Forecasting a Storm: Divining Optimal Configurations using Genetic Algorithms and Supervised Learning

Michael Trotter, Timothy Wood
The George Washington University
Washington D.C., USA
{trotsky, timwood}@gwu.edu

Jinho Hwang
IBM T.J. Watson Research Center
New York, USA
jinho@us.ibm.com

***Abstract*—With the advent of Big Data platforms like Apache Storm, computations once deemed infeasible locally become possible at scale. However, doing so entails orchestrating powerful yet expensive clusters. With its focus on stream processing, Storm optimizes for low-latency and high throughput. However, to realize this goal and thereby maximize the utility of these clusters' resources, operators must execute these tasks under their optimal configurations. Yet, the search space for finding such configurations is so vast and time-consuming to explore so as to be effectively intractable due to issues like the temporal overhead of testing new candidate configurations, the sheer number of permutations of parameters within each configuration and their interdependence among each other.**

In order to efficiently cover the search space, we automate the process with genetic algorithms. Moreover, we fuse this technique not only with additional cluster information gleaned from JMX profiling and Storm performance data but also with classifiers constructed from training data from past executions of a plethora of Storm topologies. Utilizing a diverse set of Storm benchmark topologies as evaluation data, we show that the fully enhanced genetic algorithms can efficiently find configurations that perform on average 4.67x better than "rules of thumb"-derived manual baselines. Moreover, we demonstrate that our fully refined classifiers enhance the GA throughput on average across the topologies by 22% while reducing search time by a factor of 6.47x.

***Keywords*-Apache Storm; Genetic Algorithm; Supervised Learning; Autonomic Performance Tuning**

I. Introduction

Within the realm of Big Data, computations can occur on batches or streams of inputs [1]. A popular framework for handling the latter case is Apache Storm which is capable of "doing for realtime processing what Hadoop did for batch processing" in a scalable, fault-tolerant way, and with ease in terms of both administration and programming [2]. However, optimizing the configurations for Storm applications or *topologies* is far from easy.

Indeed, there are many parameters within a Storm configuration with which to experiment, chief among them being the number of worker processes and individual threads of parallelism within a given process [3]. Setting these parameters too low risks leaving the cluster's resources idle while handicapping the performance of the topology [1]. Meanwhile, setting them too high invites the various components of the topology to compete with one another for resources, causing problems like thrashing and page faults [3]. As these components of a Storm topology share the same cluster resources, they are interdependent on one another and consequently produce shifting bottlenecks when their associated parameters change [3]. Given that this task of finding an optimal configuration is an NP-Hard problem, an automated solution would have significant utility [4].

Here we describe Forecaster, an integrated, online tuner utilizing one of several possible methods for exploring this search space: genetic algorithms. We show that it does well even unmodified compared to the "rules-of-thumb" baseline commonly used by system administrators today. Forecaster yields configurations whose throughput is a factor of 4.59x on average greater than the baseline using our testing topologies. We refine this tuner by guiding its evolution by incorporating heuristics and additional information from the Java Metrics Extension (JMX) and Storm Nimbus service, achieving a throughput improvement of 1.23x to 19.5x compared to the baseline. Nevertheless, the time required to reconfigure a Storm topology and gauge the impact on performance naively necessitates running the tuner for over a day in some cases. Thus, we seek to further refine this process using supervised learning to further reduce the search space. Finally, we demonstrate the portability of this enhanced tuner not only in regards to different types of clusters but also to a variety of Storm topologies running on said clusters. In particular, we make the following contributions:

- We design a genetic algorithm-based approach to efficiently search through Storm topology parameter sets.
- We optimize the search process with classification models that predict configuration performance and filter out candidate configurations deemed unlikely to produce good performance results.
- We describe a method to map an arbitrary topology's structure to that of the training topologies so as to enable evaluation by the regressors and classifiers.
- We propose a retraining scheme to enable portability of the models between clusters with different architectures.

II. Background

A. Apache Storm

Originally developed to perform top-*n* analysis for the trending topics of tweets, Apache Storm is a popular framework for processing real-time streams of data rather than the traditional batch processing approach of MapReduce [5]. The equivalent of a batch job in MapReduce is a *topology* which represents a directed acyclic graph (DAG) of control flow wherein data in the form of *tuples* originates from the root nodes or *spouts* and flows through one or multiple nodes or *bolts* [6]. A single *stream* of execution thus flows from a source spout through a series of bolts before terminating, and a topology has one or more of these streams [7]. A cluster can run one or more topologies concurrently as long as all of the worker processes assigned to the topologies does not exceed the worker processes available to the cluster [6]. Figure 1 demonstrates a topology with many such streams feeding into a singular sink node. To implement this logical representation of computation flow, Storm orchestrates a variety of system processes.

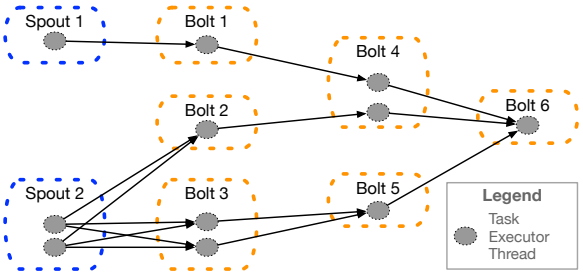

Fig. 1: An example Storm topology.

Of note are the Storm Nimbus, Supervisor and Worker processes whose relationships are shown in Figure 2. The Nimbus process schedules work to be done throughout the cluster and maintains application metrics for a given topology and its components [1]. Nimbus broadcasts these assignments via Apache ZooKeeper [7]. With their assignments in hand, the Supervisor processes forward them onto the Worker processes while ensuring these processes remain alive via regular heartbeat checks [5]. Within each Worker process, one or more executor threads perform the actual task logic as specified by the application code [2]. Application operators are responsible for specifying the topology's number of Worker processes as well as the number of executor threads for each logical spout or bolt at run time (i.e., the spout or bolt parallelism), although they can later update these parameters in real-time using the Storm cluster utilities [8]. Operators can thus manually tune topologies, a feat easier said than done due to the complex interactions between components and large search space [4].

To illustrate this quandary, we present the issue of tuning one of our training topologies: WordCount. The following components comprise its structure: a spout which continuously emits a random sentence from a predefined corpus, an intermediary bolt which splits that sentence into the words which comprise that sentence, and a sink bolt which aggregates the words into their counts as seen by the topology as a whole. Table I lists out the tunable parameters of the WordCount topology. With any given topology, the operator must specify not only the number of processes from the entire cluster to assign to the topology but also the number of threads within those processes for the components that comprise a topology, i.e. the spouts and bolts individually. Finally, some topology-agnostic parameters must be set, such as the number of "acker" threads used to implement Storm's at-least once processing guarantees by having the spout resend a given tuple into the stream of descendent components in the event of failure.

Parameter	Meaning
Workers	Number of processes distributed across the entire cluster assigned to processing a topology within its executor threads
Sentence Spout parallelism	Number of executor threads assigned to emitting sentences into the rest of the topology
Sentence Splitter bolt parallelism	Number of executor threads assigned to the execution of bolts which take in sentence and split them on whitespace
Counter bolt parallelism	Number of executor threads assigned to the execution of the terminal bolts which calculate the frequencies of individual words
Acker parallelism	Number of executor threads assigned to the acker (Required for all Storm topologies)

TABLE I: : Tunable parameters of the WordCount Topology

In a given cluster composed of 48 total cores across four worker machines running a single WordCount topology bound by the limitations of the Storm API and keeping with Storm best practices for topology parameter tuning, there are a total of 48 permutations for the number of workers and 255 possible configurations each for the parallelism values for the sentence spout, intermediary sentence splitter bolt, counter sink bolt and acker, yielding a total of 202,956,030,000 possible permutations [6], [9]. A programmer could easily extend WordCount to count the letter frequencies instead of the word frequencies by adding another bolt between the sentence splitter and counter bolts to split the words into individual letters and thus increase the number of permutations by a factor of 255. Given that Storm allows for arbitrary number of spouts and bolts in a topology so long as they form a DAG, the number of configuration permutations can quickly become intractable as the number of configurable parameters grows. Accordingly, the issue of exploring the search space is an instance of the NP-Hard Knapsack optimization problem, prohibiting finding a quick and accurate solution in all but the simplest of configurations [4].

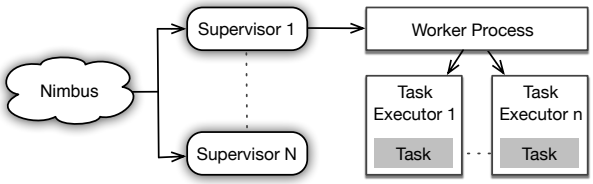

Fig. 2: System Architecture of Storm.

B. Rules of Thumb Baseline

There are several "rule of thumb" heuristics for preparing Storm configurations [6]. Due to the overhead involved with context switching especially with time sensitive applications like Storm, an executor should not execute more than one task at a time [3]. For the same reason, the sum of the number of threads for CPU-bound bolts and spouts should also not exceed the total number of cores across the cluster [6]. However, IO-bound bolts and spouts need not obey this rule since they would most likely perform a context switch while waiting on IO [3]. As it consumes very few resources, the *acker* thread needs only one thread per Worker process to send its acknowledgment or "ACK" tuples backwards from the sink nodes to the spouts as part of Storm's implementation of at-least once processing guarantees [8]. Without further refinement, our results show that these rules can provide subpar performance compared to a carefully tuned setup.

C. Bayesian Optimization

A common method to attain such refinement in an automated manner is Bayesian optimization, a probabilistic approach to finding optimal solutions of a problem [3]. Bayesian optimization models the search problem as a Gaussian process defined by a joint probability distribution of random variables representing each of the configuration parameters [10]. By sampling the current performance of a possible candidate configuration and combining that information with the history of past candidate configurations and their respective performance values, Bayesian optimization is able to iteratively update a posterior probability model of the performance values given the configuration values [11].

Existing Bayesian optimization frameworks like Spearmint use kernel density estimators and the posterior probability model to find promising new candidate configurations to test [10]. Ideally, these generated configurations should balance the trade-off between exploration of the global search space and refinement of the current candidate configuration values [3]. However, our preliminary exploration of the popular Bayesian framework Spearmint demonstrated that Bayesian optimization is a poor fit for tuning Storm's parameter values [12]. Our results in Section V show that Spearmint's search follows a path more akin to a random walk.

Bayesian optimization relies on two assumptions to work: a lack of covariance among the parameters of a configuration, and those parameters are each of a continuous range of values to set [3]. Unfortunately, those assumptions do not hold for this task. The configurable parameters are far from independent of one another. Given that the resources assigned to the topology are finite, there is a constant risk of resource competition among the components to use for processing, including CPU time, memory and I/O. Moreover, Storm's default round-robin scheduler naively ignores the resources a given a bolt or spout needs to implement its processing logic. Instead, it simply tries to ensure that the executor threads assigned to the topology are running roughly the same total number of spout and bolt instances. Lastly, the operator can only set each of the parameters to discrete positive integer values, prohibiting additional automatic refinement by Spearmint. Thus, Spearmint is never able to find a configuration around which to converge and refine. Consequently, we must explore alternative approaches to the parameter search problem.

D. Genetic Algorithms

A more comprehensive approach to discovering optimal configurations is to use genetic algorithms (GAs). Inspired by natural selection, GAs arrive at a such configuration via cycles consisting of individual population generation, selection, crossover and mutation phases [13]. During the first phase, the current population of candidates originates from a combination of the offspring and survivors of the previous generation or a randomly generated configuration set [14]. The population then faces two selection phases which first decide which candidates do not survive into the next generation and then decide which candidates may produce child candidates [15]. This filtering uses a fitness function evaluation metric tied to the configuration itself in addition to some evaluation strategy such as allowing only the top-n fittest members within subbatch of candidates to survive or randomly selecting survivors in proportion to their fitness [13]. With the parents chosen, they can combine their own configurations to produce new child configurations in the third phase using one of several strategies such as the uniform crossover's random selection of parent configuration elements or a single point crossover's swapping of parent configuration elements around a single pivot [15].

To ensure that the GA does not stall at a local maximum, mutations randomly alter configurations within the population so as to encourage additional exploration of the search space [14]. These four phases loop continuously until it meets a termination condition like total run time or convergence threshold for average fitness [15]. Since this exploration of the search space can require a significant amount of time, filtering out bad configurations without actually evaluating them as part of a candidate population is vital to achieving good performance.

E. Support Vector Machines

Our approach uses classification algorithms to implement this filtering optimization. Although we consider a plethora of supervised learning techniques over the course of our research including decision trees, neural networks and nearest neighbors algorithms, we ultimately settle on Support Vector Machines (SVM) as our means of classification. SVMs remap the original input into higher dimension space and attempt to find the optimal hyperplane which best separates these translated data points [16]. Using this hyperplane, they can then partition the points into the groups defined with supplied dataset labels [17]. To prevent overfitting, the hyperplane is typically subject to a regularization constraint or "alpha" value [17]. Since the translation into the higher dimension space is non-linear, SVMs support broad, general relationships

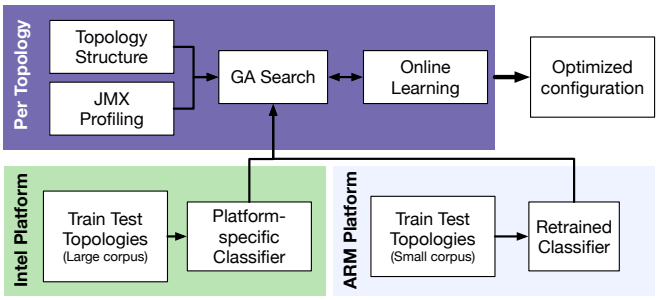

Fig. 3: Forecaster usage on two clusters.

that need not be linear themselves. This flexibility enables SVM to outperform other techniques like decision trees and neural networks in cases such as classifying digits from USPS mail data [16]. Furthermore, the complexity of the hyperplane calculation is a function of the number of iterations it runs and the number of feature vectors, making SVM relatively cheap compared to those methods especially the latter [16]. Given the broad applicability, speed and the ability to retrain SVMs, we choose it as the classifier in our ultimate implementation.

III. Design

As shown in Figure 3, Forecaster combines our process into an overarching system that: 1) uses GAs to search through each new topology's parameter space represented by the number of worker processes and the individual thread parallelism values of its spouts and bolts as single configuration to test, 2) optimizes the search process by avoiding configurations classified as unlikely to be effective, and 3) facilitates transitioning between physical clusters by allowing the models to be retrained on new hardware.

A. GA Parameter Search

As described previously, GAs provide a convenient way to intelligently search through a large search space given the means to quickly reduce its size and to benefit from the history of the search thus far. Otherwise, GAs have difficulty converging to an optimal configuration. Therefore, we enhance the standard GA in the following ways.

Initial Population: We generate a random population of configurations to begin the GA search process. To avoid obviously poor choices, we constrain the population generation to adhere to the "rules of thumb" like limiting the number of worker processes to the number of CPU cores and setting the number of "acker" threads to the number of worker processes. We use an initial population size of 50 since our tests show it provides a good balance of result quality and convergence time.

Execution Monitoring: Forecaster deploys each configuration on the Storm cluster and observes it for a period of five minutes to evaluate its effectiveness. We find that shorter time periods lead to inaccurate results due to Storm's unstable warm-up period. During each trial, Forecaster monitors the test environment by utilizing the internal Storm metrics maintained by the Nimbus process and the JMX interfaces running on the Storm worker processes. The Nimbus data includes information about the throughput at both the cluster-wide topology level and at each individual worker process running in that cluster. It also provides information about the topology's logical structure. With the JMX information, we are able to enrich this view with data about the CPU load, memory usage and other associated profiling data.

Fitness, Crossover, and Mutation: The measured throughput of the Storm topology is used as the fitness metric for selecting parameter sets which "breed" and produce the next population generation. We use a custom single point crossover whose implementation details are described in the next section. Its ability to persist good parameter subsets across generations enables the GA to converge at a high fitness. Due to Jenetics requiring that configurations be represented as a numerical array, we represent the entire parameter set for a configuration as a singular yet consistently ordered integer array. This array structure in turn enables pivoting at a given index for mixing and matching the parent configurations.

Selection: We enhance the standard GA scheme to filter out configurations that are unlikely to be effective so as to limit the amount of actual live testing needed. Using the monitor data derived from the JMX interface, Forecaster characterizes each of the Storm components as CPU-bound, IO-bound, memory-bound or not bound based on computations from a given component's system load averages per processor, utilization of any of Java's core IO-bound APIs and committed memory compared to total memory usage. Then, utilizing our Storm "rules of thumb" for each of these, the system applies a series of heuristics which invalidates proposed members of the population known to cause performance issues like exceeding the number of processing elements for the entire cluster for CPU-bound workloads or overallocating too many acker threads. We further enhance the selection process by employing a trained classifier for additional filtering as described in the next section.

Convergence: After filtering the topology, we allow the GA to run normally by deploying and monitoring each of the more promising candidate configurations in the population. Depending on the supplied configuration, this process continues until the population fitness, i.e. throughput, converges to some predefined threshold or the total execution exceeds the supplied timeout value.

B. Classifying Bad Parameters

Testing a Storm configuration is an expensive process, requiring several minutes for the deployment of a new configuration and its state to stabilize before measurement. Any means of avoiding testing fruitless configurations are therefore desirable. The filtering stage of our GA helps avoid obviously bad parameter sets, but it is only based on "rules of thumb" which do not always produce the best results.

Ideally, a model would estimate the performance of a configuration without necessitating its deployment. Then, we

could utilize efficient algorithms like Best-First Search and backtracking which rely on a heuristic to organize their searches effectively. However, we have found that predicting the throughput of different Storm parameter sets with regression models has poor precision and recall statistics due to the high variance in the measurements for a given topology and among different topologies. In short, regression models are particularly prone to overfitting. Addressing this limitation would require a fair amount of operator-level knowledge to successfully preprocess the raw features for a given topology to serve as input to the regressors. However, doing so runs counter to our goal of building a portable and generic tuner for Storm topologies. Instead, we opt for single-class classification: a more general model that only predicts whether a parameter set is "good" or "bad" but better handles the aforementioned issues.

The classifier poses a new trade-off: training a representative classification model takes time, but it can afterwards quickly predict the behavior for any new Storm application being deployed to the system. Given that the typical use case is of a cluster with relatively static hardware executing a diverse set of applications, we believe such a trade-off can yield benefits in many scenarios.

Training Corpus: To build the classifier, Forecaster first must gather training data for a range of Storm applications and parameter sets. We pick three representative Storm applications to illustrate that even a simple model can provide value for other application types: SOL, WordCount, and RollingCount. We select these applications because they stress the I/O, CPU, and memory aspects of the system respectively. For each training application, we construct a corpus from two sources. First, we randomly generate parameter sets from the search space, avoiding ones which violate our rule of thumb bounds as described above, logging the configurations and their resultant throughput. Second, we include similar log data from the configurations and throughputs observed from previous runs of our GA tuner.

Classification Models: We bifurcate the training configurations into "good" and "bad" with the classifiers with "good" being defined as being 80 percent or greater of the maximum throughput discovered in each topology training set. This threshold allows for ruling out a large number of configurations while also providing for a relatively balanced division of the training set, preventing one classification from dominating the other during training. We first tune the hyperparameters of the individual classifier using both the training data and the frameworks' built-in optimizers before selecting the classifiers which scored the best on both the test and unseen validation sets.

Forecaster automatically gathers and labels the training data set before feeding it to a supervised learning framework. This enables us to build a plethora of classifiers with little additional effort. The resulting classifiers take a parameter set for a new application as input and return a binary classification and a confidence score.

We exploit ensemble techniques for our classification, which allow multiple classification algorithms and models to be combined to build a better overall model, bolstering the f-scores, i.e. classification accuracy, as a result. We explore a plethora of classifiers trained our three training topologies utilizing a variety frameworks and algorithms to include in our final ensemble. With a diverse collection of classifiers in our ensemble, we further augment our ultimate classification by considering each classifier's classification (C) weighted in proportion to its confidence score of the decision (p_c):

$$\text{classification} = \begin{cases} 0, & \text{if } \frac{\sum_{c \in C | c=0} p_c}{\sum_{c \in C} p_c} \geq \frac{\sum_{c \in C | c=1} p_c}{\sum_{c \in C} p_c} \\ 1, & \text{otherwise} \end{cases}$$

By immediately discarding bad configurations without fully evaluating them, we are thus able to save time for the exploration of the search space and provide negative feedback to the GA with which to steer it away from similarly bad configurations.

Mapping Parameter Sets: Before a classifier built from the corpus of training application data can be applied to a new test application, the parameters need to be normalized and mapped together. This transformation is necessary because each Storm topology's configuration defines not only the number of worker processes it has but also each spout and bolt's own number of thread executors. Since Storm allows for an arbitrary number of spouts and bolts to form a topology's DAG, the complexity of a given configuration thus varies. We handle this dimensionality problem via the following categorization technique.

Forecaster translates the parameters of the configurations of an arbitrary topology to those of the training corpus by examining its structure. Each topology's components falls into one of three categories: a terminal sink, an intermediary processor, or a originating spout. A spout element has no input streams and emits tuples to one or more output streams. Conversely, a sink element has no output streams and receives tuples from one or more input streams. Finally, a processor element has both input and output streams of tuples. A stream of processing thus flows from a given spout through zero or more intermediary processor bolts before ending at one or more sink bolts.

Each of the training corpus topologies consists of a singular flow of execution from a spout through a processor bolt to a sink bolt. Thus, our generic Storm configuration used for tuning assumes that all topologies match this flow. We use this configuration as the input to the objective function for evaluating the impact of a configuration on the overall throughput of the topology. Yet, even the test topologies do not all match this structure, forcing us to devise a simple scheme for mapping a test topology's structure to that of the training topologies.

Our strategy is an admittedly simple one and the subject for future potential research. For test topologies lacking a processor bolt, we merely assign zero for its value in our generic

configuration. For example, one of our training topologies, CSNBT, consists of a spout and a sink (NullBolt) without an intermediary bolt in between the two. We therefore apply zero for the intermediary bolt throughput for its mapping. For topologies with more than one spouts, processor bolts or sink bolts, we compute the respective average parallelism of each of these elements' thread counts. We then use these averages as the values as the respective values of the sink, processor and spout elements of a configuration to evaluate. For the letter counting topology example from the background section composed of a sentence spout, an intermediary sentence splitter bolt, another intermediary word splitter bolt and a counter sink, we would use the average of the two intermediary bolts' randomly generated parallelism values as the value of our generic configuration's intermediary bolt parallelism. This algorithm enables us to utilize our training corpus to accurately classify many test topologies while also binding the search space for quicker results. Our exploration of more advanced techniques like regularization and deeper classification of individual components based on profiling and performance characteristics yields thus far middling results and remains a topic of future work.

C. Transitioning to New Clusters and Topologies

We train each classification model on a specific hardware cluster that may differ significantly from other clusters. Similarly, each model incorporates the characteristics of our three training topologies which may not be fully representative of all Storm applications. To augment the generality of the models, we utilize classifiers that support partial fitting so as to enable retraining (for portability to new clusters) and online learning (for adapting to new topologies).

When Forecaster executes on a new physical cluster, it first reruns the three training topologies. Instead of gathering a full training set, Forecaster collects a much smaller amount of additional data to retrain the original classifier. Indeed, our evaluation shows that even with this smaller training corpus, retraining allows the original models to adapt to substantially different hardware architectures, e.g. from a high powered Intel cluster to a low power ARM-based one.

Second, Forecaster can use the results of a test topology to improve the classifier in an online manner. As the GA executes the test topology, the results of each parameter set can be fed back to the classifier as new training samples, improving accuracy over time on a wide range of topologies and cluster architectures.

IV. Implementation

We deploy Clients on each machine in the cluster to gather data from the Storm Nimbus process and the JVM processes of the Storm workers. A Tuner component resides on a separate node and runs the genetic algorithm and classification functions. When the Tuner issues a reconfiguration command to the clients, they will update the running topology with a new configuration. Unfortunately, measuring the fitness of a new configuration is far from instantaneous, slowing the progression of the genetic algorithm. This bottleneck exists because the reconfiguration command causes enormous variance in the topology's performance measurements as the topology adjusts to the new configuration before ultimately settling on a more consistent throughput statistic after about five minutes of execution.

As the GA may test hundreds of configurations, this high measurement cost per configuration emphasizes the need to avoid testing configurations that seem fruitless. Forecaster's Classifier service is composed of a series of models built in scikit-learn, Tensorflow and Keras. We utilize these frameworks since they share similar APIs and are ubiquitous in the data science community. Furthermore, the frameworks' classification and regression APIs are very similar to each other and thus enable a speedy evaluation of the regression and classification approaches with minimal effort to switch from one versus the other.

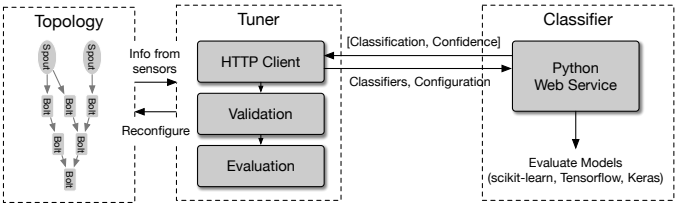

Fig. 4: Forecaster Implementation.

Due to Python being the language of choice for these frameworks, along with their limited support for Java, we utilize the Flask library's simple API to compose a web service in Python and use the Apache HTTP client in Java to enable interoperability between the Tuner and the Classifier web service. Figure 4 illustrates Forecaster's implementation. The Java client lists the classifiers to use in the ensemble classification along with the configuration in the integer array format used by the GA to evaluate in its HTTP POST payload to the web service. The service then loads the models organized into separate file system directories and performs the prediction, returning the classifications and their associated confidences back to the Tuner via HTTP. The Tuner then computes a weighted average of these classifications using their confidences for the ultimate classification evaluation as part of the selection phase of the genetic algorithm.

We utilize the Jenetics [15] library for our implementation of genetic algorithm search. The selection phase is modified to include our classification component to discard bad configurations during validation. The evaluation phase is where the Tuner deploys the actual configuration and calculates its fitness, i.e. throughput, based on the information provided by the sensors. To quickly eliminate poor configurations, we use a tournament selector which divides the population into subgroups and selects the fittest members of each to survive. We use three subgroups in our implementation. To ensure diversity in the resulting child generations, we use a roulette wheel selector to choose parents from the current population, utilizing a single point crossover. To do so necessitates representing the configurations as an integer array with a consistent

one-to-one mapping of each of the configuration parameters. Jenetics models each element of the array or parameter as a "gene" and an entire configuration set of parameters as a "chromosome." Thus, the crossover can consistently split the parent chromosomes and generate the children chromosomes from the splits. The mutation phase also uses this structure when randomly choosing a gene and altering the integer stored there. The GA runs until its average fitness stabilizes within ten percent of itself for five iterations. Due to only having a single cluster with which to evaluate configurations and consequently being limited to only testing a single configuration at any given time, we run Jenetics serially despite its parallel optimizations.

V. Evaluation

Our evaluation of Forecaster demonstrates not only can the system yield significantly better configurations than the "rules of thumb" control and a Bayesian optimization-based tuner but can do so in a timely and portable manner. To assess both the quality and timeliness of the derived configurations, we compare the control against our Spearmint implementation, our heuristic-based guided GA, our GA guided by the offline-only trained classifiers, and our GA algorithm guided by the offline classifier retrained with online data. To evaluate the quality of the individual classifiers and the impact of the training corpus' size on them, we train the classifiers with subsets of varying sizes from the same corpus and compare the resultant F1 scores. Finally, we evaluate the effect of retraining the classifier to a new cluster so as to analyze the portability of the system and ultimately demonstrate the full utility of our system. We make use of the following two clusters:

Intel Cluster: five machines each with an Intel(R) Xeon(R) six core E5-2420 CPU running at 1.90GHz, 16GB of memory running Ubuntu Linux 14.04 LTS with Storm 1.1.0, ZooKeeper 3.4.8, and Oracle JDK 1.8 Update 111.

ARM Cluster: five machines each with a low power 64-bit ARMv8 (Atlas/A57) eight core CPU at 2.4 GHz, 64GB of memory running Ubuntu 18.04 LTS with Storm 1.1.0, ZooKeeper 3.4.10 and Oracle JDK 1.8 Update 171.

A. Comparison with Spearmint Bayesian Optimizer

With multiple avenues for the direction of Forecaster in mind, we initially construct two competing approaches, a Spearmint-derived tuner and heuristic GA tuner with no classifiers, inspired by the approaches taken in state of the art applications [3], [11]. Thus, we originally pit the tuners against each other to see which is worthy of additional refinement and research. In particular, we utilize the Intel Cluster running each of the training topologies one at a time. We run the heuristic-GA tuner until the population's average fitness converges within one percent of itself and Spearmint for a hard limit of twelve hours since the latter lacks the terminating semantics of the former. The heuristic GA tuner is able to consistently albeit slowly find a good configuration around which to converge as shown in Figure 5 with the SOL topology. In contrast, Figure 6 shows that Spearmint is unable to account for the dependencies of the configuration parameters, leading to a random exploration of the search space that does not produce as high quality a result.

Although both the heuristic GA and the Spearmint tuner outperform the baseline as shown in Figure 7, the heuristic GA consistently outperforms the Spearmint tuner in each case. The GA approach provides between a 1.1x to 2.2x improvement in throughput compared to Spearmint.

Given that the heuristic GA produced higher quality results than the Spearmint implementation consistently, we thus focus Forecaster solely around the GA and utilize machine learning techniques to mitigate its major drawback: the long evaluation time required to converge around a good configuration.

B. Comparison of the Classifiers

Although our system supports classifiers built with scikit-learn, Tensorflow and Keras, we ultimately focus our analysis on the scikit-learn classifiers due to the high resource cost of training and retraining the latter two frameworks, in addition to the their additional hardware, library dependency installation and cumbersome API concerns. The scikit-learn classifiers capable of retraining include the Bernoulli Naive-Bayes, Stochastic Gradient Descent boosted (SGD)-optimized SVM, Perceptron, Passive Aggressive and Multi-Layer Perceptron (MLP) classifiers from scikit-learn. We refine the hyperparameters of each of these classifiers using scikit-learn's *GridSearchCV* tuner before training them on a separate offline corpus derived from the training topologies: RollingCount, SOL and WordCount.

In brief, our results show that the majority classifiers need a corpus of fewer than 500 training examples before their F1-scores converge. The SGD-boosted SVM and MLP classifiers are the most useful classifiers, converging around an average F1 score of 0.85 with a corpus size of 500 or greater as shown in Figure 8. An F1 score of 1 represents perfect accuracy in terms of both precision and recall. As noted in our discussion of Spearmint's poor performance, the Bayesian assumption of feature independence does not hold with this application due to the interdependencies of the Storm components and their configurations. Thus, the Naive-Bayes classifier has difficulty tuning Storm, with an F1 score consistently lower than the others at 0.75 as shown in Figure 8. Likewise, the throughput of a topology in relation to the parallelism of its threads and processes does not follow a linear equation. Consequently, the Passive Aggressive and Perceptron classifiers are poor fits for this task, achieving average F1 scores as low as 0.5 as shown in Figure 8. Since the SVM classifier is simpler and quicker to tune than the MLP classifier while also achieving comparable results, we ultimately comprise our ensemble classifier of SGD-optimized SVM classifiers individually built on each of our training topologies using an alpha value or regularization constant of 0.1 as informed from our hyperparameter tuning.

C. Comparison of the Tuners on the Intel Cluster

To begin our evaluation, we first use a manual configuration adhering to the Storm "rules of thumb" in its operators' guide to establish a baseline. Next, we run our standard GA until

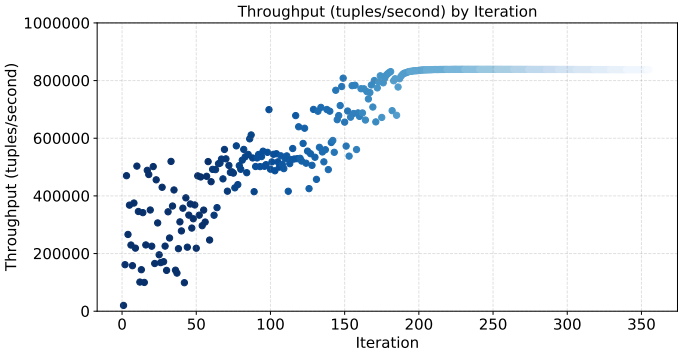

Fig. 5: SOL Exploration by the Convergence-Limited GA.

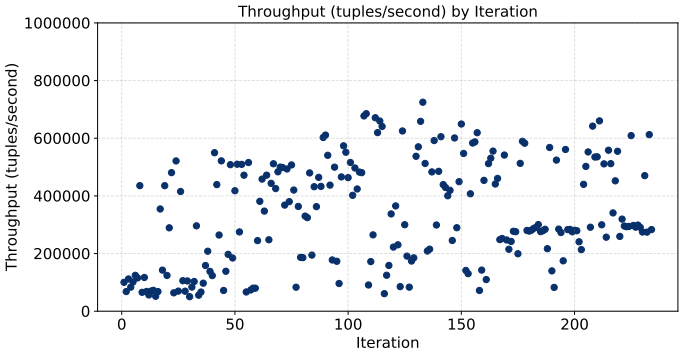

Fig. 6: SOL Exploration by Bayesian Optimizer.

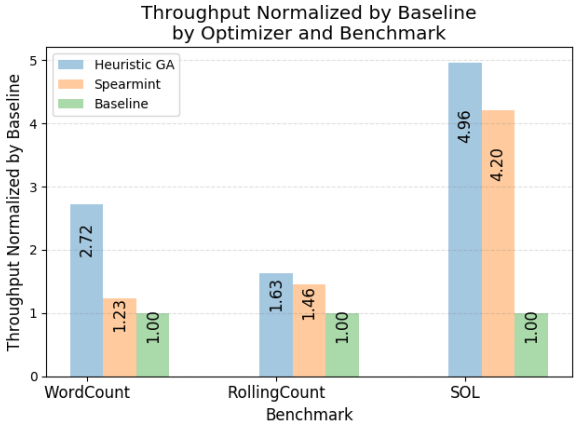

Fig. 7: Throughput Normalized by Baseline.

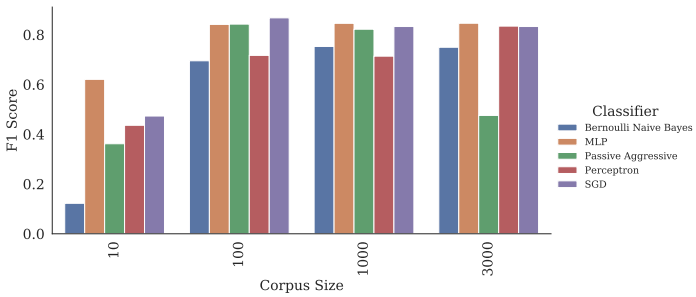

Fig. 8: Classifier F1 Scores vs Corpus Size for WordCount.

convergence as described beforehand. Afterwards, we combine the GA with our classifiers built with a large set of configurations from our three training topologies. To build these classifiers, we first assemble a set of training corpora of our three main topologies (WordCount, SOL and RollingCount). Henceforth, we refer to these topologies as **WC**, **SOL** and **RC** respectively. For each of these training topologies, we generate between 5000 and 7500 test configurations to train each classifier.

Finally, we run the GA again with classifiers retrained with a small data set consisting of fewer than 500 examples of each of the training topologies. They include ConstSpoutIdBoltNullBoltTopo, ConstSpoutNullBoltTopo, ResourceAwareWordCount and SlidingTupleTimestamps, in addition to our training topologies. For brevity's sake, we refer to those topologies hereafter as **CSIBNBT**, **CSNBT**, **RAWC** and **STT** respectively. We use online learning with the test topologies, thereby utilizing transfer learning to refine the classifiers built on WC, SOL, and RC for each of our test topologies.

The results are shown in Figures 9 and 10 in terms of throughput and execution time respectively. For throughput, Forecaster's tuners significantly outperform the "rules of thumb" control for most topologies. In the worst case, the GA-guided CSIBNBT outperforms the control configuration by only a factor of 1.1x, but RAWC improves upon it by 19.5x when employing the retrained classifier. In some cases, the GA alone outperforms the classifier-based approaches because it explores a wider search space although at a steep cost in search time.

Due to WC's much larger corpus size of circa 8500 samples versus the sizes of SOL and RC's corpora of roughly 4600 and 5600 respectively, the classifier for WC is better tuned than the others in the ensemble. Furthermore, the test topologies most like original training topologies benefit more from transfer learning since said classifier undergoes comparatively smaller changes during its refinement training. RAWC is a variation of WC while CSIBNBT shares some structural similarities with SOL, resulting in their good performance. Moreover, given the relative dearth of training examples especially for transfer learning and the dissimilarity of test topologies like CSNBT and STT compared with the training set of topologies, they do not see the immediate benefit in throughput that others like CSIBNBT see. Indeed, they suffer from underfitting. Although more training examples would alleviate that problem, they come with the price of additional data collection time. Nevertheless, there are still considerable time savings in using the classifiers compared to the default GA.

As shown in Figure 10, the classifiers yield a significant reduction in the time required for the GA to converge. Topologies like SOL and CSIBNBT take over 300 hours with the standard GA and see an improvement of greater than 40x in execution time for convergence compared with their default GA implementations. All of the GA + Classifier runs take less than 10 hours to complete, while achieving comparable throughput to the much slower GA. This improvement is also visible in how the evolution proceeds during the GA search. Figure 11 demonstrates the impact of the classifier by plotting

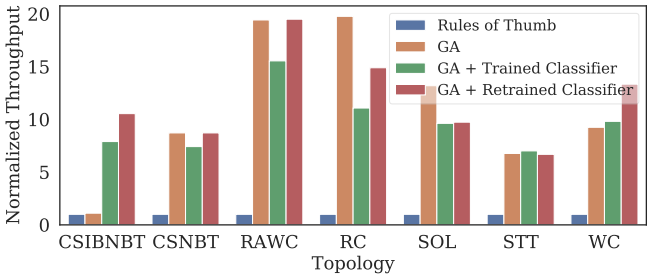

Fig. 9: Throughput for Tuners on the Intel Cluster.

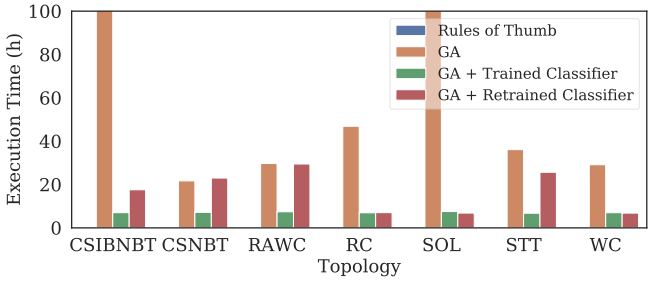

Fig. 10: Execution Time for Tuners on the Intel Cluster.

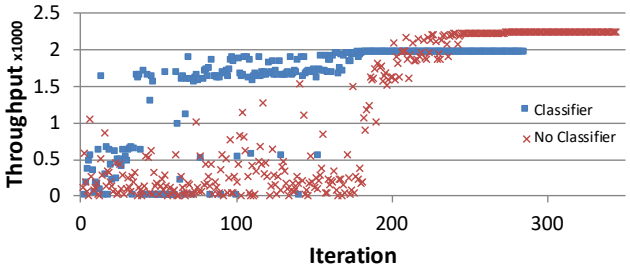

Fig. 11: GA Exploration with or without a classifier.

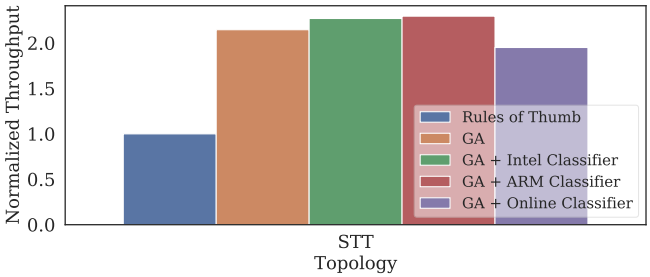

Fig. 12: Throughput of Tuners on the ARM Cluster.

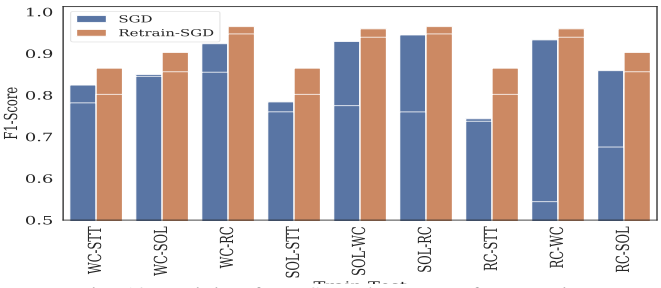

Fig. 13: Training from Scratch vs Transfer Learning.

the throughput achieved in each iteration of the search process. With the classifier enabled, Forecaster finds a desirable region of the parameter space in fewer than fifty iterations in stark comparison to the two hundred iterations without it.

Unfortunately, underfitting likewise affects the refined classifiers resulting in higher execution times particularly for the most dissimilar topologies in the test set. Regardless, they too almost always achieve quicker execution times while providing comparable throughput to the GA. Indeed, Forecaster is able to improve throughput of the seven topologies on average by 1.02x and 1.22x with the unrefined classifiers and refined classifiers enabled respectively and likewise reduce search time by 15.08x and 6.47x.

D. Porting the Tuners to the ARM Cluster

We next consider how utilizing transfer learning on our original classifiers can allow Forecaster to be applied to completely different cluster architectures. Our experiments thus consist of running two new variations on our low-power ARM cluster as opposed to our Intel-based Nimbus cluster: the GA with classifiers retrained on our three training topologies running on this new cluster and finally the GA with the classifiers refined with simulated online learning. We focus on the STT topology as the subject of our case study, as it is relatively dissimilar to our training topology set yet still derives a noticeable benefit from utilizing the classifiers.

Compared with the control tuner, the standard GA tuner achieves an improvement in throughput of roughly 2x on this new cluster. Although the same GA tuner achieves a much greater improvement of nearly 6.78x on the training cluster, the training cluster has additional and more robust CPU cores. The GA using our original classifiers derived from the training cluster improves upon the throughput of the GA by a factor of 1.05x. Using the classifiers refined using the three training topologies on the new cluster yields an improvement versus the original GA of a factor of nearly 1.07x. Unfortunately, the further training via simulated online training fails to improve upon the original GA's throughput and instead yields a throughput roughly .91x of the original GA's. Figure 12 summarizes these observations with the throughputs normalized to that of the "rules of thumb" tuner.

Underfitting is particularly to blame for the last classifier's inability to improve upon the original GA performance. This issue is of particular concern especially given the considerably smaller size of the online training set for further refinement tuning versus the sizes of the other corpora.

Although a larger corpus would alleviate this problem, the time to generate the hundreds of examples required would negate much of the utility of online training. Nevertheless, as shown in Figure 13, transfer learning yields a better F1-score than simply training a new set of classifiers from scratch especially given the limited size of the corpus. Similarly, we see benefit in using these refined classifiers with the execution time of the tuners.

Compared with the original GA tuner, the Intel-derived

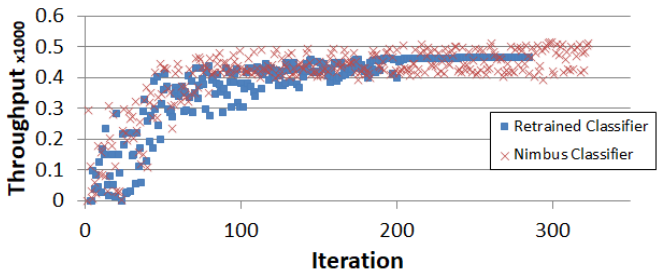

Fig. 14: STT Evolution with Nimbus and Retrained Classifiers.

classifier achieves an improvement of more than 1.02x in execution time while the refined ARM-derived classifier achieves an improvement of more than 1.08x. Unfortunately, the online-tuned classifier fails to improve upon the original GA tuner, yielding a 1.03x slowdown in execution time.

However, execution time is not the entire story. Indeed, examining the evolution of analysis for both the original trained classifier and the retrained classifier yields a surprising result as show in Figure 14. The evolution oscillates between two local maxima for the former before ultimately terminating. In contrast, the retrained classifier has a much smoother evolution before achieving a higher throughput. Thus, the quality of the analysis for the retrained classifier is better than the Nimbus classifier despite the slower run time.

VI. Related Work

Much of the research in this area originates from work done for auto-tuning the parameters of Apache Hadoop, MapReduce and streaming processing frameworks. Evolutionary algorithms such as genetic algorithms, hill climbers, simulated annealing and particle swarms prove themselves useful in this endeavor [14]. An alternative is Gaussian optimization which minimizes the objective function comprised of each parameter represented as a blackbox function [11]. Moreover, machine learning provides a plethora of solutions ranging from supervised learning like support vector machines, multiple linear regressions and random forests to unsupervised learning like K-Means [8], [13]. Finally, network queuing theory can yield optimal configurations given enough information about a topology's current and ideal flows at each subcomponent within it. With such a diverse base of research from which to draw, operators have many options for tuning Storm and other similar stream processing frameworks.

Of note are several techniques which are similar to the techniques utilized in this paper. Van der Veen et al. [8] describe using JMX information to detect overallocation and underallocation of cluster resources and dynamically scale the cluster as needed. Bayesian optimization auto-tuning of Storm is the subject of a paper by Jamshidi and Casale [11] and another by Fisher, Gao and Bernstein [3] who both see significant increases in throughput in Storm topologies as a result of their auto-tuning. However, both make no use of JMX profiling information but rather just raw performance numbers in their fitness functions. Instead, both focused on optimizing their respective Bayesian optimization framework's internal search techniques. Jamshidi and Casale [11] build a customized Bayesian optimization system using the gpml Bayesian optimization library as a base and then incorporate optimizations like Latin Hypercube Design for bootstrapping, Lower Confidence Bound for selecting the next configuration and multi-started quasi-Newton hill-climbers for maximizing marginal likelihood. Furthermore, when comparing their system to other search techniques like simulated annealing and genetic algorithms, the authors do not perform any optimizations like they did for Bayesian optimization, resulting in better overall performance of their system versus the other search techniques. Conversely, Fisher, Gao and Bernstein [3] use an older version of Spearmint which they then customize to incorporate pre-processed topological structural information and to quickly halt fruitless searches altogether rather than simply discard or alter poor configurations. As a result, both previous works make use of heavy optimizations to Bayesian optimization libraries to attain their performance numbers in contrast to the relatively stock usage of Spearmint presented in this paper.

Although not explored herein, network queuing optimization theory remains an alternative approach to the aforementioned optimization techniques. Like us, Beard and Chamberlain [18] first measure the throughput at the edges of network, though they later assume that their topology follows a Jackson network flow for their model which is subject to a variety of constraints like network does not throttle flow, resource contention not occurring and that the network is always in equilibrium. We do make any of these assumptions about the topology and instead utilize the information gleaned from our JMX and Nimbus monitors which indicate when such events occur. De Matteis and Mencagli [19] similarly use queuing theory for their optimizations of SpinStreams, a FastFlow-derived distributed processing framework bearing many similarities to Storm, and a singular target application similar to the STT topology we described earlier. However, rather than focus on optimizing long term throughput as we do, they instead focus on optimizing for quality of service, latency, stability and resource utilization in response to short-term data-stream flows by minimizing a receding horizon Kingman's formula-derived cost model taking these factors into account. Gulisano et al. [20] take a sliding window application that consists of many joins of data streams and similarly focus on optimizing for latency by building a latency model taking into account the periodicity of the inputs, the preceding latencies of the pipeline and the latency cost to perform the windowing operation itself. To optimize for throughput instead, Mencagli et al. [21] apply operator fusion and fission to statically optimize topologies in another instance of SpinStreams based on Akka instead of FastFlow. However, their optimizations rely upon knowing vital information about the topology beforehand like the ideal throughput of a given component in the graph, its probability of outputting to a downstream component and that the mechanism of backpressure is to block the sender until the recipient has capacity to process the sender's output.

We do not make these assumptions nor have this information readily at hand. Instead, we optimize instead using a gray box method.

VII. Conclusion

In this paper, we explore using a genetic algorithm-based approach to automatically tune Storm and using classifiers to further refine its search of the configuration space. With our training topologies, the stock GA is able to improve upon the standard "rules of thumb" tuning strategy's throughput by an average of 4.6x, with between 1.1x and 2.2x improvement over Spearmint, a state of the art Bayesian optimization-based tuner. Employing classifiers to optimize the search process reduces the execution time of our GA tuner by over 40x in certain cases. We further demonstrate that the tuner is portable across clusters of different CPU, memory and even architectures while still being capable of vastly improving the performance of topologies executing in Storm compared with the standard approach to tuning.

Acknowledgments: This work was supported in part by NSF grant CNS-1253575.

References

[1] 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 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[2] Apache Software Foundation. (2018) Apache storm. [Online]. Available: http://storm.apache.org/

[3] L. Fischer, S. Gao, and A. Bernstein, "Machines tuning machines: Configuring distributed stream processors with bayesian optimization," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, 2015.

[4] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: Tapping the performance potential of systems via automatic configuration tuning," *CoRR*, vol. abs/1710.03439, 2017.

[5] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.

[6] S. T. Allen, M. Jankowski, and P. Pathirana, *Storm Applied: Strategies for real-time event processing*. Manning Publications Co., 2015.

[7] P. Córdova, "Analysis of real time stream processing systems considering latency," *University of Toronto patricio@cs.toronto.edu*, 2015.

[8] J. S. van der Veen, B. van der Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer, "Dynamically scaling apache storm for the analysis of streaming data," in *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, 2015.

[9] M. Bilal and M. Canini, "Towards automatic parameter tuning of stream processing systems," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17, 2017.

[10] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[11] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, 2016.

[12] M. Trotter, G. Liu, and T. Wood, "Into the storm: Descrying optimal configurations using genetic algorithms and bayesian optimization," in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2017.

[13] D. Cheng, J. Rao, Y. Guo, and X. Zhou, "Improving mapreduce performance in heterogeneous environments with adaptive task tuning," in *Proceedings of the 15th International Middleware Conference*, 2014.

[14] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based auto-tuning of mapreduce," in *European Conference on Parallel Processing*, 2013.

[15] F. Wilhelmstötter. (2016) Jenetics library user manual. [Online]. Available: http://jenetics.io/manual/manual-3.7.0.pdf

[16] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[17] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[18] J. C. Beard and R. D. Chamberlain, "Analysis of a simple approach to modeling performance for streaming data applications," in *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2013, pp. 345–349.

[19] T. De Matteis and G. Mencagli, "Elastic scaling for distributed latency-sensitive data stream operators," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 2017, pp. 61–68.

[20] V. Gulisano, A. V. Papadopoulos, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "Performance modeling of stream joins," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM, 2017, pp. 191–202.

[21] G. Mencagli, P. Dazzi, and N. Tonci, "Spinstreams: a static optimization tool for data stream processing applications," in *Proceedings of the 19th International Middleware Conference*. ACM, 2018, pp. 66–79.