Maximum Sustainable Throughput Prediction for Data Stream Processing over Public Clouds

Shigeru Imai, Stacy Patterson, and Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute
{*imais,sep,cvarela*}*@cs.rpi.edu*

***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 dynamically changing the number of VMs used by the system. In this paper, we identify a common data processing environment used by modern data stream processing systems, and we propose MST prediction models for this environment. We train the models using linear regression with samples obtained from a few VMs and predict MST for a larger number of VMs. To minimize the time and cost for model training, we statistically determine a set of training samples using Intel's Storm benchmarks with representative resource usage patterns. Using typical use-case benchmarks on Amazon's EC2 public cloud, our experiments show that, training with up to 8 VMs, we can predict MST for streaming applications with less than 4% average prediction error for 12 VMs, 9% for 16 VMs, and 32% for 24 VMs. Further, we evaluate our prediction models with simulation-based elastic VM scheduling on a realistic workload. These simulation results show that with 10% over-provisioning, our proposed models' cost efficiency is on par with the cost of an optimal scaling policy without incurring any service level agreement violations.**

***Keywords*-cloud computing; performance prediction; resource management; auto-scaling**

I. 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 VM configurations 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. It is an application performance metric that is useful from the 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. Recent elastic stream processing studies primarily focus on guaranteeing latency [16], [17], [18], [19]. ElasticStream [20] estimates maximum throughput using a model that is linear in the number of VMs, which is not realistic for all applications, as we show in this work.

We propose models to predict MST for both linearly and non-linearly scalable applications. We identify a common data processing environment used among modern stream processing systems and propose a framework to measure MST for streaming applications. The models assume a homogeneous VM instance type and predict MST only by the number of VMs. We train and test the models using linear regression with performance results obtained from Intel's Storm benchmarks [21] running on Amazon EC2 cloud. To minimize the cost for training, we statistically determine a good set of training samples from a few VMs. We then use this model to predict MST up to a larger number of VMs (32 VMs). Further, we evaluate models' cost-effectiveness for elastic stream processing by simulation. In summary, our contributions are as follows:

1) A maximum sustainable throughput measurement framework for a common data stream environment.
2) Maximum sustainable throughput prediction models for stream processing systems.
3) Evaluations, using Storm benchmarks from Intel running on Amazon EC2, that show that the proposed models are reasonably accurate and cost effective.

The rest of the paper is organized as follows. In Section II, we present related work for performance models used in stream processing. In Section III, we describe the background on stream processing systems, including a common data processing environment and the concept of maximum sustainable throughput. Section IV introduces a framework for maximum sustainable throughput measurement. Section V presents the proposed maximum sustainable throughput prediction models and how we statistically select training samples. Section VI shows the evaluation of our models' prediction accuracy, and then Section VII shows the cost-effectiveness of a proposed model for elastic stream processing by simulation. Section VIII discusses the results of experiments. Finally, we conclude the paper in Section IX.

II. Related Work

In stream processing, one of the most common performance metrics is latency. There are several proposals on topology-aware latency models [16], [17], [18], [19]. They all assume that the user gives a logical topology composed of multiple processing units, which are parallelized as threads (also often called tasks) at runtime for distributed execution. This model is commonly used in modern stream processing systems such as Apache Storm [8] and Apache Flink [10]. 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. R-Storm implements a resource-aware task scheduler on top of Storm [22]. It tries to increase throughput by maximizing resource utilization while 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 [23].

To alleviate excessive incoming workload for stream processing, techniques such as random sampling [24] and backpressure [25] 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.

To safely scale up a cluster to process fluctuating workload without relying on these techniques, it is important to know the maximum processing capacity of the cluster. Maximum sustainable throughput (MST) is a general metric for services-based applications [26], but has received less attention compared to latency in stream 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 VI. Unlike ElasticStream, we model MST for both linearly and non-linearly scalable applications.

III. Background on stream processing systems

In this section, we describe the background on stream processing systems. We first show a data processing environment commonly used by multiple data processing frameworks (Section III-A), and then, we introduce the concept of maximum sustainable throughput (MST) for this environment (Section III-B). Also, we show how we can use MST in SLAs for stream processing systems (Section III-C).

A. 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 producer to the data store. The producer sends events at the rate of λ and they are appended to message queues in Kafka. Kafka is a message queuing system that is scalable, high-throughput, and fault-tolerant [27]. The consumer (*i.e.*, a stream processing system) pulls data out of Kafka as quickly as it can at the throughput of τ. Note that there can be multiple producers and consumers working simultaneously with one Kafka instance, which can be multiple nodes. After the consumer processes events, it optionally stores results in the data store (*e.g.*, a file system or database).

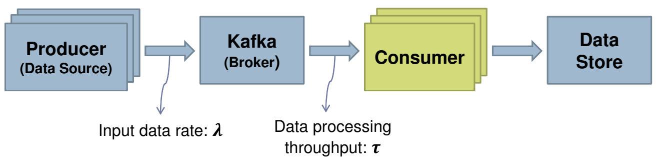


Figure 1. Common stream processing environment.

B. *Maximum sustainable throughput*

Figure 2 shows examples of stream processing by Storm ingesting events from Kafka with increasing data input rates. Using the stream processing environment in Figure 1, these experiments are performed with a network intensive application as shown in Figure 2(a) and a memory intensive application as shown in Figure 2(b). For both examples, we gradually increase the input data rate by adding data producers, where each producer generates 1.0 Mbytes/sec. In Figure 2(a), the data processing throughput ingested by Storm starts diverging from the input data rate at around 140 seconds. This means that Storm cannot keep up with the input data rate, and the incoming events start accumulating in Kafka. This data processing is not sustainable as the accumulated events will eventually exceed the capacity of available storage in Kafka. Thus, the MST for this example is about 2.5 Mbytes/sec. In Figure 2(b), the data processing throughput rate matches the input data rate until around 250 seconds but starts oscillating after that. From the log, we found that an out-of-memory error occurred in one of the Storm worker nodes when the output rate started dropping around 250 seconds, and then Nimbus (the master node of Storm) relaunched a worker. After the relaunched worker becomes active at around 400 seconds, the workers rapidly digest accumulated events in Kafka. Just as the first example, when the data processing throughput starts diverging from the input data rate at around 250 seconds, events start accumulating in Kafka. For the same reason as the first example, the MST is around 1.5 Mbytes/sec for the second example.

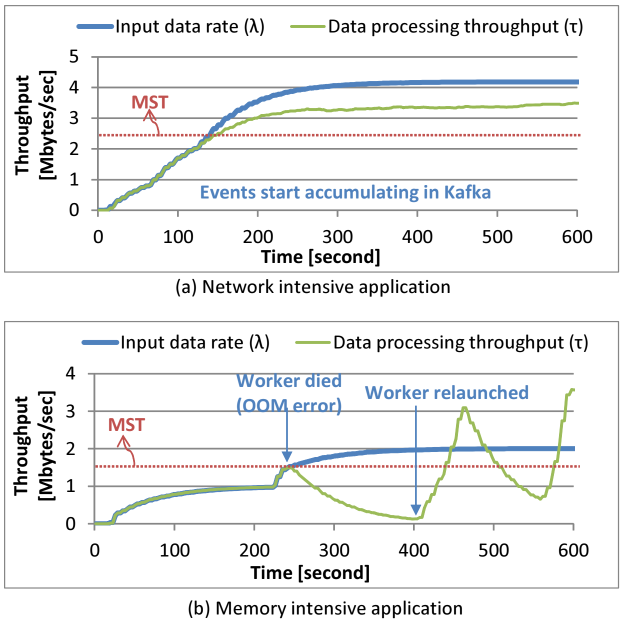


Figure 2. Input data rates and data processing throughput by Apache Storm.

As we can see from these examples, MST tells us the limit for sustainable data processing. To scale up a cluster to process fluctuating workload, we need to predictively scale up the cluster by adding VMs before it hits the MST. Therefore, we need a prediction model to estimate how many VMs we need to satisfy the required MST.

C. *Performance objectives and Service Level Agreements*

In the common stream processing environment we have shown in Figure 1, latency is the time it takes to process an event, transmitted from Kafka, by the consumer. Also, end-to-end latency is the time taken for an event produced at the producer to get to the data store. As we have shown in Section III-A, data processing throughput τ is the rate of events ingested by the consumer. According to Etzion and Niblett, the following performance objectives are commonly used in event processing systems [28]: 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 #5 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. That is, they restrict the latency l to be smaller than or equal to some constant latency $l_{\max}$:

$$l \leq l_{\max}. \tag{1}$$

In this paper, the SLA of interest is related to #1 maximize input throughput, but not identical. 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:

$$\lambda \leq \tau_{\text{ms}}, \tag{2}$$

where λ is the input data rate and τ_{ms} 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 [28].

IV. Maximum sustainable throughput measurement framework

A. *Stream processing system model*

The common stream processing environment we have shown in Figure 1 has a high-level stream processing system model as shown in Figure 3. While we fix the master node of the cluster, we scale up the worker nodes by simply adding new VM 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 can be accessed from the consumer 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.

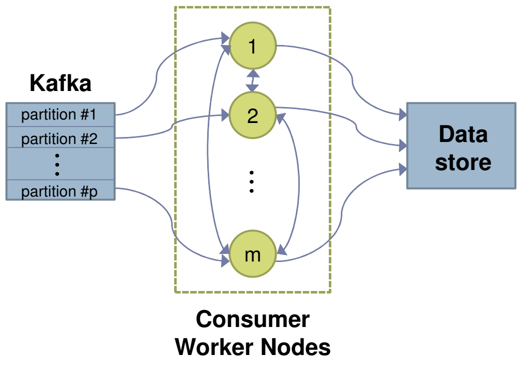


Figure 3. High-level stream processing system model.

B. Scaling policy

We define system-independent scaling rules that are applicable to multiple stream processing systems. 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 4 as an example). Each processing unit receives events from the preceding units and emits processing results to the following units. Processing units can be duplicated to an arbitrary number of threads (*i.e.*, tasks) when they are deployed on the cluster. 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.

$$d = \max(1, \lfloor (m \cdot \gamma)/n \rfloor). \tag{3}$$

This rule is a generalization of technique used in Yahoo Streaming Benchmarks [29]. When $n = 3, \gamma = 2$, examples of scaling to $m = 3$ and $m = 5$ are shown in Figure 4.

C. Obtaining maximum sustainable throughput

To measure MST, we gradually increase the input data rate until we observe one of the two cases we have shown in Section III-B. We collect both input data rate and data processing throughput from Kafka through the Java Management Extensions (JMX) interface. Since we do not need to access a stream data processing framework to get these metrics, this metric collection scheme is applicable to multiple stream data processing frameworks that support Kafka. We use the following `Mbeans` to get

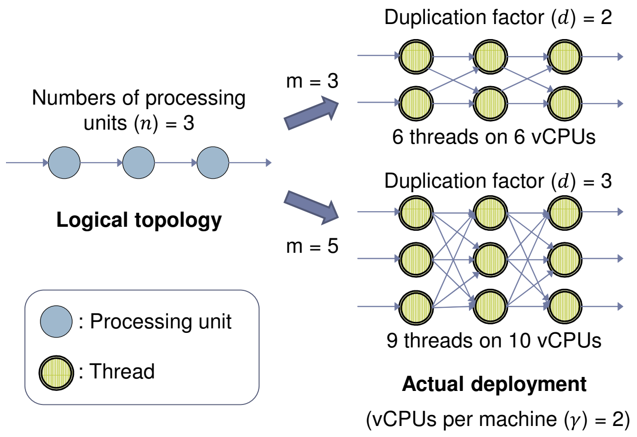

Numbers of processing units (n) = 3
Logical topology
m = 3
m = 5
Duplication factor (d) = 2
6 threads on 6 vCPUs
Duplication factor (d) = 3
9 threads on 10 vCPUs
: Processing unit
: Thread
Actual deployment
(vCPUs per machine (γ) = 2)

Figure 4. 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 ($\gamma = 2$).

the input data rate and processing throughput respectively.

```
kafka.server:name=BytesInPerSec,type=BrokerTopicMetrics
kafka.server:name=BytesOutPerSec,type=BrokerTopicMetrics
```

V. Modeling stream processing performance

We propose MST prediction models in Section V-A. To reduce the training cost and time, we use as few training samples as possible as we describe in Section V-B. We use Storm version 1.0.2 [8] throughout this paper for evaluation.

A. Performance model

We design two performance models for MST based on the system model shown in Figure 3, where there are m worker nodes (VMs) that communicate with each other. They take input data events from Kafka and optionally transmit processed results to the data store.

Factors: We assume that a combination of the following factors determine the total performance of the system shown in Figure 3.

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.

Model 1: As shown in Equation (4), Model 1 estimates MST ($\hat{\tau}_{\text{ms}}$) as a function of the number of VMs m. It is defined as the inverse of event processing time (t), 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.*, $w_i \geq 0, i = 0, ..., 3$).

$$\hat{\tau}_{\text{ms}}(m) = \frac{1}{t(m)} = \frac{1}{w_0 + w_1 \cdot \frac{1}{m} + w_2 \cdot m + w_3 \cdot m^2}. \quad (4)$$

This model is inspired by Ernest [30], which models job completion time for 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: As shown in Equation (5), Model 2 models MST ($\hat{\tau}_{\text{ms}}$) as a polynomial of the number of VMs m. The terms have the following meanings: base throughput (w_0), parallel processing gain (w_1), inter-worker communication overhead (w_2). Same as Model 1, 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.

$$\hat{\tau}_{\text{ms}}(m) = w_0 + w_1 \cdot m - w_2 \cdot m^2. \quad (5)$$

B. Training sample selection

We train Model 1 and Model 2 using linear regression with training samples of MST obtained from a few VMs. We statistically determine the most effective training samples in terms of prediction error and cost for Model 1 and Model 2, respectively, by exhaustive search. To find such training samples, we use the following *simple resource benchmarks* from Intel Storm Benchmark [21].

- *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. The sample selection steps are as follows.

Step 1. Collect MST samples: We run the three simple resource benchmarks, Word Count, SOL, and Rolling Sort, on Amazon EC2 using the `m4.large` instance type for two sets of VM configurations: $M_{\text{train}} = \{1, 2, 3, ..., 8\}$ and $M_{\text{test}} = \{12, 16, 24, 32\}$. After collecting MST for these VM configurations, we divide collected samples into two sets: D_{train} that contains training samples from M_{train} and D_{test} that contains test samples from M_{test}. That is, $D_{\text{train}} = \{(\tau_m, m) \mid m \in M_{\text{train}}\}$ and $D_{\text{test}} = \{(\tau_m, m) \mid m \in M_{\text{test}}\}$, where τ_m is the MST obtained from m VMs.

Step 2. Enumerate training subsets: Since we do not know which samples of D_{train} best predict MST in D_{test}, we create all possible training subsets with cardinality greater than one. That is, given $k \in \{2, 3, ..., 8\}$, we enumerate all the possible k combinations out of 8 training samples in D_{train} and create $\binom{8}{k}$ training subsets called $D^{k,j}_{\text{train}}$ as follows:

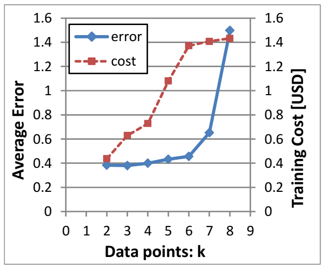
$$D^{k,j}_{\text{train}} \subseteq D_{\text{train}}, \quad j = 1, 2, ..., \binom{8}{k}. \quad (6)$$

Step 3. Identify best samples: For each benchmark, we train Model 1 and Model 2 using the training subsets $D^{k,j}_{\text{train}}$ and test the trained models with D_{test}. We then evaluate the total cost and the average test error over the three benchmarks for each k. Note that the cost is computed as $(\textit{total test hour}) * \$0.12/hr$ (cost per hour for `m4.large` as of November 2016). Since we use the homogeneous VM type `m4.large` only, the cost is linearly proportional to the time needed to obtain MST. When training Model 1, we change Equation (4) to a linear form in terms of weights to be able to apply linear regression as follows:

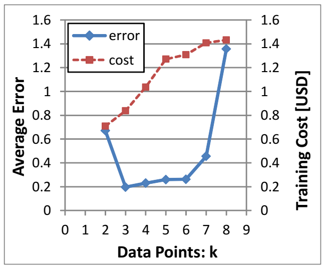
$$\frac{1}{\hat{\tau}_{\text{ms}}(m)} = t(m) = w_0 + w_1 \cdot \frac{1}{m} + w_2 \cdot m + w_3 \cdot m^2. \quad (7)$$

For Model 2, we directly apply linear regression to Equation (5).

The results are shown in Figure 5. For each data point k, the average prediction error and cost obtained from the best training subset that gives the lowest average error are plotted. Both models take the lowest average error when $k = 3$, and set of data points are $M_1 = \{3, 5, 8\}$ for Model 1 and $M_2 = \{5, 6, 8\}$ for Model 2. Since the cost of model training is reasonably low, we use M_1 and M_2 as the training samples for the model accuracy evaluation in Section VI.



(a) Model training result for Model 1



(b) Model training result for Model 2

Figure 5. Model training results for Model 1 and Model 2.

VI. Model accuracy evaluation

Using the best sets of training samples, M_1 for Model and M_2 for Model 2, we evaluate our prediction models using the the following four *typical use-case benchmarks* from Intel Storm Benchmark.

- *Grep*: Text inputs are pattern-matched with a regular expression and the number of matched sentences is counted.
- *Rolling Count*: Similar to word count, the number of word counts is emitted every x seconds.
- *Unique Visitor*: From web access logs, the number of unique visitor to website is counted.
- *Page View*: From web access logs, the number of page views per website is counted.

A. Experimental settings

As in Section V-B, we run the four typical use-case benchmarks, Grep, Rolling Count, Unique Visitor, Page View, on Amazon EC2 using the `m4.large` instance type for a set of VMs $M = M_1 \cup M_2 \cup M_{\text{test}} = \{3, 5, 6, 8, 12, 16, 24, 32\}$, and collect MST for these VMs. For Model 1, we use M_1 to learn parameters and predict MST for $M_1 \cup M_{\text{test}} = \{3, 5, 8, 12, 16, 24, 32\}$. Similarly, for Model 2, we use M_2 to learn parameters and predict MST for $M_2 \cup M_{\text{test}} = \{5, 6, 8, 12, 16, 24, 32\}$. For Grep and Rolling Count, we use the texts from The Adventures of Tom Sawyer [31] as inputs. For Unique Visitor and Page View, we artificially create web access logs that contain access to 100 randomly generated websites. We use default parameter settings for Storm version 1.0.2 [8].

B. Model accuracy results

Learned parameters for Model 1 and Model 2 are shown in Table I and II respectively. In Model 1, w_2 is always zero, and also w_3 is almost always zero. With these learned parameters, Model 1 is effectively equivalent to Amdhal's law [32], except for Grep. Also, in Model 2, w_2 is close to zero except for Grep, which means the learned models are almost linear except for Grep.

Table I
LEARNED PARAMETERS FOR MODEL 1

Benchmark	w_0	w_1	w_2	w_3
Grep	0.0119	1.0111	0.0000	0.0002
Rolling Count	0.2076	1.3401	0.0000	0.0000
Unique Visitor	0.0459	1.2375	0.0000	0.0000
Page View	0.0000	2.3239	0.0000	0.0000

Table II
LEARNED PARAMETERS FOR MODEL 2

Benchmark	w_0	w_1	w_2
Grep	0.0000	1.0676	0.0280
Rolling Count	0.5011	0.2914	0.0000
Unique Visitor	0.0000	0.6776	0.0014
Page View	0.0000	0.6897	0.0038

Prediction results for all VM configurations are shown in Figure 6 and 8 for Model 1 and Model 2 respectively. Also, prediction accuracy results for M_{test} normalized by (predicted / actual) MST are shown in Figure 7 and 9 for Model 1 and Model 2 respectively.

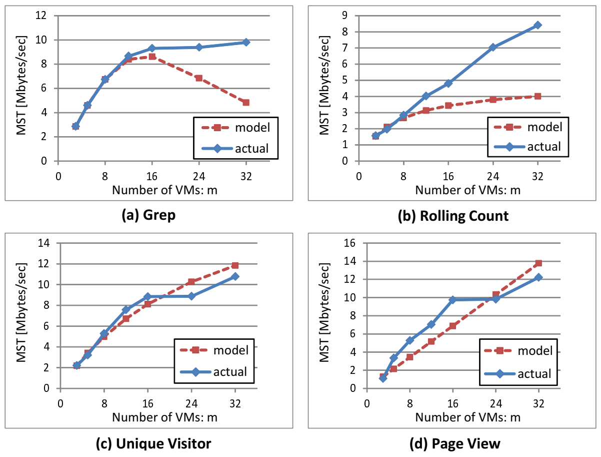


Figure 6. Maximum sustainable throughput prediction results obtained from Model 1.

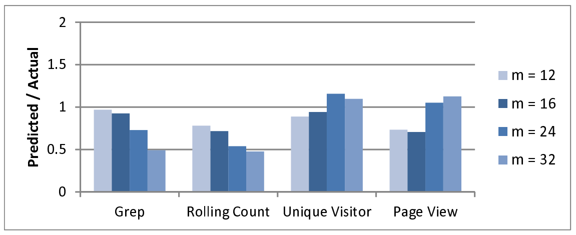


Figure 7. Normalized prediction accuracy obtained from Model 1.

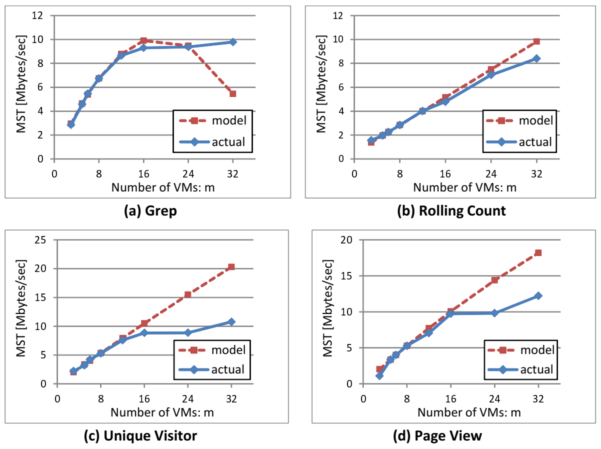


Figure 8. Maximum sustainable throughput prediction results obtained from Model 2.

Model 1 shows relatively good accuracy for Unique Visitor and Page View with error up to 29%; however, it

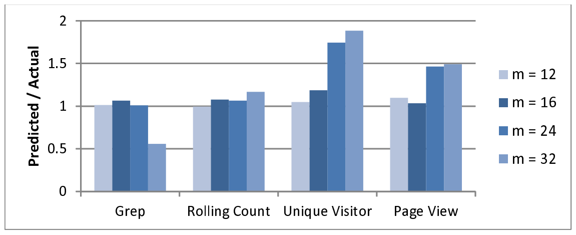


Figure 9. Normalized prediction accuracy obtained from Model 2.

does not work well for Grep and Rolling Count except for $m = 12, 16$. One of the reasons of these large errors is due to the fact that we apply linear regression to the inverse of $\hat{\tau}_{\text{ms}}$ as shown in Equation (7). Even though we minimize the training error in Equation (7), the error is magnified in the original Equation (4). This effect can clearly be seen for Page View with $m = 5, 8$ in Figure 6(d).

As we can see from Figure 8 and 9, Model 2 can predict up to $m = 16$ well (max error 18% for Unique Visitor, avg error 9%); however, the error becomes larger for $m = 24$ (max error 74% for Unique Visitor, avg error 32%) and $m = 32$ (max error 88% for Unique Visitor, avg error 49%) due to non-linear behavior of MST. Even though our models are non-linear, training samples obtained from M_2 almost linearly improve for Unique Visitor and Page View. That is why Model 2 cannot predict non-linear behavior for these two benchmarks. However, Model 2 captures the linear behavior of Rolling Count well and shows consistently good accuracy throughout all the VM configurations with error up to 17% when $m = 32$.

VII. Cost efficiency evaluation

We evaluate the cost efficiency of the trained models we have shown in Section VI using simulation-based elastic VM scheduling. In the simulation, we do not consider any overhead for VM allocation and application reconfiguration. Therefore, this evaluation is not fully realistic, but shows upper-bounds of cost efficiency that can be achieved by our models.

A. *Experimental settings*

The workload we use is based on the FIFA World Cup 1998 website access logs over three weeks time (500 hours) [33], as shown in the blue region of Figure 10. Since the original workload is presented in the number of access to the website, we convert the number of access to the input data rate. After this conversion, the peak workload is 7.7 Mbytes/sec and the average workload is 4.58 Mbytes/sec, so our models are applicable to this workload. We see this workload as a sequence of input data rates $\lambda(t)$, for $t = 1, 2, ..., 500$. Our SLA is to keep higher MST than input data rates (*i.e.* $\lambda \leq \tau_{\text{ms}}$ as shown in Section III-C).

Evaluation is simulation-based with the scheduling interval of one hour. For every t, a VM scheduler has a chance to allocate or deallocate one or more VMs without any startup time delay. We do not consider any overhead related to the stream processing system's reconfiguration either, such as rebalancing tasks. We simulate two benchmarks, Grep and Rolling Count, separately and assume each benchmark tries to process incoming events shown in Figure 10. We choose these two benchmarks since their scaling characteristics are quite different from each other. For each benchmark, we use Model 1 and Model 2 with the parameters in Table I and II respectively. We use the `m4.large` instance type of Amazon EC2 whose cost is \$0.12 per hour (as of November 2016). We compare the following scaling policies:

- *Ground truth*: Optimal scaling policy based on the actual measurement of MST (solid lines) shown in Figure 6(a) (or Figure 8(a)) for Grep and Figure 6(b) (or Figure 8(b)) for Rolling Count. Allocate the minimum number of VMs that satisfies incoming input data rate.
- *Static (peak)*: Static VM allocation policy that covers the peak load of 7.7 Mbytes/sec.
- *Static (average)*: Static VM allocation policy that covers the average load of 4.58 Mbytes/sec.
- *Elastic (Model 1, x%)*: Allocate the minimum number of VMs that satisfy the input data rate multiplied by $(1.0+x/100)$ based on the predicted MST (dotted lines) shown in Figure 6(a) (or Figure 8(a)) for Grep and Figure 6(b) (or Figure 8(b)) for Rolling Count. We test $x = 0, 5, 10$.
- *Elastic (Model 2, x%)*: Same as above except for using Model 2.

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

SLA violations are counted if the following condition is true:

$$\tau_{\text{ms}}(m) < \lambda(t), \tag{8}$$

where τ_{ms} is the ground truth function given by the actual measurement of MST, m is the allocated number of VMs as the result of a scaling decision, $\lambda(t)$ is the input data rate at time t. Violations are evaluated as the percentage against the total 500 scaling attempts over $t = 1, 2, ..., 500$.

B. *Cost efficiency results*

Table III and IV show results of average hourly VM usage cost and SLA violations for Grep and Rolling Count respectively. Both evaluations show that all the elastic policies have 0.0% SLA violation with 10% over-provisioning and with much lower cost compared to the Static (peak) policy. Notably, the elastic policy of Model 1 with 5% over-provisioning achieves 0% violations with just 5% more cost than the ground truth. Also, comparing to the "Static (peak)" allocation policy, this elastic policy uses 49% less cost per hour. These results show that with a reasonable percentage of over-provisioning, the proposed model's cost efficiency is on par with the optimal scaling policy.

Table III
AVERAGE HOURLY VM USAGE COST AND SLA VIOLATIONS FOR GREP BENCHMARK (EC2 INSTANCE TYPE: M4.LARGE).

Policy	VMs/hour	Cost/hour	Violations[%]
Ground truth	5.15	$0.62	0.00
Static (peak)	11.00	$1.32	0.00
Static (average)	5.00	$0.60	36.40
Elastic (Model 1, 0%)	5.12	$0.61	4.60
Elastic (Model 1, 5%)	5.41	$0.65	0.00
Elastic (Model 1, 10%)	5.69	$0.68	0.00
Elastic (Model 2, 0%)	5.04	$0.60	12.20
Elastic (Model 2, 5%)	5.34	$0.64	0.60
Elastic (Model 2, 10%)	6.00	$0.72	0.00

Table IV
AVERAGE HOURLY VM USAGE COST AND SLA VIOLATIONS FOR ROLLING COUNT BENCHMARK (EC2 INSTANCE TYPE: M4.LARGE).

Policy	VMs/hour	Cost/hour	Violations[%]
Ground truth	13.76	1.65	0.00
Static (peak)	28.00	3.36	0.00
Static (average)	14.00	1.68	43.00
Elastic (Model 1, 0%)	22.27	2.67	7.40
Elastic (Model 1, 5%)	23.33	2.80	0.80
Elastic (Model 1, 10%)	24.14	2.90	0.00
Elastic (Model 2, 0%)	12.99	1.56	61.40
Elastic (Model 2, 5%)	13.70	1.64	22.20
Elastic (Model 2, 10%)	14.42	1.73	0.00

Figure 10 presents the input data rate sequence and MST generated by elastic VM scheduling using Model 1 for the Grep benchmark, which shows the minimum violations with minimum cost among other models. We can see from the graph that the "Elastic (model 1, 5%)" and "Elastic (model 1, 10%)" generate slightly over-provisioned MST, but closely follow the input data rate.

VIII. DISCUSSION

In this section, we discuss the results of the experiments as well as our models' applicability to large-scale prediction and VM instance type heterogeneity.

A. Likely cause of the bottlenecks

From the experiments in Section VI, the scalability for Grep, Unique Visitor, and Page View 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 the MST is bounded by the performance of that single thread. In Unique Visitor and Page View, they show similar scalability patterns, which can be attributed to skewed web access log patterns. That is, since URLs in the log are not evenly distributed, a hash function fails to distribute the workload to worker nodes uniformly. Since our approach is application-agnostic, we do not attempt to diminish the load imbalance; however, we may be able to detect it to predict the performance bottlenecks. As we have seen in Figure 8(c) and 8(d), it is difficult to predict long term performance trends from a limited number of training samples. One way to improve the prediction accuracy is monitoring resource usage patterns over time and predict when they will hit the resource capacity.

B. Qualitative characteristics of prediction in elastic scheduling

In Section VI, we look at prediction performance for each VM configuration; however, when we use the model for elastic VM scheduling, the overall relationship between the model and actual MST becomes important. For example, in Figure 6(a), when we need MST of 6 Mbytes/sec, the model tells you either 6 or 30 VMs would give you that performance. Obviously, one would not choose 30 VMs since 6 VMs would give you the required MST. More generally, for elastic VM scheduling, accurate prediction of the peak performance (in this case 16 VMs) is important and prediction after the peak is less important. Another observation is that the relationship between the model and actual measurement decides whether we over-provision or under-provision VMs. As shown in Figure 6(b), to get MST of 4 Mbytes/sec, the model tells you to allocate 32 VMs, but it actually generates only about 8.5 Mbytes/sec. This means when the predicted MST from the model is below the actual, we over-provision VMs. Likewise, when the predicted MST is above the actual, we under-provision VMs.

C. Prediction for large-scale applications

We predict MST up to 32 VMs because that is the maximum number of VMs we are allowed to use on Amazon EC2. Looking at Figure 6, we can process data for typical use-case benchmarks with the throughput of about 10 Mbytes/sec (= 35 Gbytes/hour) using 32 VMs. This processing throughput would probably be enough for many regular data analytics applications. Given enough training samples, we believe our modeling approach itself is applicable to larger-scale applications that require more than 32 VMs. However, predicting MST for larger-scale applications leads to a larger prediction uncertainty. Thus, we need to improve prediction accuracy by implementing techniques such as online model updates from actual runs or the performance bottleneck detection as discussed in Section VIII-A

D. Homogeneous vs. heterogeneous VM types

We use the homogeneous VM type (*i.e.*, `m4.large` from Amazon EC2) in our MST prediction models. If tasks created from a stream application show different resource usage patterns, it is meaningful to use heterogeneous VM types for the performance optimization (*e.g.*, compute optimized VM

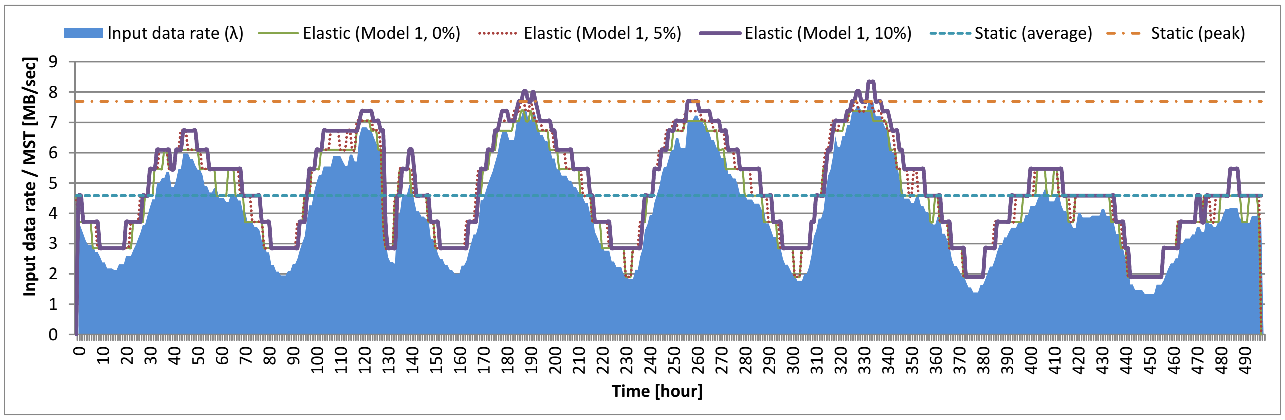


Figure 10. Input data rate sequence created from the FIFA World Cup 1998 website access logs and maximum sustainable throughput generated by elastic VM scheduling based on prediction model (Model 1) for the Grep benchmark.

for CPU intensive tasks, memory optimized VM for memory intensive tasks). As we have shown in Section IV-B, we control the parallelism of the target streaming application, but not the actual task assignment. It is determined by the stream processing framework's task scheduler. Especially in Storm, the default task scheduler tries to schedule resources evenly to application tasks. Thus, as long as we use such task scheduler, using homogeneous VM types is sufficient.

IX. Conclusion

We have presented a method to estimate the maximum sustainable throughput for public cloud-based stream processing applications. We identified a common data processing environment used by modern stream processing systems and proposed two models for MST prediction. The models use a homogeneous VM instance type and predict MST based on the number of VMs. For evaluation, we used Intel's Storm benchmarks running on Amazon's EC2 cloud. To minimize the time and cost for training, we statistically determined a set of training samples from running the application on a few VMs (up to 8). We use our model to predict MST for a larger number of VMs (in our experiments, up to 32). Our experiments show that one of our models can predict MST for Storm applications with less than 4% average prediction error for 12 VMs, 9% for 16 VMs, and 32% for 24 VMs. Further, we evaluated our prediction models with simulation-based elastic VM scheduling for a realistic data streaming workload. Simulation results showed that with 10% over-provisioning, the proposed models cost efficiency is on par with the optimal scaling policy without incurring any SLA violations.

For future work, we plan to integrate system resource usage metrics such as CPU, memory, and network utilization to the model to better predict performance bottlenecks. Also, we plan to apply the proposed models to other data processing frameworks such as Flink [10], [11] to confirm the applicability of our modeling approach. Other interesting future directions include online learning to improve the performance prediction model accuracy over time, the use of meta-algorithms such as boosting to construct a prediction model from multiple weak models, and large-scale performance prediction using a cloud environment simulator such as CloudSim [34].

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, in *IEEE 9th International Conference on Cloud Computing*.

[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] The Apache Software Foundation, "Apache Storm," http://storm.apache.org/, Accessed: 2017-02-15.

[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 *Proceedings of the 2014 SIGMOD International Conference on Management of Data*, 2014, pp. 147–156.

[10] The Apache Software Foundation, "Apache Flink," http://spark.apache.org/, Accessed: 2017-02-15.

[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] The Apache Software Foundation, "Apache Samza," http://samza.apache.org/, Accessed: 2017-02-15.

[13] ——, "Apache Spark," http://spark.apache.org/, Accessed: 2017-02-15.

[14] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth 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 *IEEE 35th 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 *IEEE 35th International Conference on Distributed Computing Systems*, 2015, pp. 411–420.

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

[21] Intel Corporation, "Storm benchmark," https://github.com/intel-hadoop/storm-benchmark, Accessed: 2017-02-15.

[22] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th ACM Annual Middleware Conference*, 2015, pp. 149–161.

[23] 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.

[24] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 1–16.

[25] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015, pp. 239–250.

[26] Microsoft, "Measuring maximum sustainable engine throughput," https://msdn.microsoft.com/en-us/library/cc296884(v=bts.10).aspx, Accessed: 2017-02-15.

[27] J. Kreps and L. Corp, "Kafka : a distributed messaging system for log processing," *ACM SIGMOD Workshop on Networking Meets Databases*, p. 6, 2011.

[28] O. Etzion and P. Niblett, *Event processing in action*. Manning Publications Co., 2010.

[29] Yahoo! Inc., "Yahoo streaming benchmarks," https://github.com/yahoo/streaming-benchmarks, Accessed: 2017-02-15.

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

[31] Mark Twain, "The project gutenberg ebook of the adventures of tom sawyer," https://www.gutenberg.org/files/74/74-h/74-h.htm, Accessed: 2017-02-15.

[32] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the Spring Joint Computer Conference*, 1967, pp. 483–485.

[33] 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.

[34] 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.