 2004.

[19] C. Delimitrou and C. Kozyrakis, "iBench: Quantifying interference for datacenter applications," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[20] Scikit-learn, http://scikit-learn.org/stable/.