Maximum Sustainable Throughput Prediction for Large-Scale Data Streaming Systems

Abstract—In cloud-based stream processing services, the maximum sustainable throughput (MST) is defined as the maximum throughput that a system composed of a fixed number of virtual machines (VMs) can ingest indefinitely. If the incoming data rate exceeds the system's MST, unprocessed data accumulates, eventually making the system inoperable. Thus, it is important for the service provider to keep the MST always larger than the incoming data rate by allocating a sufficient number of VMs. In this paper, we propose a cost-effective framework to predict MST values for a given number of VMs for stream processing applications with various scalability characteristics. Since it may be difficult to find one prediction model that works well for various stream processing applications, we first train several models using linear regression for each application. We then select the best-fitting model for the target application through the evaluation of extra MST samples. To save cost and time to collect MST samples while achieving high prediction accuracy, we statistically determine the most effective set of VMs within a budget. For evaluation, we use Intel's Storm benchmarks running on Amazon EC2 cloud. Using up to 128 VMs, experiments show that the models trained by our framework predict MST values with up to 15.8% average prediction error. Further, we evaluate our prediction models with simulation-based elastic VM scheduling for a realistic data streaming workload. Simulation results show that with 20% over-provisioning, our framework is able to achieve less than 0.1% SLA violations for the majority of test applications. We save 36% cost compared to a static VM scheduling that covers the peak workload to achieve the same level of SLA violations.

Index Terms—Cloud computing, maximum sustainable throughput, performance prediction, data streaming.

1 Introduction

The need for real-time stream data processing is ever increasing as we are facing an unprecedented amount of data generated at high velocity. Upon the arrival of a stream event, we want to process it as quickly as possible to timely react to events such as aircraft airspeed sensor failure [1] or unusually high CPU usage in data centers [2]. Traffic management [3], [4] and sensor data processing from Internet-of-Things (IoT) devices [5], [6], [7] are also common real-time stream processing applications.

To process these fast data streams in a scalable and reliable manner, a new generation of stream processing systems has emerged: systems such as Storm [8], [9], Flink [10], [11], Samza [12], and Spark Streaming [13], [14] have been actively used and developed in recent years. Cloud computing can add on-demand elasticity to these stream processing systems to deal with fluctuating computing demand using *autoscaling* [15]. To guarantee a service level agreement (SLA) in terms of application performance, we need a prediction model that connects the number of virtual machine (VM) instances and application performance so that the autoscalers can estimate and provision the right number of VMs.

We define *maximum sustainable throughput* (MST) as the maximum throughput that the stream processing system can ingest indefinitely for a given number of VMs. It is an application performance metric that is useful from the data processing service provider's perspective. If the incoming data rate exceeds the system's MST, unprocessed data accumulates, eventually making the system inoperable. By dynamically allocating and deallocating VMs, service providers can keep the MST larger than the incoming data rate to maintain stable service operation while only paying VMs that are needed.

To realize this MST-based elastic stream processing in a cost efficient manner, we need an MST prediction model for a given stream application and number of VMs. However, most recent elastic stream processing studies primarily focus on guaranteeing latency [16], [17], [18], [19]. Therefore, these studies only use prediction models for latency. ElasticStream [20] is the only elastic streaming system that estimates maximum throughput. It uses a model that is linear in the number of VMs, which is not realistic for all applications, as we show in this work.

Due to the complex nature of distributed data processing, it is not always feasible to model application performance analytically, without any observations of application performance. Recent works have used supervised learning to model the performance of distributed batch processing applications [21], [22], [23]. These works collect performance metric samples from actual application runs and train prediction models using regression. Among them, our work was inspired by Ernest [22], which models job completion time for Apache Spark's batch processing as a polynomial of the input data size and number of machines. It uses training samples obtained from a few machines to predict the performance for a larger number of machines. We take a similar approach to Ernest to predict MST values for stream processing applications. While Ernest uses a single prediction model, we note that there are cases where a single model cannot be trained to work well for multiple stream processing applications.

In this work, we propose a cost-effective framework to

• *S. Imai, S. Patterson, and C. A. Varela are with the Department of Computer Science, Rensselaer Polytechnic Institute of Technology, Troy, NY, 12180.*
E-mail: {imais,sep,cvarela}@cs.rpi.edu

predict MST values for stream processing applications with various scalability characteristics. Since it may be difficult to find one prediction model that works well for all of the applications, we first train several models using linear regression. We then select the best-fitting model for the target application through the evaluation of extra MST samples. To save cost and time to collect MST samples while achieving high prediction accuracy, we statistically determine the most effective set of VMs within a budget. For evaluation, we use Intel's Storm benchmarks [24] running on Amazon EC2 cloud. Using up to 128 VMs, experiments show that the models trained by our framework accurately predict MST. We also simulate elastic VM scheduling with the trained models using a realistic data streaming workload and confirm the cost-effectiveness of our approach.

The rest of the paper is organized as follows. In Section 2, we present related work on performance models used in elastic data processing. In Section 3, we describe the concept of maximum sustainable throughput and its measurement method. Section 4 presents our MST prediction framework. Section 5 shows the evaluation of our models' prediction accuracy, and Section 6 shows the cost-effectiveness of the proposed framework in simulation-based elastic stream processing. Section 7 discusses the results of experiments. Finally, we conclude the paper in Section 8.

2 Related Work

In stream processing, one of the most common performance metrics is latency. There are several proposals for topology-aware latency models [16], [17], [18], [19]. Li et al. propose a topology-aware average latency model that uses thread-level statistics such as the average tuple processing latency and tuple transfer latency [16]. Heinze et al. try to minimize latency spikes due to stateful operator migration when scaling stream applications [17]. Their model considers an application topology consisting of multiple operators. It enumerates operators that need to be paused and restarted for migration, and it estimates latency including operator pause time. Queuing theory is also used for latency prediction. Nephele (a prototype of Apache Flink) models each processing unit to be a G/G/1 single server system with a degree of parallelism [18]. DRS models multiple threads derived from a processing unit to be a M/M/c multiple server system [19].

Another important metric for stream processing is throughput. There are some proposals that do not explicitly model throughput but try to achieve high throughput by improving task scheduling on a fixed number of VMs [25], [26]. R-Storm implements a resource-aware task scheduler on top of Storm [25]. It tries to increase throughput by maximizing resource utilization and co-locating tasks communicating with each other. Similarly, Fischer and Bernstein use graph partitioning to minimize network communication across different machines and also to minimize load imbalance [26]. To alleviate excessive incoming workload for stream processing running on a fixed number of VMs, techniques such as random sampling [27] and backpressure [28] have been proposed. Random sampling randomly picks up events from a stream to reduce the amount of data to process, but the answers are approximate. Backpressure is a mechanism in which the data receiver sends a signal to request the data sender to halt its data transmission.

There have been a number of research projects on predicting job completion time for batch processing. There are prediction models specifically designed for MapReduce. AROMA [21] takes a purely data-driven approach, in which it combines clustering of resource usage profiles and regression with Hadoop MapReduce-specific variables. ARIA [29] shows an analytically-designed job completion time model based on the general map-reduce programming framework [30]. Ernest [22] models job completion time for Spark's batch processing based on computation and communication topology. Ernest represents job completion time as a polynomial of the number of machines and the size of input data. Compared to AROMA and ARIA, the model used in Ernest only requires target scaling factors (*i.e.*, input data size and number of machines), and therefore it is more widely applicable. Our approach is similar to Ernest as we use the number of machines as the only variable in our models. However, Ernest assumes a single model, whereas our framework uses multiple models and selects the model that is expected to give the least prediction error for each test application. This helps us predict performance for both linearly and non-linearly scaling applications, as we show later in this paper.

To safely scale up a cluster to process fluctuating workload, it is important to know the maximum processing capacity of the cluster. Metrics similar to MST has been used for web service applications [31], [32], but maximum throughput has received less attention compared to latency in stream data processing. To the best of our knowledge, ElasticStream [20] is the only elastic stream processing system that tries to maintain the cluster's maximum throughput to handle fluctuating input data rates through automated VM allocation. It uses a linear model to predict maximum throughput; however, there are applications for which maximum throughput is not linearly scalable as we show in Section 5. Unlike ElasticStream, we model MST for both linearly and non-linearly scalable applications.

We presented a preliminary version of MST prediction framework first time in [33]. In this work, we extend our previous work by adding the following new research efforts:

- A model selection method from several candidate models.
- Experiments with Intel's Storm benchmarks [24] and a stream machine learning application from SAMOA [34]. We evaluate prediction results of MST values for up to 128 VMs (previously up to 32 VMs).
- More realistic elastic VM scheduling simulation with VM allocation overhead (previously no overhead was considered).

3 Maximum sustainable throughput

In this section, we explain the motivating topic of this paper: maximum sustainable throughput. We first show technical background including a common data processing environment in Section 3.1, and we then introduce the concept of maximum sustainable throughput (MST) and how to measure it in Section 3.2. We also show how we can use MST in SLAs for stream processing systems in Section 3.3.

3.1 Technical Background of Stream Processing

3.1.1 Common Stream Processing Environment

Figure 1 shows a commonly used stream processing environment, which works for frameworks including Storm [8], Samza [8], Flink [10], [11], and Spark Streaming [13]. Data streams flow from left to right, starting from the data producer to the data store. The data producer sends events at the input data rate of $\lambda(t)$ Mbytes/sec, and they are appended to message queues in Kafka. Since the input data rate fluctuates over time, λ is a function of time. Kafka is a message queuing system that is scalable, high-throughput, and fault-tolerant [35] and is supported by major stream processing frameworks. The stream processing system pulls data out of Kafka as quickly as it can at the throughput of $\tau(m)$ Mbytes/sec, where m is the number of VMs. Note that there can be multiple producers and consumers working simultaneously to avoid Kafka from being a bottleneck. After the stream processing system processes events, it optionally stores results in the data store (*e.g.*, a file system or database).

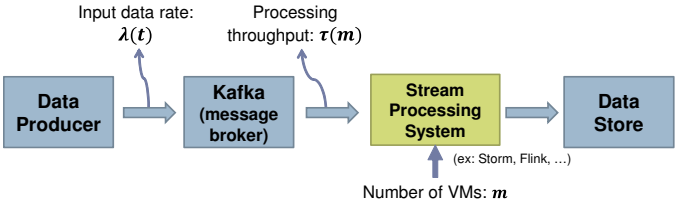


Fig. 1. Common stream processing environment.

3.1.2 Scaling Policy

To scale up the execution of stream processing applications, We define an application-independent scaling policy. We assume that the user writes stream applications in the form of connected processing units (see the logical topology shown on the left of Figure 2 as an example) and each of the processing units can be duplicated to an arbitrary number of threads when they are deployed on the cluster. Each processing unit receives events from the preceding units and emits processing results to the following units. The goal of the scaling policy is to assign one thread per virtual CPU (vCPU). Actual binding between threads and machines is up to the stream processing system's task scheduler. Parameters for the scaling decision are shown as follows:

- n: Number of processing units.
- m: Number of virtual machines.
- γ: Number of vCPUs per virtual machine.

Given these parameters, we can compute duplication factor (or parallelism) d as follows.

$$d = \max(1, \lfloor (m \cdot \gamma)/n \rfloor). \quad (1)$$

This rule is a generalization of technique used in Yahoo Streaming Benchmarks [36]. When $n = 3, \gamma = 2$, examples of scaling to $m = 3$ and $m = 5$ are shown in Figure 2.

The stream processing system's worker nodes interact with Kafka and the data store as shown in Figure 3 . While we fix the master node of the stream processing system, we scale up the worker nodes by simply adding new VM

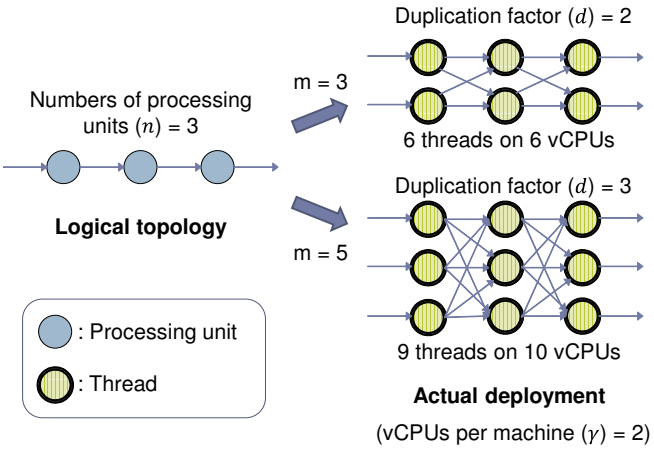


Fig. 2. Examples of scaling a topology with three processing units ($n = 3$) to three and five machines ($m = 3$ and 5) respectively. Each machine has two vCPUs ($\gamma = 2$).

instances of the same type. We assume Kafka and the data store have enough resources to not become a bottleneck. A series of messages (called a *topic*) in Kafka is stored across p partitions, which are consumed by the worker nodes in parallel. Since the number of partitions defines the parallelism of data processing, the number of threads consuming data from Kafka needs to match the number of partitions.

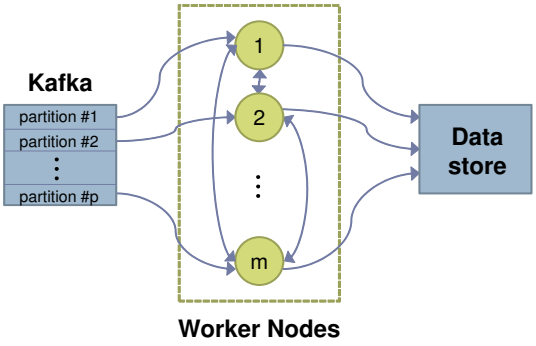


Fig. 3. Interactions between worker nodes and surrounding entities in the stream processing environment.

3.2 Measuring MST

As we have defined in Section 1, MST is a metric to quantify the maximum processing capacity for the stream processing system. In this sub-section, we show how we measure the true MST of the stream processing system for a given number of VMs.

In the common stream processing environment in Figure 1, the processing throughput $\tau(m)$ never exceeds the input data rate $\lambda(t)$ since the stream processing system cannot process data unless they are provided by the data producer. To obtain the true performance of the target stream processing system, we have to ensure that the condition $\lambda(t) > \tau(m)$ always holds so that the data is always

available to process for the stream processing system. However, the data producer or Kafka can be a bottleneck and hinder the stream processing system from processing the data at its maximum speed. To avoid this issue, we can load enough data to Kafka in advance as shown in Figure 4, and effectively simulate the condition $\lambda(t) > \tau(m)$.

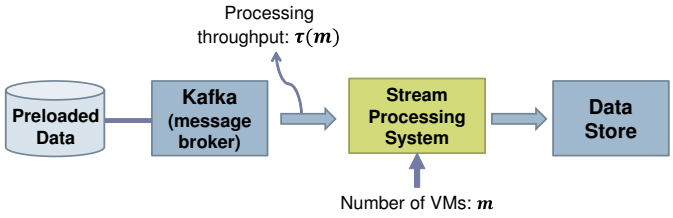


Fig. 4. Maximum sustainable throughput measurement environment.

Figure 5 shows an example of throughput transition over time for a stream application that processes web access logs (*i.e.*, *Unique Visitor* in Section 5.1), observed in the environment shown in Figure 4. In this setting, the stream processing system pulls data from Kafka as quickly as possible. The throughput gradually grows until 100 seconds and then converges. After the convergence, we start sampling the throughput, and that is the MST we measure. In this example, MST is around 42 Mbytes/sec. To detect convergence of the throughput, we use the following *K out of N* method: we keep monitoring the latest N samples and if K samples are within $\pm\delta\%$ from the previous samples, we determine that the throughput is converged. For the experiments in this work, we used $N = 5, K = 4, \delta = 5\%$. As long as enough data is pre-loaded in Kafka, the stream processing system is able to process the data indefinitely up to the rate of MST.

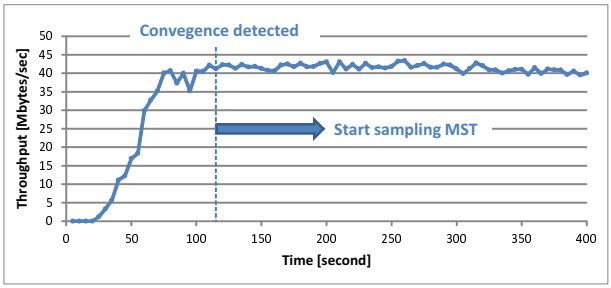


Fig. 5. Convergence of throughput for a web access log processing streaming application.

3.3 Performance Objectives and Service Level Agreements

The following performance objectives are commonly used in event processing systems [37]: 1) maximize input throughput, 2) maximize output throughput, 3) minimize average latency, 4) minimize maximal latency, 5) latency leveling, 6) real-time constraints. Note that latency leveling refers to minimizing the variance of the latency. The SLAs used in previous works [16], [17], [18], [19] focusing on latency are equivalent to #6 real-time constraints.

In this work, the SLA of interest is related to #1 maximize input throughput. For example, the objective is for the system to keep up with the input data rate without completely saturating the system capacity. So, the SLA is as follows:

$$\lambda \leq MST, \tag{2}$$

where λ is the input data rate and MST is the MST of the system. Since latency and throughput are related and important metrics in stream data processing, performance objectives and SLAs may consist of multiple metrics in practice [37]. If the SLA in (2) is not satisfied, incoming messages start accumulating in Kafka and thus it increases end-to-end processing latency. If this situation persists, eventually all the system resources are consumed and the system may become inoperable in the worst case.

4 MST PREDICTION FRAMEWORK

In this section, we propose an MST prediction framework that is designed with the following considerations.

1) **Application as a black box**: We see stream applications as a black box so that our framework is generally applicable to a wide range of stream processing frameworks.
2) **Default task scheduler**: Following the first assumption, we use a default task scheduler and do not control task scheduling.
3) **Homogeneous VM type**: We only use a single VM type, namely m4.large on Amazon EC2. If different stream processing tasks have different resource usage requirements, there may be room for performance improvement by optimizing the task scheduling of these tasks with heterogeneous VM types (*e.g.*, maximizing resource utilization or minimizing inter-machine communication). However, since we choose to use a default task scheduler and do not aim for performance optimization, homogeneous VM types are sufficient.
4) **Minimizing cost for training**: We assume that users want to minimize the time and cost for collecting training samples.

In the following sub-sections, we show the details of the proposed framework. We first explain linear regression in Section 4.1, give an overview in Section 4.2, and describe the the following building blocks of the framework in order: MST prediction models (Section 4.3), VM subset selection (Section 4.4), and model training and selection (Section 4.5).

4.1 Linear Regression

We use linear regression [38] as a method to model the relationship between MST (dependent variable: y) and the number of VMs m (independent variable: $\mathbf{x}$). Given a training dataset $\mathcal{D}_{\text{train}} = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, linear regression looks for the optimal weight vector $\mathbf{w}$ in a prediction model $h(\mathbf{x}) = \mathbf{w}^{\mathbf{T}}\mathbf{x}$ that minimizes the following mean square error:

$$E(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n) - y_n)^2. \tag{3}$$

By taking the gradient of Equation (3), we can analytically obtain the optimal $\mathbf{w}$. Training is the process to obtain the optimal $\mathbf{w}$ for the prediction model $h(\mathbf{x})$ given the training

dataset $\mathcal{D}_{\text{train}}$, whereas testing is to evaluate the trained model on a new test dataset.

4.2 Framework Overview

Figure 6 shows an overview of the proposed framework, which consists of two phases as follows.

Phase 1: In this phase, we determine the most effective set of VMs to obtain training samples. First, we collect MST samples from representative benchmark applications in terms of resource usage patterns. Next, we enumerate subsets of a candidate training VMs set $\mathcal{V}_{\text{train}}$ to create various training sets. We train the models using linear regression with each training set and select the best VM subset $\hat{\mathcal{S}}$ with the lowest test prediction error. We do this process once offline.

Phase 2: In this phase, we train a new test application that the user wants to predict the MST values. We collect training MST samples only for the VM subset $\hat{\mathcal{S}}$ determined from Phase 1. After training the models with the collected samples, we obtain extra validation samples that are not included in $\hat{\mathcal{S}}$ and select the model with lower validation error. We run this process per test application.

4.3 MST Prediction Models

In this sub-section, we design two MST prediction models based on the stream processing environment we have shown in Section 3.1. We assume that a combination of the following factors determines the MST for a given number of VMs m.

1) *Parallel processing gain*: Performance improves as m increases.
2) *Input/output distribution overhead*: Performance decays linearly as m increases due to event transmissions from Kafka to the m worker nodes and also due to result transmissions from the m worker nodes to the data store.
3) *Inter-worker communication overhead*: Performance decays quadratically as m increases due to the $m(m-1)$ communication paths between m worker nodes.

Based on these factors, we create the following two models.

Model 1: As shown in Equation (4), This model predicts MST as a function of the number of VMs m. It is defined as the inverse of event processing time, which is represented as a polynomial of the number of VMs m. The terms have the following meanings: serial processing time (w_0), parallel processing time (w_1), input/output distribution time (w_2), and inter-worker communication time (w_3). Note that all the weights are restricted to be non-negative (*i.e.*, $w_i \geq 0, i = 0, ..., 3$).

$$MST^{(1)}(m) = \frac{1}{Time(m)} = \frac{1}{w_0 + w_1 \cdot \frac{1}{m} + w_2 \cdot m + w_3 \cdot m^2}. \quad (4)$$

This model was inspired by Ernest [22], which models job completion time for Spark's batch processing jobs by considering computation and communication topology. Ernest represents job completion time as a polynomial of the number of machines and the size of input data. We take the inverse of processing time to model throughput.

Model 2: This model is a simple polynomial equation as shown in Equation (5). The terms have the following meanings: base throughput (w_0), parallel processing gain (w_1), inter-worker communication overhead (w_2). All the weights are restricted to non-negative (*i.e.*, $w_i \geq 0, i = 0, 1, 2$), but we add a minus sign for w_2 to account for negative impact of the inter-worker communication.

$$MST^{(2)}(m) = w_0 + w_1 \cdot m - w_2 \cdot m^2. \quad (5)$$

For Models 1 and 2, depending on the values of specific weights after training (*i.e.*, w_2 and w_3 for Model 1 and w_2 for Model 2), predicted MST can have a peak at certain VMs. This means that we can have more than two different VM counts to obtain a certain value of MST. Since it is not reasonable to choose the larger VM counts when the smaller one can achieve the same MST, we can effectively see that the predicted MST flattens out after its peak. Taking into this "peak effect" consideration, we assume MST has a constant value after its peak and compute prediction error under this assumption.

4.4 Phase 1: VM Subset Selection

In this phase, we statistically determine the most effective subset of VM counts for training in terms of prediction error by exhaustive search. We first describe a method to select such subset and then show the results of selected VM subsets.

4.4.1 VM Subset Selection Method

As part of Phase 1, we perform the following steps to find the best VM subset.

Step 1. Collecting MST samples: First, we run a set of benchmark applications $\mathcal{A} = \{a_1, a_2, ...\}$ on each VM count in $\mathcal{V} = \{m_1, m_2, ...\}$ for K times per application. $\mathcal{V}$ contains the number of VM instances, for which the user might need to predict the MST (*e.g.*, up to 128 VMs). We collect MST samples using the MST measurement method in Section 3.2. After collecting MST samples, we normalize the collected samples by the maximum MST value for each application to avoid bias towards one specific application when computing test prediction error in Step 2.

Step 2. Select the best VM subset: Let $\mathcal{S}$ be a subset of $\mathcal{V}$ used to train Models 1 and 2. Our goal in this step is to find the best subset that gives the lowest average prediction error over all applications and models. We want to collect training samples at a low cost, and so we define the maximum number of VMs to search, M_{train}, as shown in Figure 7. Let $\mathcal{V}_{\text{train}}$ be the candidate set of training VM subsets to choose $\mathcal{S}$ from: $\mathcal{V}_{\text{train}} = \{m_i \mid m_i \leq M_{\text{train}}, m_i \in \mathcal{V}\}$. Since there are many possible way to choose such subset $\mathcal{S}$ from $\mathcal{V}_{\text{train}}$, we enumerate all possible subsets of VM counts that have c elements in $Subsets$. The number of the enumerated subsets is equal to $\binom{|\mathcal{V}_{\text{train}}|}{c}$. For each $\mathcal{S} \in Subsets$, we create a training dataset and use the same training data set to train all the models. Using the trained models, we compute the Root Mean Square Error (RMSE) over all applications and models as shown in Equations (6)-(8). In Equation (6), we compute the sum of square error (SSE) between actual and predicted MST values: $MST(a_i, m, k)$ is the k-th actual MST sample obtained for application a_i running on m VMs, and

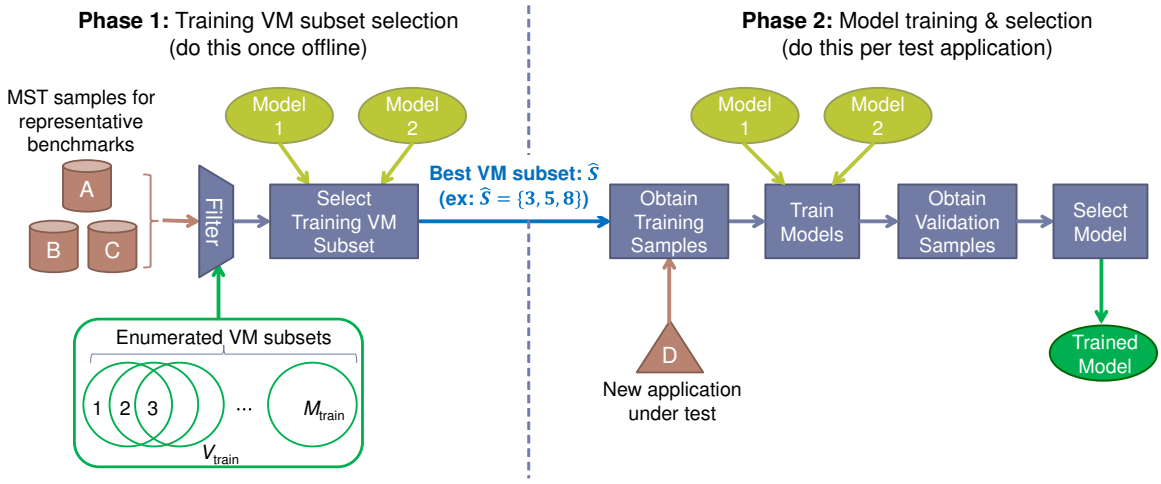


Fig. 6. Overview of the MST prediction framework.

$MST^{(j)}(a_i, \mathcal{S}; m)$ is a predicted MST value for m VMs using Model j trained with a dataset created from a VM subset $\mathcal{S}$ for application a_i. Finally, we select the best subset $\hat{\mathcal{S}}$ in terms of prediction error that is computed from test samples that are measured for $\mathcal{V} - \mathcal{S}$, as shown in Equation (8). Note that increasing c leads to exponential growth in the size of $Subsets$. Thus, we start from a small number for c and gradually increment it until the error becomes low enough.

$$SSE(\mathcal{S}) = \sum_{a_i \in \mathcal{A}} \sum_{j=1}^{2} \sum_{k=1}^{K} \sum_{m \in \mathcal{V}-\mathcal{S}} \left(MST(a_i, k, m) - MST^{(j)}(a_i, \mathcal{S}; m) \right)^2 \quad (6)$$

$$\hat{\mathcal{S}} = \underset{\mathcal{S} \in Subsets}{\operatorname{argmin}} \; RMSE(\mathcal{S}) \quad (7)$$

$$= \underset{\mathcal{S} \in Subsets}{\operatorname{argmin}} \sqrt{\frac{SSE(\mathcal{S})}{|\mathcal{A}| \cdot 2 \cdot K \cdot |\mathcal{V} - \mathcal{S}|}}. \quad (8)$$

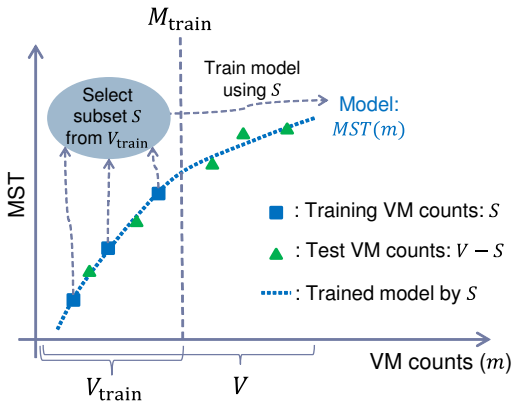


Fig. 7. Selecting VM count subset consists of less than or equal to M_{train} VMs.

4.4.2 VM Subsets Selection Results

We collect MST data from experiments and apply the VM subset selection method presented in Section 4.4 to the collected data.

Experimental settings: We run the following three *simple resource benchmarks* with Apache Storm version 0.10.1 [8] on Amazon EC2.

- *Word Count*: CPU intensive, typical word count for text inputs.
- *SOL* (*i.e.*, Speed-Of-Light): Network intensive, received events are transferred to the following processing units immediately without any processing.
- *Rolling Sort*: Memory intensive, received events are accumulated in a ring buffer and are sorted every x seconds.

We choose these three benchmarks since they have representative orthogonal resource usage patterns (*i.e.*, CPU, network, and memory intensive), and thus, the training samples obtained from these benchmarks can be generalizable to other applications.

As described in Section 3.2, we pre-load data in Kafka and let Storm pull the data as quickly as possible. After we start the application, we wait until throughput converges (see Section 3.2 for the convergence criteria) or until 90 seconds have passed. Subsequently, we monitor traffic going from Kafka to Storm slave nodes for 20 seconds and compute the throughput as MST. For traffic monitoring, we modified tcpdump [39] to enable monitoring network traffic between specific nodes [1]. Kafka offers various metrics including outgoing throughput through the Java Management Extensions interface; however, since it only provides one-minute moving average values and takes longer to converge, we decided to monitor raw network traffic using tcpdump. After the sampling, we shutdown the application and wait for 10 seconds for the next round of sampling. We repeat this process 6 times (*i.e.*, $K = 6$) for all three simple resource benchmarks with the following VM counts up to 128 VMs (EC2 instance type: m4.large):

$$\mathcal{V} = \{1, 2, 3, ..., 7, 8, 12, 16, 24, 32, 48, 64, 80, 96, 128\}. \quad (9)$$

After we collect MST samples in the environment mentioned above, we apply the VM subset selection method

1. Available at https://github.com/imais/tcpdump.

in Section 4.4.1 to the MST samples with the following parameters:

- Combinations to search: $c \in \{3, 4, 5\}$.
- Maximum number of VMs to include in the training data set: $M_{\text{train}} \in \{5, 6, 7, 8, 12, 16, 24, 32\}$.

We could increase M_{train} up to 128 VMs and potentially achieve good prediction results; however, this would incur high time and cost penalties. Therefore, we limit M_{train} to 32 VMs (= 25% of 128 VMs).

Selected VM subsets: Table 1 shows the best VM subset $\hat{\mathcal{S}}$ and prediction error in the RMSE for each M_{train}. We get a reasonably low error of 0.08, considering the fact that the range of MST values is $[0, 1]$ after normalization. Since enumerated subsets created from a higher M_{train} contains all the elements of the ones created from a lower M_{train}, errors monotonically decrease as we increase M_{train}. For the evaluation of Phase 2, we assume that the user have enough budget to run up to 24 VMs, and we choose the best subset $\hat{\mathcal{S}} = \{3, 4, 6, 8, 24\}$ when $M_{\text{train}} = 24$ or 32. Since we use this subset to collect training samples in Phase 2, hereafter we refer to this best subset as S_{train}.

Figure 8 shows prediction results with S_{train}. Model 1 fits better than Model 2 for Word Count, whereas Model 2 fits better than Model 1 for Rolling Sort. For SOL, both models fit equally well. These results show that a single model is not sufficient to capture the performance of all applications. Therefore, we need to choose a better-fitting model for each application from Models 1 and 2.

TABLE 1
Best VM subsets $\hat{\mathcal{S}}$ and prediction errors in RMSE for variable maximum VM counts ($M_{\text{train}} = 5, ..., 32$).

M_{train}	Best subset: $\hat{\mathcal{S}}$	RMSE
5	$\{2, 4, 5\}$	0.2864
6	$\{2, 4, 5\}$	0.2864
7	$\{2, 4, 5, 7\}$	0.1394
8	$\{2, 4, 5, 7\}$	0.1394
12	$\{2, 4, 5, 7\}$	0.1394
16	$\{6, 8, 16\}$	0.1263
24	$\{3, 4, 6, 8, 24\}$	0.0813
32	$\{3, 4, 6, 8, 24\}$	0.0813

4.5 Phase 2: Model Training & Selection

In Phase 2, we first take a new test application from the user, which is subject to performance prediction. We obtain training MST samples only for a selected subset $\mathcal{S}_{\text{train}}$ from Phase 1, and we train both Models 1 and 2 using the the same set of training samples.

Followed by the model training, we compare trained Models 1 and 2 using some extra *validation* samples to estimate which model is more accurate. Then, we select the model with the lower validation error as the final output of Phase 2.

Validation data points selection: We collect validation MST samples for VMs where their predicted MST values by both models start to diverge. If $MST^{(1)}(m)$ and $MST^{(2)}(m)$ intersect each other at one or more positive real numbers of m, we take the ceiling of these numbers and put them in a set $\mathcal{I} = \{m_1, m_2, ..., m_I\}$, $m_i < m_{i+1}$. If the models do not intersect, $\mathcal{I} = \emptyset$. We get these intersecting points analytically if possible. If not, we can numerically estimate these intersecting points by plugging in multiple integers for m. Assuming both models are well trained up to the maximum VM counts in $\mathcal{S}_{\text{train}}$, we should validate models with VMs larger than any VM counts in $\mathcal{S}_{\text{train}}$. Also, to make sure there is some discrepancy between the two models, we introduce a discrepancy threshold τ. Based on these ideas, we determine validation data points as shown in Algorithm 1. First, we filter the intersecting VMs set $\mathcal{I}$ with the maximum VM counts in $\mathcal{S}_{\text{train}}$ and store the filtered VMs set in $\mathcal{I}'$ (Line 3). Even if $\mathcal{I}$ is empty, we still want to search for divergent VMs starting from $m_{\max}(\mathcal{S}_{\text{train}}) + 1$ (Line 6). To search for one divergent VM count per element in $\mathcal{I}'$, we loop through $\mathcal{I}'$ (Line 9-24). In Line 18, we compute the discrepancy ratio between the predicted values of Models 1 and 2. If it is greater than τ, we assume that there is enough discrepancy between the two models and add corresponding m to the validation set S_{val}. Note that we need to obtain S_{val} for each test application in Phase 2 whereas we obtain S_{train} only once in Phase 1.

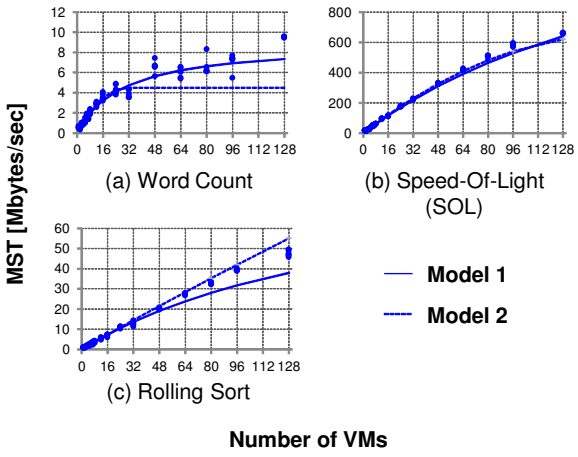


Fig. 8. MST prediction results using the best VMs subset: $\{3, 4, 6, 8, 24\}$. X-axis: number of VMs. Y-axis: MST [Mbytes/sec].

Model selection through validation error: Once we get S_{val}, we obtain MST values for the numbers of VMs in S_{val} and compute validation errors in the RMSE for both models. Finally, we select the model with the lowest validation error. If S_{val} is empty, which means the discrepancy between the two models never goes above τ, so we select the model with the lowest training error.

5 Evaluation of MST Prediction

In this section, we evaluate the MST prediction models that are trained and selected by our framework.

5.1 Experimental Settings

We use the following *typical use-case benchmarks* from Intel Storm Benchmarks [24].

- *Grep*: Match a given regular expression with text inputs and count the number of matched lines of texts.
- *Rolling Count*: Count the number of words and output the word counts every 60 seconds.

Algorithm 1: Selection of validation data points

input : $\mathcal{I}$: intersecting VMs set, $\mathcal{S}_{\text{tran}}$: training VMs subset, τ: discrepancy threshold
output: $\mathcal{S}_{\text{val}}$: set of VMs for validation

```
// Filter I with the max VM counts in S_train
```
$m_{\max}(\mathcal{S}_{\text{train}}) = \max_{m_j \in \mathcal{S}_{\text{train}}}(m_j)$;
if $\mathcal{I} \neq \emptyset$ **then**
  $\mathcal{I}' = \{m_i \mid m_i > m_{\max}(\mathcal{S}_{\text{train}}), m_i \in \mathcal{I}\}$;
end
else
  $\mathcal{I}' = \{m_{\max}(\mathcal{S}_{\text{train}}) + 1\}$;
end
$\mathcal{S}_{\text{val}} = \emptyset$;
```
// Search for divergent VMs
```
for $i = 1, ..., |\mathcal{I}'|$ **do**
  // m_i: i-th element of $\mathcal{I}'$
  $m_{\text{start}} = m_i$;
  if $i = |\mathcal{I}'|$ **then**
    $m_{\text{end}} = MAX\_VMS$;
  end
  else
    // m_{i+1}: $(i+1)$-th element of $\mathcal{I}'$
    $m_{\text{end}} = m_{i+1} - 1$;
  end
  for $m = m_{\text{start}}, ..., m_{\text{end}}$ **do**
    $d(m) = \frac{|MST^{(1)}(m) - MST^{(2)}(m)|}{\min(MST^{(1)}(m), MST^{(2)}(m))}$;
    if $d(m) > \tau$ **then**
      $\mathcal{S}_{\text{val}} = \mathcal{S}_{\text{val}} \cup \{m\}$;
      break;
    end
  end
end
return $\mathcal{S}_{\text{val}}$;

- *Unique Visitor*: Count the number of unique visitors to websites from web access logs.
- *Page View*: Count the number of page views per website from web access logs.
- *Data Clean*: Filter the logs with 200 HTTP status code from web access logs.

Also, from Apache SAMOA [34], we use the following stream machine learning application.

- *Vertical Hoeffding Tree (VHT)* [40]: Incrementally create a decision tree from data streams.

For Grep and Rolling Count, we use the texts from The Adventures of Tom Sawyer [41] as inputs. For Unique Visitor, Page View, and Data Clean, we synthesize web access logs that contain access to 100 randomly generated websites. For VHT, we use a randomly generated data set with 200 features and 10 classes.

For each benchmark, we perform the following steps corresponding to Phase 2 described in Section 4.5. First, we measure MST values for $\mathcal{S}_{\text{train}} = \{3, 4, 6, 8, 24\}$ using the same environment as described in Section 4.4.2. Next, we train Models 1 and 2 with samples measured for $\mathcal{S}_{\text{train}}$. After training, we further obtain more samples for $\mathcal{S}_{\text{val}}$ using Algorithm 1 with a discrepancy threshold of $\tau = 0.10$ for validation. Once we select a model for each benchmark, we obtain new measurements for all VMs in $\mathcal{S}_{\text{train}}$ and $\mathcal{S}_{\text{val}}$. Finally, we evaluate the predication error of the models using the newly measured MST values.

5.2 MST Prediction Results

Table 2 shows the weights of Models 1 and 2 after training. From the table, we can see that except for Data Clean and VHT, the values of w_2 and w_3 representing communication penalties are 0 for Model 1. Also, for Model 2, the values of w_2 corresponding to penalty for the inter-worker communication are small (up to 0.01).

In Figure 9, we plot actual and predicted MST values as functions of the number of VMs. Actual measurements of MST are shown in dots. Predicted values for Model 1 are shown in solid lines and predicted values for Model 2 are shown in dotted lines. From the figure, we can see that the actual MST values either hit a bottleneck (*i.e.*, Grep, Rolling Count, Data Clean, and VHT) or linearly improving (*i.e.*, Unique Visitor and Page View). Model 1 captures the bottlenecks for Grep and Rolling Count relatively well. Model 2 is trained almost as a linear function (*i.e.*, w_2 is at most 0.0016) for Unique Visitor and Page View, and therefore, it captures their linear behavior well. However, Model 2 fails to capture the non-linear behavior of Grep. Both models are equally well fitted to Data Clean and VHT.

Figure 10 shows the validation error in RMSE for model selection. To obtain these results, we used the following validation samples selected by Algorithm 1: 26 VMs for Grep, 25 and 72 VMs for Rolling Count, 25 VMs for Unique Visitor, 25 VMs for Page View. For Data Clean and VHT, since both models produced very close predicted MST values, there were no discrepancies larger than the $\tau = 0.10$ threshold. Thus, the models for these two benchmarks were compared using training errors. As the result of validation, Model 1 is chosen for Grep, Rolling Count, and Model 2 is chosen for the rest of benchmarks. Looking at Figure 9, these model selection results are visually convincing. Rolling Count has similar RMSE errors for Model 1 and Model 2. However, due to the second validation sample from 72 VMs, Model 2 is selected. For Data Clean and VHT, both models are visually very close and the training errors are almost identical.

Figure 11 shows prediction error in the Mean Absolute Percentage Error (MAPE) by the selected models. The MAPE is defined as:

$$MAPE = 100 \cdot \frac{1}{n} \sum_{i=1}^{n} \frac{|t_i - p_i|}{|t_i|}, \tag{10}$$

where n is the number of samples, t_i is the i-th sample's true value, and p_i is the i-th sample's predicted value. The figure plots MAPE error computed from the selected models and MAPE error computed from the mean value of actual MST samples. Since the mean value is known to minimize the sum of squared error ($\sum_{i=1}^{n}(t_i - p_i)^2$), even though it does not guarantee to minimize the MAPE, it shows low error in MAPE. Overall, the MAPE error for our prediction framework is up to 15.8%. Since there is some variance in the actual MST samples, even the prediction made from the mean has MAPE error up to 6.9%.

TABLE 2
Weights of Models 1 and 2 after training.

Benchmark	Model 1				Model 2		
	w_0	w_1	w_2	w_3	w_0	w_1	w_2
Grep	0.01617	1.04913	0	0	0.63117	0.74233	0.00063
Rolling Count	0.11748	2.75500	0	0	0.16593	0.27683	0.00322
Unique Visitor	0.02050	1.67805	0	0	0.01615	0.56212	0.00160
Page View	0.03512	1.61983	0	0	0.27180	0.49439	0.00089
Data Clean	0.11039	1.62948	0	0.00004	0.12286	0.49693	0.01233
VHT	0.01958	0	0.00005	0	50.69306	0	0.00394

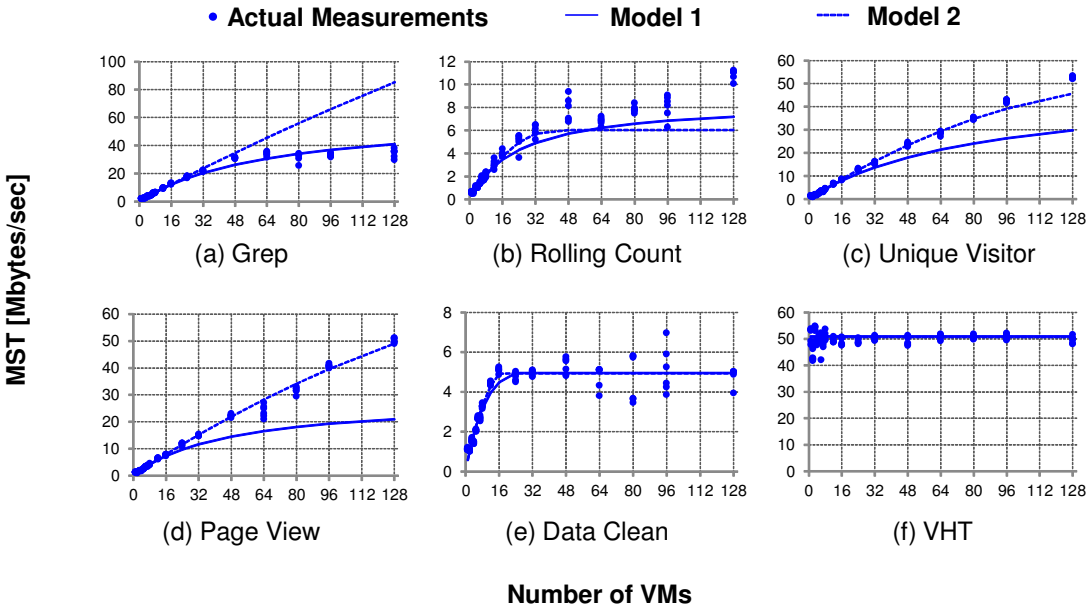


Fig. 9. MST prediction results for typical use-case benchmarks: (a) Grep, (b) Rolling Count, (c) Unique Visitor, (d) Page View, and (e) Data Clean, and a machine learning application: (f) VHT, using $\mathcal{S}_{\text{train}} = \{3, 4, 6, 8, 24\}$ for Models 1 and 2. X-axis: number of VMs. Y-axis: MST [Mbytes/sec].

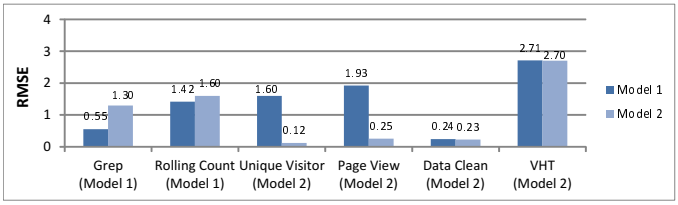


Fig. 10. Validation error (RMSE) and selected models for the typical use-case benchmarks. Models are trained with $\mathcal{S}_{\text{train}} = \{3, 4, 6, 8, 24\}$.

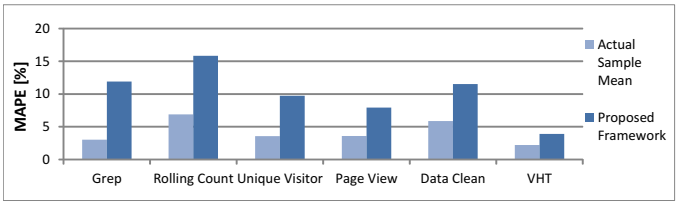


Fig. 11. Prediction error (MAPE) for the typical use-case benchmarks using mean value of actual MST samples and prediction made by the proposed framework.

6 Evaluation of Simulation-Based Elastic VM Scheduling

We evaluate the cost efficiency and SLA violations of the framework we proposed in Section 4 with simulation-based elastic VM scheduling. We run the simulation on a realistic workload while considering a constant overhead for VM allocation and application reconfiguration.

6.1 Experimental Settings

The workload we use is based on the FIFA World Cup 1998 website access logs over three weeks time (500 hours) [42], as shown in the light blue regions of Figure 12. We simulate elastic VM scheduling separately for the five typical use-case benchmarks from Intel Storm Benchmarks and VHT from SAMOA that we used in Section 5. The original workload is presented in the numbers of access to the website. Therefore, we need to convert it to input data rates to be able to apply our framework. We normalize the input data rates by 90% of the maximum MST samples observed for each benchmark. After this conversion, the peak input data rates are: 32, 10, 48, 46, 5, and 49 Mbytes/sec for Grep, Rolling Count, Unique Visitor, Page View, Data Clean, and VHT respectively. One can think as if we simulate web-based

data processing services, which need to serve fluctuating user requests over time.

We denote the sequence of hourly input data rates as $\lambda(t)$ for $t = 0, 1, ..., 499$ hours. Our SLA is to keep higher MST than input data rates (*i.e.,* $\lambda \leq MST$ as shown in Section 3.3). The scheduling interval of the simulation is one hour. For every t, our VM scheduler has a chance to allocate or deallocate one or more VMs. We schedule the minimum number of VMs $\hat{m}$ to satisfy the SLA as follows:

$$\hat{m} = \min\ MST(m) \quad \text{s.t.} \quad \lambda(t) \leq MST(m), \tag{11}$$

where $\hat{m}$ is given by the prediction model. In case even the maximum number of VMs (*i.e.*, 128 VMs) cannot satisfy the SLA, we schedule 128 VMs. After the scheduler makes a scheduling decision, we consider one minute delay including overhead expected for VM allocation and the stream processing system's reconfiguration such as rebalancing tasks. Thus, we need to keep using allocated VMs from the previous scheduling time slot during this delay period.

To evaluate our framework, we define the *ground truth model* MST_{tru} as follows: Given a VM count m, MST_{tru} returns the average of actual MST values measured for m. Using the ground truth model, SLA violations are counted if the following condition is met:

$$MST_{\text{tru}}(\hat{m}) < \lambda(t). \tag{12}$$

Violations are evaluated as the percentage of violation time to the total simulation time of 500 hours.

For each benchmark, we train Models 1 and 2 with MST values measured for $\mathcal{S}_{\text{train}} = \{3, 4, 6, 8, 24\}$. We use the m4.large instance type of Amazon EC2 which costs $0.10 per hour (as of August 2017). We compare the following scaling policies in terms of hourly cost and SLA violation rate:

- *Static (peak)*: Static VM allocation policy that covers the peak load.
- *Static (average)*: Static VM allocation policy that covers the average load.
- *Elastic (ground truth)*: Scaling policy based on the ground truth model. We plug in MST_{tru} for MST in Equation (11) and schedule the predicted minimum number of VMs $\hat{m}$. Also, no delay for VM allocation and application reconfiguration is considered. This corresponds to instantaneous reconfiguration, which is impossible in current cloud computing environments. This policy simulates an extremely ideal case and allocates minimum VMs with 0% violation rate.
- *Elastic (OP: x%)*: Scaling policy based on the selected MST prediction models in Section 5 with over-provisioning. When we apply x% of over-provisioning, we allocate $\lceil (1+\frac{x}{100})\hat{m} \rceil$ VMs. We test $x \in \{0, 5, 10, 20\}$ [%].
- *Elastic (OP: 1% violation)*: Scaling policy based on the selected MST prediction models in Section 5 with over-provisioning. We gradually increase the over-provisioning rate starting from 1% until the violation rate becomes less than 1%. We test this policy to evaluate the cost-efficiency of our prediction models when we get a comparable violation rate to the "Static (peak)" and "Elastic (ground truth)" policies.

Note that moderate over-provisioning is a common practice in actual VM provisioning to account for the inaccuracy of prediction models.

6.2 Cost Efficiency Results

Table 3 shows results of average hourly VM usage cost and SLA violations for the tested benchmarks. From the "Elastic (OP: 1% vio.)" policy, Grep, Rolling Count, Unique Visitor, and VHT achieve less than 1% violations with reasonably low over-provisioning rates: 1, 1, 4, and 0% respectively. Grep and Unique Visitor required 17% and 8% more cost per hour compared to the "Elastic (ground truth)" policy to achieve less than 1% violations, whereas Rolling Count required 57% more cost per hour. This is due to the conservative prediction of Model 1, as shown in Figure 9(b): the predicted values are consistently lower than the actual MST values, and thus, Model 1 predicts more VMs than necessary. For Grep, violation rates are increased from 0.077% to 0.637% when we increase the over-provisioning rate from 5% to 10%. This is due to the fact that the actual MST values are lower for $m = 80$ and $m = 96$ than for $m = 64$ as shown in Figure 9(a); however, the violation rates are quite low (less than 1%) for both 5% and 10% over-provisioning. VHT does not require over-provisioning at all due to its non-scalable behavior. In fact, static scheduling with one VM ends up with no violations.

Unlike the rest of the test applications, Page View and Data Clean required large over-provisioning rates, 19% and 34%, to achieve less than 1% violations with the "Elastic (OP: 1% vio.)" policy. For Page View, this is due to under-provisioning caused by Model 2 for 64 to 80 VMs as shown in Figure 9(d). In contrast to what happened with Rolling Count, for these VM counts, the model predicts higher MST values than the ground truth MST. Therefore, to compensate for under-provisioning, Model 2 requires the large over-provisioning rate of 19% to avoid violations. For the same reason, Model 2 causes under-provisioning for Data Clean. Since most of violations occurred in a lower MST range (*i.e.*, around 1.5 to 3 Mbytes/sec), it is not visibly apparent from Figure 9(e).

Overall, despite the different values of over-provisioning rates, compared to the "Static (peak)" policy, the "Elastic (OP: 1% vio.)" policy saves at least 36% cost to achieve the same level of violations. Also, using a 20% of over-provisioning rate with "Elastic (OP: 20%)" policy, we achieved less than 0.1% violation rates except for Data Clean, which violation rate remained 4.013%. However, in practice, the demand usually oscillates, as we can see in Figure 12. As long as a model both under-provisions and over-provisions for different phases of the oscillating demand, the stream processing system can eventually processes data in Kafka that are accumulated during the under-provisioned phases.

Figure 12 presents the sequences of input data rates and allocated VMs/MST values scheduled by different policies in Table 3. For visual clarity, we only plot "Elastic (OP: 0%)" and "Elastic (OP: 1% vio.)" for our models. We plot these sequences for (a) Grep, (b) Rolling Count, and (c) Page View benchmarks. For Grep in Figure 12(a) and Page View in Figure 12(c), we can see that the allocated MST and VMs for our elastic scheduling policies closely follow the

TABLE 3
Average hourly VM usage cost and SLA violations for the typical use-case benchmarks (Grep, Rolling Count, Unique Visitor, Page View, and Data Clean) from Intel Storm Benchmarks and a machine learning application (VHT) from SAMOA.

Policy	Grep (Model 1)		Rolling Count (Model 1)		Unique Visitor (Model 2)		Page View (Model 2)		Data Clean (Model 2)		VHT (Model 2)	
	Cost/hr	Vio.[%]	Cost/hr	Vio.[%]	Cost/hr	Vio.[%]	Cost/hr	Vio.[%]	Cost/hr	Vio.[%]	Cost/hr	Vio.[%]
Static (peak)	5.300	0.000	11.900	0.000	11.400	0.000	11.500	0.000	1.600	0.000	0.100	0.000
Static (average)	2.400	44.600	2.800	44.600	5.600	44.800	6.600	45.600	0.800	28.400	0.100	0.000
Elastic (ground truth)	2.482	0.000	3.651	0.000	5.678	0.000	5.991	0.000	0.714	0.000	0.100	0.000
Elastic (OP: 0%)	2.808	6.140	5.643	1.460	5.876	34.637	5.716	49.517	0.712	28.480	0.100	0.000
Elastic (OP: 5%)	2.998	0.077	5.815	0.010	6.193	0.290	6.051	35.493	0.813	4.050	0.200	0.000
Elastic (OP: 10%)	3.137	0.637	5.950	0.003	6.451	0.107	6.333	25.453	0.831	4.050	0.200	0.000
Elastic (OP: 20%)	3.410	0.007	6.200	0.003	6.932	0.030	6.872	0.097	0.896	4.013	0.200	0.000
Elastic (OP: 1% vio.)	2.908	0.087	5.723	0.010	6.142	0.330	6.825	0.100	1.019	1.000	0.100	0.000
OP rate [%]	1		1		4		19		34		0	

"Elastic (ground truth)" policy; however, for Rolling Count in Figure 12(b), there is wasted VMs/MST capacity between our elastic scheduling policies and the "Elastic (ground truth)" policy. This is due to over-provisioning caused by Model 1 as shown in Figure 9(b). When the ground truth policy allocated only 43 VMs to satisfy 7.22 Mbytes/sec demand, "Elastic (OP: 0%)" policy allocated 128 VMs. Since the maximum number of VMs allowed to schedule is 128, the sequence looks flattens out at 128 VMs around the demand peaks. Looking at Figure 12(c) for Page View, "Elastic (OP: 0%)" frequently schedules lower VMs/MST values than the input data rates due to under-provisioning; however, using 19% of over-provisioning rate, the "Elastic (OP: 1% vio.)" policy effectively pushes MST values above the input data rates.

In summary, we need to give a relatively large over-provisioning rate to compensate for the large variance for Data Clean; however, other than Data Clean, our framework is able to achieve less than 0.1% SLA violations with 20% over-provisioning. Moreover, we save 36% cost compared to a static VM scheduling that covers the peak workload to achieve the same level of SLA violations. These simulation results show that the prediction models trained by our framework are useful for cost-efficient elastic stream processing.

7 Discussion

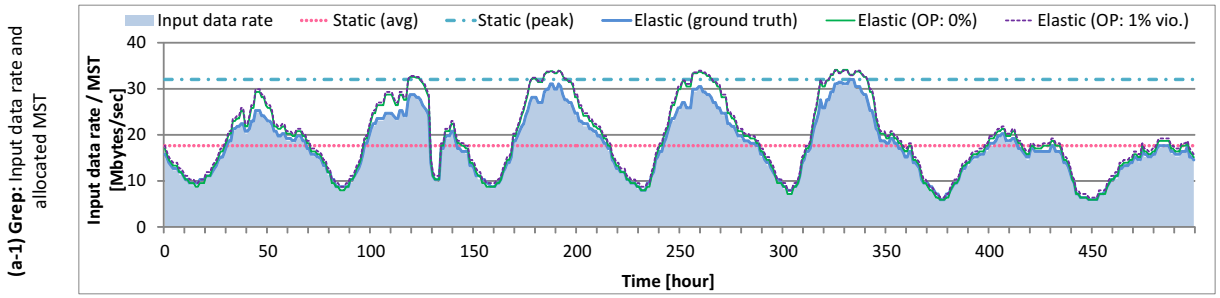
In this section, we discuss the results from the experiments we have done in Sections 5 and 6.

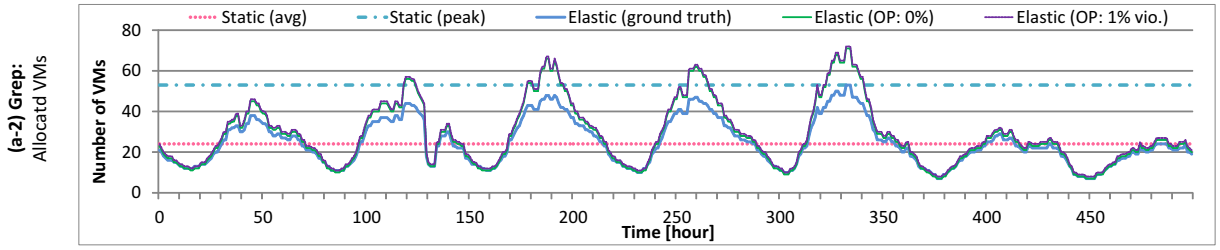
Qualitative characteristics of prediction in elastic scheduling: In Section 5, we looked at prediction performance of our models through RMSE and MAPE, which evaluate absolute differences between actual and predicted MST values. However, when we use a trained model for elastic VM scheduling in Section 6, the relationship between the model and actual measurements becomes more important than the absolute differences. For example, Model 1 for Unique Visitor (Figure 9(c)) and Page View (Figure 9(d)) show consistently lower prediction values than the actual MST values. As a result, the framework allocates more VMs than needed. On the other hand, Model 2 for Grep (Figure 9(a)) predicts consistently higher MST values than are actually obtained. Thus, the framework under-provisions VMs. In a realistic use-case, service providers may want to over-provision VMs to provide consistent user experience rather than under-provision VMs to save some cost. In Section 6, we used over-provisioning rates to compensate for models that predict too few VMs. An alternative solution could be to use a metric that favors over-provisioning models over under-provisioning ones when we select VM subsets in Phase 1 and also select models in Phase 2.

Likely cause of the bottlenecks: From the experiments in Section 5, the scalability for Grep, Rolling Count, Data Clean, and VHT is limited. The reason seems to be load imbalance between workers. For the Grep benchmark, to compute the total count of matched patterns, the global counter is incremented by a single thread, and MST is bounded by the performance of that single thread. For Rolling Count, the bottleneck may be caused by imbalanced distributions of words. Just as the map-reduce programming framework [30], once the first layer of processing units splits texts into words, the next processing units are determined by the hash value of a word. Thus, depending on the distribution of words, some nodes are more loaded than other nodes. Similar to Grep, maximum performance is bounded by the nodes that are assigned frequently appearing words. The Data Clean's performance limitation seems to be caused by a similar reason to Grep: URLs filtered by the 200 status code go to the same node. Since the objective of our prediction framework is to accurately predict MST for larger numbers of VMs, we are not concerned about application-specific bottlenecks in this paper; however, clarifying the mechanism behind these bottlenecks could help improve the accuracy of prediction models in future.

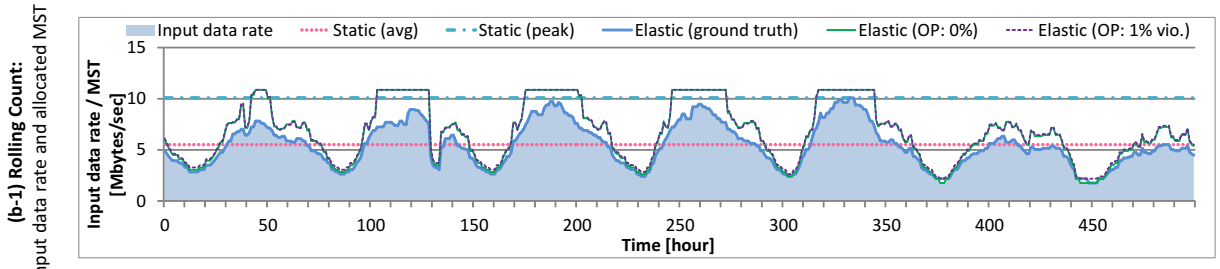
Time complexity and generalization of the proposed framework: Due to the exhaustive search, the time complexity of the subset search method presented in Section 4.4.1 is $O(|\mathcal{V}_{\text{train}}|^c \cdot |\mathcal{A}|)$. With an implementation using octave, the search with $M_{\text{train}} = 32$ took only a few minutes to find the best VM subset S_{train} on a laptop PC with Intel Core i5 CPU. As we increase the size of $\mathcal{V}_{\text{train}}$ through M_{train} and/or c, the runtime is expected to grow exponentially. One way to keep the runtime manageable is to use random sampling to identify S_{train}, rather than performing an exhaustive search over all subsets. We should be able to find the right balance between the runtime and the likelihood of finding the optimal subset.

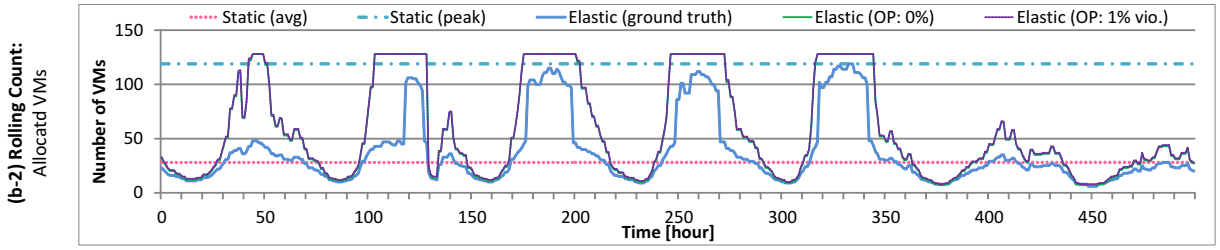
Currently when we determine data points for validation in Algorithm 1, we assume the framework only uses the two models we presented in Section 4.3. We can generalize



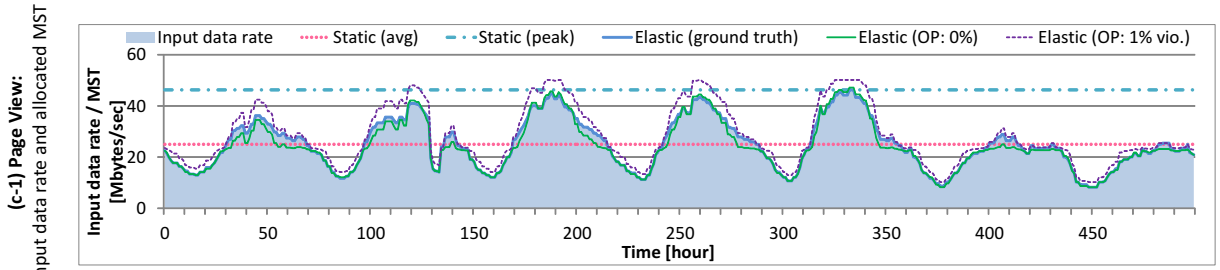


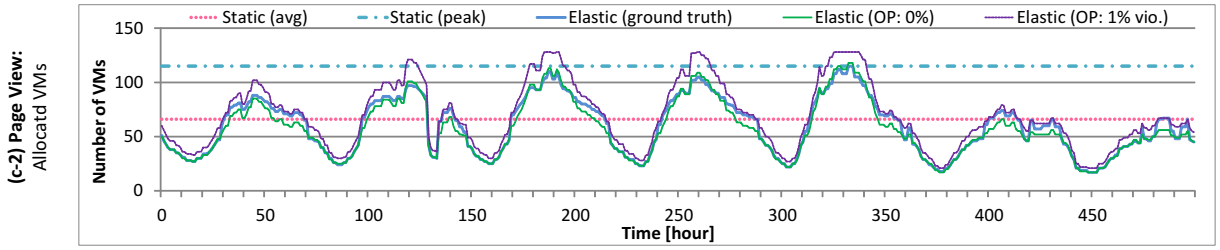
(a) Grep





(b) Rolling Count





(c) Page View

Fig. 12. Input data rate sequence created from the FIFA World Cup 1998 website access logs and MST generated by elastic VM scheduling based on prediction models for (a) Grep, (b) Rolling Count, and (c) Page View benchmarks.

this algorithm to support N models by defining the average pairwise discrepancies between N models as follows:

$$d(m) = \frac{1}{\binom{n}{2}} \cdot \sum_{i,j} \frac{|MST^{(i)}(m) - MST^{(j)}(m)|}{\min(MST^{(i)}(m), MST^{(j)}(m))}. \quad (13)$$

With this change, the entire framework will be compatible to N models.

SLA violations in practical settings: In Section 6, we strictly counted the time when allocated VMs failed to satisfy Inequality (12); in practice, we can catch up accumulated events in Kafka using extra processing power afforded by over-provisioned VMs. Therefore, even if the input data rate is greater than the stream processing system's MST, as long as this situation is temporary, the stream processing system can keep operating normally with a temporary increase in latency.

8 Conclusion

We have presented a framework to predict the maximum sustainable throughput (MST) for cloud-based stream processing applications. We identified a common data processing environment used by modern stream processing systems and presented two models for MST prediction. We statistically determine the best subset of VM counts in terms of prediction error to collect training samples. For each new application, we train the framework models using this subset. The framework takes several trained models and selects the model that is expected to predict MST values for the target application with the lowest error. We evaluated our framework on streaming applications in Apache Storm, using up to 128 VMs. Experiments showed that our framework can predict MST with up to 15.8% average prediction error. Further, we evaluated our prediction models with simulation-based elastic VM scheduling for a realistic data streaming workload. Simulation results showed that with 20% over-provisioning, our framework was able to achieve less than 0.1% SLA violations for the majority of the applications we tested. We also saved 36% cost compared to a static VM scheduling that covers the peak workload to achieve the same level of SLA violations.

In future work, we plan to apply the proposed prediction framework to other stream data processing engines such as Flink to confirm the applicability of our approach. Other interesting future directions include online learning to improve the performance prediction model accuracy over time, the use of meta-algorithms such as ensemble learning to construct a prediction model from multiple weak models, and even larger-scale performance simulation using a cloud environment simulator such as CloudSim [43].

Acknowledgments

This research is partially supported by the DDDAS program of the Air Force Office of Scientific Research, Grant No. FA9550-15-1-0214 and NSF Awards, Grant No. 1462342, 1553340, and 1527287. The authors would like to thank an Amazon Web Services educational research grant and a Google Cloud Credits Award.

References

[1] S. Imai, R. Klockowski, and C. A. Varela, "Self-healing spatio-temporal data streams using error signatures," in *IEEE 2nd International Conference on Big Data Science and Engineering*, 2013, pp. 957–964.

[2] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, and J. B. Ingram, "Spark-based anomaly detection over multi-source VMware performance data in real-time," in *IEEE Symposium on Computational Intelligence in Cyber Security*, 2014, pp. 1–8.

[3] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. N. Koutsopoulos, M. Rahmani, and B. Gü, "Real-time traffic information management using stream computing," *IEEE Data Engineering Bulletin*, vol. 33, pp. 64–68, June 2010.

[4] S. Imai, S. Patterson, and C. A. Varela, "Elastic virtual machine scheduling for continuous air traffic optimization," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2016, pp. 183–186.

[5] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic stream processing for the internet of things," in *9th IEEE International Conference on Cloud Computing*, 2016, pp. 100–107.

[6] C. Hochreiner, S. Schulte, S. Dustdar, and F. Lecue, "Elastic stream processing for distributed environments," *IEEE Internet Computing*, vol. 19, no. 6, pp. 54–59, Nov 2015.

[7] A. Shukla and Y. Simmhan, "Benchmarking distributed stream processing platforms for IoT applications," *arXiv preprint arXiv:1606.07621*, 2016.

[8] Apache Software Foundation, "Apache Storm," http://storm.apache.org/, Accessed: 2017-08-02.

[9] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *SIGMOD International Conference on Management of Data*, 2014, pp. 147–156.

[10] Apache Software Foundation, "Apache Flink," http://spark.apache.org/, Accessed: 2017-08-02.

[11] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin, in the special issue on Next-gen Stream Processing*, vol. 38, no. 4, Dec 2015.

[12] Apache Software Foundation, "Apache Samza," http://samza.apache.org/, Accessed: 2017-08-02.

[13] ——, "Apache Spark," http://spark.apache.org/, Accessed: 2017-08-02.

[14] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *24th ACM Symposium on Operating Systems Principles*, 2013, pp. 423–438.

[15] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, Dec 2014.

[16] T. Li, J. Tang, and J. Xu, "A predictive scheduling framework for fast and distributed stream data processing," in *IEEE International Conference on Big Data*, 2015, pp. 333–338.

[17] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, 2014, pp. 13–22.

[18] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *35th IEEE International Conference on Distributed Computing Systems*, 2015, pp. 399–410.

[19] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: Dynamic resource scheduling for real-time analytics over fast streams," in *35th IEEE International Conference on Distributed Computing Systems*, 2015, pp. 411–420.

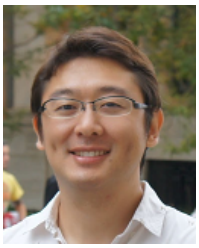
[20] A. Ishii and T. Suzumura, "Elastic stream computing with clouds," in *4th IEEE International Conference on Cloud Computing*, 2011, pp. 195–202.

[21] P. Lama and X. Zhou, "AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud," *9th ACM International Conference on Autonomic Computing*, p. 63, 2012.

[22] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th Usenix Conference on Networked Systems Design and Implementation*, 2016, pp. 363–378.

[23] G. Mariani, A. Anghel, R. Jongerius, and G. Dittmann, "Predicting cloud performance for hpc applications: a user-oriented

approach," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2017, pp. 524–533.
[24] Intel Corporation, "Storm benchmark," https://github.com/intel-hadoop/storm-benchmark, Accessed: 2017-02-15.
[25] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *16th ACM Annual Middleware Conference*, 2015, pp. 149–161.
[26] L. Fischer and A. Bernstein, "Workload scheduling in distributed stream processors using graph partitioning," in *IEEE International Conference on Big Data*, 2015, pp. 124–133.
[27] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 2002, pp. 1–16.
[28] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *ACM SIGMOD International Conference on Management of Data*, 2015, pp. 239–250.
[29] A. Verma, L. Cherkasova, and R. Campbell, "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," *8th ACM International Conference on Autonomic Computing*, pp. 235–244, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1998637
[30] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," *6th Symposium on Operating Systems Design and Implementation*, pp. 137–149, 2004.
[31] Microsoft, "Measuring maximum sustainable engine throughput," https://msdn.microsoft.com/en-us/library/cc296884(v=bts.10).aspx, Accessed: 2017-02-15.
[32] C. Davatz, C. Inzinger, J. Scheuner, and P. Leitner, "An approach and case study of cloud instance type selection for multi-tier web applications," in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2017, pp. 534–543.
[33] S. Imai, S. Patterson, and C. A. Varela, " Maximum Sustainable Throughput Prediction for Data Stream Processing over Public Clouds," in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2017.
[34] G. D. F. Morales and A. Bifet, "SAMOA: Scalable Advanced Massive Online Analysis," *Journal of Machine Learning Research*, vol. 16, pp. 149–153, Jan 2015.
[35] J. Kreps and L. Corp, "Kafka : a distributed messaging system for log processing," *ACM SIGMOD Workshop on Networking Meets Databases*, p. 6, 2011.
[36] Yahoo! Inc., "Yahoo streaming benchmarks," https://github.com/yahoo/streaming-benchmarks, Accessed: 2017-08-02.
[37] O. Etzion and P. Niblett, *Event processing in action*. Manning Publications Co., 2010.
[38] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from data*. New York, NY, USA: AMLBook, 2012.
[39] Luis Martin Garcia, "TCPDUMP/LIBPCAP public repository," http://www.tcpdump.org/, Accessed: 2017-08-02.
[40] N. Kourtellis, G. D. F. Morales, A. Bifet, and A. Murdopo, "Vht: Vertical hoeffding tree," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 915–922.
[41] Mark Twain, "The project gutenberg ebook of the adventures of tom sawyer," https://www.gutenberg.org/files/74/74-h/74-h.htm, Accessed: 2017-08-02.
[42] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE Network*, vol. 14, no. 3, pp. 30–37, May 2000.
[43] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, Jan 2011.



Shigeru Imai is a PhD candidate in the Department of Computer Science at Rensselaer Polytechnic Institute. He received his B.E. and M.E. from Tokyo Institute of Technology, Japan. His research interests include performance prediction and elastic resource scheduling for cloud-based systems.

Stacy Patterson is the Clare Boothe Luce Assistant Professor in the Department of Computer Science at Rensselaer Polytechnic Institute. She received the MS and PhD in computer science from the University of California, Santa Barbara in 2003 and 2009, respectively. From 2009-2011, she was a postdoctoral scholar at the Center for Control, Dynamical Systems and Computation at the University of California, Santa Barbara. From 2011-2013, she was a postdoctoral fellow in the Department of Electrical Engineering at Technion - Israel Institute of Technology. Dr. Patterson is the recipient of a Viterbi postdoctoral fellowship, the IEEE CSS Axelby Outstanding Paper Award, and an NSF CAREER award. Her research interests include distributed systems, cloud computing, sensor networks, and the Internet of Things.

Carlos A. Varela is an Associate Professor in the Department of Computer Science at Rensselaer Polytechnic Institute. Dr. Carlos A. Varela received his B.S. with honors, M.S., and Ph.D. in Computer Science at the University of Illinois at Urbana-Champaign. Dr. Varela is Associate Editor and Information Director of the ACM Computing Surveys journal, and has served as Guest Editor of the Scientific Programming journal. Dr. Varela is a recipient of several research grants including the NSF CAREER award, two IBM SUR awards, and two IBM Innovation awards. His current research interests include data streaming, cloud computing, middleware for adaptive distributed systems, concurrent programming models and languages, and software verification.