# GML: Efficiently Auto-tuning Flink's Configurations via Guided Machine Learning

Yijin Guo, Huasong Shan, *Member, IEEE,* Shixin Huang, Kai Hwang, *Fellow, IEEE,* Jianping Fan, *Member, IEEE,* and Zhibin Yu, *Member, IEEE*

**Abstract**—The increasingly popular fused batch-streaming big data framework, Apache Flink, has many performance-critical as well as untamed configuration parameters. However, how to tune them for optimal performance has not yet been explored. Machine learning (ML) has been chosen to tune the configurations for other big data frameworks (e.g., Apache Spark), showing significant performance improvements. However, it needs a long time to collect a large amount of training data by nature. In this paper, we propose a guided machine learning (GML) approach to tune the configurations of Flink with significantly shorter time for collecting training data compared to traditional ML approaches. GML innovates two techniques. First, it leverages generative adversarial networks (GANs) to generate a part of training data, reducing the time needed for training data collection. Second, GML guides a ML algorithm to select configurations that the corresponding performance is higher than the average performance of random configurations. We evaluate GML on a lab cluster with 4 servers and a real production cluster in an internet company. The results show that GML significantly outperforms the state-of-the-art, DAC (Datasize-Aware-Configuration) [1] for tuning the configurations of Spark, with $2.4\times$ of reduced data collection time but with $30\%$ reduced 99th percentile latency. When GML is used in the internet company, it reduces the latency by up to $57.8\times$, compared to the configurations made by the company.

**Index Terms**—Big Data Systems, Batch-Stream Fused Processing, Flink, Configuration Optimization, Generative Adversarial Networks (GAN)

## 1 Introduction

Apache Flink is a distributed processing framework for stateful computations over *unbounded* and *bounded* data streams. It has been designed to run in all common cluster environments such as YARN [2] and perform computations at *in-memory* speed and at *any scale* [3]. Processing of bounded data streams is also known as batch processing whereas processing of unbounded data streams is called streaming processing as well. Previously, batch and streaming processing can only be performed on separated frameworks (e.g., Apache Hadoop [4] and Spark [5] for batch processing and Storm [6] for streaming processing). In contrast, Flink fuses batch and streaming processing in a single framework, significantly easying the maintenance of different kinds of big data analysis. Consequently, Flink is getting increasingly popular.

Most internet giants such as Alibaba, eBay, Netflix, and Uber are using Flink to build enterprise-level real-time services [7]. 85% of Flink applications were business intelligence, anomaly detection [8], and real-time recommendations [9] by the end of 2017, reported by a survey from Data Artisan [10]. These enterprise-level applications have two features. First, an application repeatedly runs on a given cluster for a long time but with different data. Second, all the applications require Flink to provide high performance. In other words, the latency of Flink services needs to be short whereas their throughput needs to be high.

The performance of an enterprise-level Flink program can be expressed by:

$$perf = f(ps, pg, cf, sp) \tag{1}$$

with $perf$ the latency of a Flink service or the throughput of a Flink system, $ps$ the physical servers and other hardware, $pg$ the Flink program, $cf$ the configurations of the Flink program, and $sp$ the data speed. For a `given` Flink program running on a `given` cluster, $ps$ and $pg$ are fixed. The only way we can tune for high performance is to tune $cf$ and $sp$.

For flexibility, Flink has more than 300 configuration parameters ($cf$) [11] and some of them are performance-critical. For example, the Flink configuration parameter $taskmanager.memory.fraction$ specifies the relative amount of memory that the task manager reserves for sorting, hash tables, and caching of intermediate results. Obviously, the value of this parameter significantly affects the performance of a Flink program.

However, similar to Apache Hadoop and Spark, the number of performance-critical configuration parameters of Flink is large and the parameters may interact with each other in a complex way. This makes simple analytical models inaccurate for tuning the configuration parameters, let alone a manual approach. Machine learning based models have therefore been employed to help tune the configuration parameters for Hadoop and Spark [1], [12], [13], [14], [15],

- *Yijin Guo, Shixin Huang, Jianping Fan and Zhibin Yu are with Shenzhen Institute of Advanced Technology (SIAT), Chinese Academy of Science (CAS), Shenzhen, China, 518055. E-mail: yj.guo,huangsx, jp.fan,zb.yu@siat.ac.cn*
- *H. Shan is with JD.com American Technologies Corporation, Mountain View, CA 94043. E-mail: huasong.shan@jd.com.*
- *K. Hwang is with Computer Science and Engineering, Chinese University of Hong Kong (CUHK), Shenzhen, China.*

*Manuscript received May 28, 2020.*
*(The first two authors have equal contribution. Shixin Huang contributes the additional experiments required by the revision. Zhibin Yu is the corresponding author. Kai Hwang polishes the paper and Jianping Fan suggests this research direction and attends all the discussions.)*

[16], [17], [18], and traditional distributed systems [19], [20]. Although these machine learning based approaches achieve significant performance improvements, they need a long time to collect a large amount of training data. For example, the state-of-the-art, DAC for tuning Spark configurations, takes two days to collect training data for each Spark program [1], hindering it from being widely employed.

In this paper, we propose a guided machine learning (GML) approach to efficiently auto-tune the configuration parameters of Flink programs with significantly shorter time for collecting training data compared to traditional ML approaches. GML innovates two techniques. First, it leverages generative adversarial networks (GANs) to generate a part of training data to reduce the time needed for collecting training data from real systems. Second, GML carefully selects configurations from a number of randomly generated ones as the inputs of GANs, which the corresponding performance of the selected configurations is higher than the average performance (e.g., shorter latency or higher throughput) corresponding to the random configurations. As such, GML guides GANs to generate configurations with better performance, reducing the time for searching the configuration for optimal performance.

Note that GML is an offline (static) configuation optimization technique for a *given* Flink program running on a *given* cluster. It is designed for a common industry scenario where a Flink program repeatedly runs for a long time (e.g., months or years) with different but similar streaming rates. If a Flink cluster dynamically changes over time, for example, different servers are added to the cluster, the program running on the cluster changes from time to time, or more than one heterogeneous workloads share the same Flink cluster, we need to re-run GML to find a new optimal configuration for the new situation, which is tedious. It is possible to develop an online configuration optimization technique for Flink systems which can adapt to their dynamic changes. But this needs to modify the source code of the Flink framework substantially. We therefore plan to work on it in the future.

We employ a lab cluster consisting of 4 servers with all the 4 Flink programs in HiBench [21], the streaming benchmark from Yahoo! [22], and a real production cluster from an internet company with its log analysis service to evaluate GML. The results show that GML significantly outperforms the state-of-the-art configuration auto-tuning approach, DAC for tuning the configurations of Spark programs, with much shorter time used for collecting training data and searching the optimal configurations.

In particular, this paper makes the following main contributions:

- We propose an approach to optimize the configurations of Apache Flink with respect to performance. To our best knowledge, we are the first to study on the configuration optimization of Flink programs.
- We propose to leverage GANs to generate a part of training data for machine learning models used for configuration optimization and in turn to reduce the time needed for collecting the training data.
- We guide GAN to select configurations with higher performance than the average performance of randomly generated configurations as inputs.
- we demonstrate that GML only takes 3 hours to reduce the 99th percentile latency by $2.2\times$ on average and up to $3.7\times$ compared to the default configurations. For the batch jobs, if they are configured by our GML approach, they are $1.5\times$ and $2.9\times$ faster than they are configured by the DAC approach and an expert approach, respectively. More importantly, GML reduces the configuration optimization time by $2.4\times$ compared to DAC.
- We show that GML improves the latency of a Flink program used for log analysis in a real production environment by $10.5\times$ on average and up to $57.8\times$.

The rest of this paper is organized as follows. Section 2 describes the background and motivation. Section 3 depicts our guided machine learning approach. Section 4 describes our experimental methodology. Section 5 presents the experimental results and analysis. Section 6 describes the related work and Section 7 concludes the paper.

## 2 Background and Motivation

In this section, we describe the background and motivation.

### 2.1 Apache Flink

Apache Flink [23] is an open source framework for distributed data processing which embraces the Google dataflow model [24]. It enables users to write programs that can be distributed over a number of servers (workers), which makes it possible to process large-scale datasets faster than a single computer. Unlike other big data frameworks such as Hadoop and Storm, Flink provides a uniform architecture for processing both batch and stream data by treating them both as streams. Note that although Spark has streaming processing paradigm as well, it uses micro-batch to simulate streaming processing, which significant increases the latency of steaming services.

Figure 1 illustrates an overview of the Flink architecture. As can be seen, there is a daemon called *Job Manager* running on the master node of a Flink cluster. In each worker node, there is another daemon named *Task Manager* that manages the running of all tasks on the node. Each Task Manager has a memory and I/O manager, a network manager, and multiple task slots, which are used to manage the memory, network, and CPU resources, respectively. Moreover, the Actor Systems serve the communications between the Job Manager and the Task Managers, between a Flink program and the Job Manager, and between multiple Task Managers. If one wants to run a Flink program (job), he/she must firstly submit it to the Job Manager. The Job Manager then decomposes the job into a number of tasks and schedules them to the Task Managers to run.

Internally, Flink represents a job by using directed acyclic graphs (DAGs) [25]. The nodes of a DAG act as either *sources*, *sinks*, or *operators*. Source nodes read in or generate input data whereas sink nodes produce outputs. The inner vertices of a DAG are operators which execute arbitrary *user-defined functions* (UDFs) that consume input from incident nodes and provide input for adjacent nodes. The DAGs need to be transformed into more concrete execution graphs

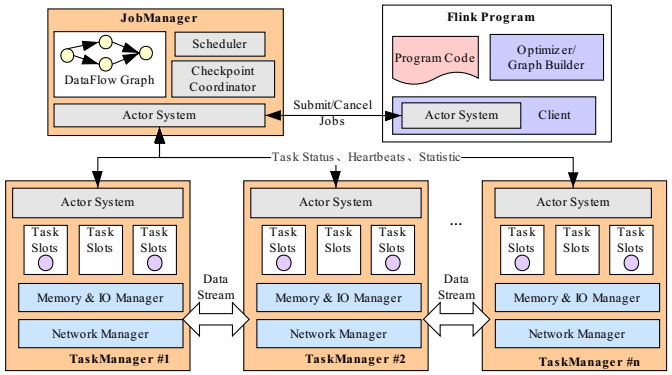



Fig. 1. Apache Flink Architecture Overview

which contain necessary information for running a job on a cluster. During the transformation, an input dataset is generally partitioned into multiple small datasets and each dataset is processed by a subtask. Note that all the subtasks execute the same UDF in parallel but with different data.

For flexibility such as running on different cluster managers including Mesos [26], YARN [2], and Kubernetes [27], Apache Flink provides more than 300 configuration parameters which specify 33 aspects such as HDFS, core, Job Manager, and Task Manager [28]. Some configuration parameters, for example, *jobmanager.archive.fs.dir* (specifying the directory where the archive file system of the Job Manager locates in a Flink cluster), **do not** affect the performance of a Flink program significantly. However, other configuration parameters such as *taskmanager.memory.fraction* and *akka.framesize* **do**. *taskmanager.memory.fraction* specifies the relative amount of memory that the task manager reserves for sorting, hash tables, and caching of intermediate results. Since different programs may need different size of memory for caching intermediate results, a too small value of *taskmanager.memory.fraction* results in a too small memory buffer for caching the intermediate results and in turn significantly reduces the performance. The later (*akka.framesize*) specifies the maximum size of messages which are sent between the Job Manager and the Task Managers. Obviously, this size significantly affects the network performance. We call these *performance-critical* configuration parameters and focus on tuning them for high performance.

### 2.2 Generative Adversarial Networks

Generative Adversarial Networks (GANs) has become the most prominent examples of a new class of ML algorithms which generative models and discriminative models are simultaneously generated in a competitive setting [29]. The objective of discriminative models is to correctly label samples from either the generative models or the training data. In contrast, the objective of generative models is to deceive the discriminative models. In other words, the generative models produce samples that are categorized as training data by the discriminative models. As such, generative models can be thought of as a team of counterfeiters while the discriminative model can be treated as a police trying to detect the counterfeit currency. This competition drives both teams to improve their methods until the counterfeits are indistinguishable from the genuine articles [30].

### 2.3 Motivation

Apache Flink is a new and unique big data processing framework which fuses the batch and streaming processing in one platform. Like other big data frameworks such as Apache Haddop and Spark, Flink has a larger number ($> 200$) of configuration parameters and many of them are performance-critical. Although a number of researches have been done for optimizing the configurations of other big data frameworks such as Hadoop [12], [31], [32], [33] and Spark [1], optimizing the performance of Flink programs by tuning the configuration parameters has not been explored, which is the first motivation of this study.

Moreover, the configuration optimizing researches [1], [12], [31], [32], [33] for Hadoop and Spark show that machine learning based approaches [1], [12] are the most promising solutions compared to analytical model [31], [32] and statistical reasoning based approaches [33]. However, the nature of machine learning based approaches needs a large number of training data, which takes a long time to collect them. For example, no matter for Hadoop or Spark, the machine learning based approaches [1], [12] all take one to two days to collect enough training data from real clusters to train accurate enough performance models. Although this can be accepted by users who repeatedly run their big data programs for a long time, they still ask for approaches with short time for data collection. We therefore ask a question: Is it possible to reduce the time for collecting training data but still achieve the same optimization results for the machine learning based approaches? This question is the second motivation of this work.

## 3 Guided Machine Learning Approach

In this section, we describe our guided machine learning (GML) approach to optimize the configurations of Flink programs with respect to performance.

### 3.1 Overview

We propose an approach named GML which is designed to optimize the performance of a Flink program by tuning the performance-critical configuration parameters. The goal of GML is to tune the configurations of a Flink program by using ML techniques but with much shorter time to collect the training data compared to traditional ML algorithms. The key idea is to leverage GANs to reduce the time for training data collection. As Section 2.2 described, the generative models of GANs finally generate data which is indistinguishable from the genuine data. We therefore leverage the generative models of GANs to generate a part of training data. During this process, we design a technique called *mean processing* to guide GANs to use configurations with better performance than the mean performance of a set of random configurations. As such, GANs can capture the distributions of configurations with high performance.

Figure 2 shows an overview of our GML approach. It consists of four components: *generating*, *profiling*, *modeling*, and *searching*. The generating component generates configurations which are used to study the relationship between configurations and the performance of a Flink program. The profiling component is used to collect the performance (e.g.,

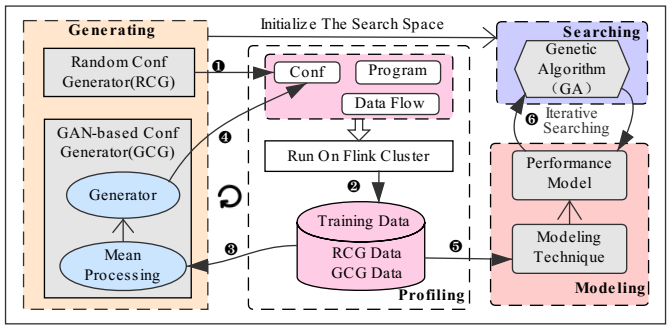



Fig. 2. An Overview of our Guided Machine Learning (GML) Approach.

latency or throughput) of a Flink program with a number of different configurations. The modeling component leverages a suitable machine learning algorithm to build a performance model which takes the configuration parameters as inputs and the performance (latency or throughput) as output. The searching component is used to search the configuration for a Flink program which can achieve the optimal performance.

The relationship between the components is as follows. In step ①, the random configuration generator (RCG) randomly generates $k$ configurations (each configuration is a vector of the configuration parameter values, see Section 3.2). The profiling component subsequently runs a Flink program with the $k$ configurations on a real Flink cluster and collects the performance (e.g., latency or throughput) of the program with each configuration, as step ② shows. Thirdly, GML selectively inputs the configurations which their corresponding performance is higher than the mean performance of the $k$ configurations to the GAN-based configuration generator (GCG), as shown in step ③. This is what we call *mean processing*. Note that it is extremely important to determine a suitable $k$ yet notoriously challenging. Larger $k$ increases the time needed for profiling whereas smaller $k$ may make that the randomly generated configurations failed to represent the real case. We determine the suitable $k$ in Section 5.1.

Next, GCG generates $p$ configurations which capture the distribution of the ones with high performance from the $k$ randomly generated configurations, as the step ④ shows. Subsequently, the profiling component collects the performance for these $p$ configurations. Note that the step ②,③, and ④ may repeat a number of times, leading to configurations with higher and higher performance. After we get enough profiling data, the modeling component uses them as training data to train performance models by using machining learning algorithms, as shown in step ⑤. Finally, the performance model is used by the searching component to search the configuration for a Flink program which can achieve the optimal performance, as the step ⑥ shows. Note that ⑤ and ⑥ may also repeat a number of times, aiming to find the optimal performance.

One may think that it is unnecessary to build performance models because we can know the performance of a program with a certain configuration by running the program with that configuration when we search the optimal configuration for the program. However, depending on a Flink program, running it with a certain configuration on a real cluster may take several to tens of seconds. Moreover, searching the optimal configuration for a program typically needs a large number (e.g., 10,000) of real program executions in this case, resulting in a very long time for finding the optimal configuration for the program. If we have a performance model, knowing the performance for a Flink program with a certain configuration only takes several milliseconds, which is much faster than running the real program. The only problem is how to build a performance model with high accuracy, which is the focus of this study.

### 3.2 Generating Configurations

As shown in Figure 2, the generating component of GML consists of two configuration generators: random configuration generator (RCG) and GANs-based configuration generator (GCG). RCG randomly generates a value for each configuration parameter in its corresponding value range while GCG tries to generate a value for each configuration parameter to mimic the value distribution of the GANs' input configurations. When we run a Flink program, we need to set the value of each parameter, which can be done by either RCG or GCG.

We use the following vector to represent a configuration:

$$conf_i = \{c_{i1}, c_{i2}, ..., c_{ij}, ..., c_{in}\}, i = 1, ..., m \quad (2)$$

with $conf_i$ the $i^{th}$ configuration and $c_{ij}$ the value of the $j^{th}$ configuration parameter in the $i^{th}$ configuration; $n$ is the number of configuration parameters of Flink that can be easily adjusted and significantly affect performance. In this study, we consider 27 performance-critical configuration parameters (shown in Table 2). $m$ is the number of different configurations. Note that as long as two values of at least one parameter of two configurations are different, we treat them as two different configurations.

To observe how the performance-critical configuration parameter affect the performance of a Flink program, we execute each Flink program a number of times and each execution with a different configuration. Therefore, $m$ configurations correspond to $m$ executions.

### 3.3 Profiling Performance

For each configuration generated by the generating component, we execute a Flink program with the configuration in a Flink cluster and collect the 99th percentile latency and throughput. Then we construct the following execution vector for each Flink program as a profiling data item:

$$pv_i = \{perf_i, conf_i, dspeed_i\}, i = 1, ..., m \quad (3)$$

with $pv_i$ the vector of the $i^{th}$ execution, $perf_i$ the 99th percentile latency or the throughput of the $i^{th}$ execution, $conf_i$ the configuration for the $i^{th}$ execution, and $dspeed_i$ (in bytes/s) the input data speed for the $i^{th}$ execution.

Now each vector $pv_i$ contains a specific configuration and its corresponding performance. $m$ such vectors for a Flink program form a matrix which is used as the training data set in the ML-based configuration auto-tuning techniques. Generally, $m$ must be large enough for building accurate performance models (e.g., model error $< 10\%$)

and different programs may need different values of $m$ for accurate performance models. Since one execution of a program can only obtain one vector $pv_i$, larger $m$ causes more executions of the program and in turn longer profiling time. The goal of this study is to reduce the amount of the profiling data to decrease the time needed for training data collection but keep the same model accuracy.

### 3.4 Modeling Performance

To efficiently tune the configurations for Flink programs, we need to build performance models as described in Section 3.1. We can build a performance model for a single Flink program or a group of Flink programs but with more challenges. In this study, we build the model for a single program as follows.

$$perf = f(c_1, c_2, ..., c_i, ..., c_{27}, dspeed) \quad (4)$$

with $perf$ the performance, $c_i$ the value of the $i^{th}$ configuration parameter, and $dspeed$ the input data speed (bytes/s).

Since the number of input parameters is 28 which is large for modeling, it is very difficult to build accurate models by using analytical and statistic reasoning based approaches [1]. We therefore use ML algorithms to construct performance models for Flink programs. However, there are so many different ML algorithms. Which one is a good fit in our case? It is challenging to answer this question because there are no theory guidelines.

We therefore explore the ML algorithms for Flink configuration auto-tuning by experiments. The goal is to identify an algorithm that can build accurate performance models but with an as small as possible amount of training data. The challenge is that the amount of training data needs to be balanced: a large number of training data increases profiling and training time, whereas a small amount of training data may not enable accurate performance models. In particular, long profiling and training time significantly limits the performance improvement of a Flink program.

We tried five different standard ML algorithms: Supported Vector Machine (SVM), Artificial Neural Network (ANN), Gaussian Process (GP), Random Forests (RF), and Stochastic Gradient Boosted Regression Tree (SGBRT). We also tried one ML algorithm named hierarchical modeling (HM) customized for Apache Spark configuration optimization by the DAC paper [1]. We apply the total six algorithms on the same set of training data to construct six performance models and compare their accuracy which is calculated by:

$$err = (\sum_{i=1}^{n} \frac{|pre_i - act_i|}{act_i})/n \times 100\%$$

with $pre_i$ the performance predicted by the models for a given Flink application, $act_i$ the actual performance of that application, and *n* the total number of testing set. Note that *n* is a quarter of the training set and there is *no overlap* between the training set and the testing set.

Figure 3 shows the errors of the models built by the six algorithms. The *T60* along the X-axis means the training set consists of 60 vectors and the same applies to *T120*, *T180*, and so on. As can be seen, the accuracies of the models generally improve when the amount of training data increases. When the training set size achieves 300, the errors of all models are less than 10%. In particular, the errors of the models built by SGBRT and HM can be less than 10% when the training set size is 240 while others can not. This indicates that models built by SGBRT and HM can achieve high accuracy with relatively less amount of training data compared to the other four ML algorithms. However, the errors of the models built by HM are all higher than those built by SGBRT, as shown in Figure 3. We therefore employ SGBRT in our Flink configuration auto-tuning study.

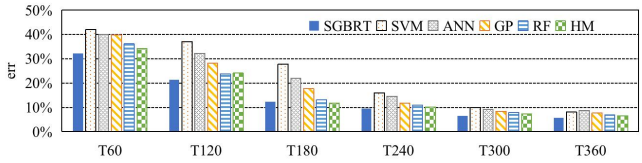



Fig. 3. The errors of the performance models constructed by Stochastic gradient boosted regression tree (SGBRT), Supported Vector Machine (SVM), Artificial Neural Network (ANN), Gaussian Process (GP), Random Forests (RF), and Hierarchical Modeling (HM) with different sizes of training sets. T60 means training set consists of 60 vectors defined by equation (3).

### 3.5 Searching Optimal Configuration

By far, we still do not know the optimal configuration parameter values for a given Flink application. To address this issue, we need an automatic searching approach. There are many algorithms that can be used to search a complex solution space such as recursive random search (RSS) [34], tabu search (TAS) [35], and Genetic Algorithms (GA [36]. RSS can easily stuck in local optima and TAS typically suffers from the local convergence rate and the initial solution space. GA is an evolutionary algorithm inspired by evolutionary biology such as inheritance, mutation, selection, and crossover, which is robust against local optima. Our goal is to find a Flink program-input pair from the global space of parameters, which is a complex space to explore with many local optima. We therefore choose GA in this study.

Figure 4 shows the workflow of our *searching* component. In step ①, we input a set of initial configurations to the performance model of a Flink program, and the performance model outputs a *perf* which can be either 99th percentile latency or throughput. Note that initial configurations consist of two parts: 50% of the configurations are generated by GANs and the other 50% are generated randomly, which is significantly different from previous configuration auto-tuning approaches where the initial configurations for the GA are all randomly generated. This optimizes the quality of the initial population (configurations) of the GA, guiding it to speed up the search for optimum configurations. The reason is that if all the initial configurations are randomly generated, the GA starts to search optimal configuration in a random direction, leading to a long time to find the optimal configuration. In contrast, if some initial configurations are randomly generated and others are generated by GANs with corresponding high performance, the search direction of the GA would be quickly guided to a direction where the configurations correspond to high performance, leading to a short time to identify the optimal configuration. We verify this assertion in Section 5.3.

In addition, since our performance model is built by a ML algorithm — SGBRT, we can quantify the importance of

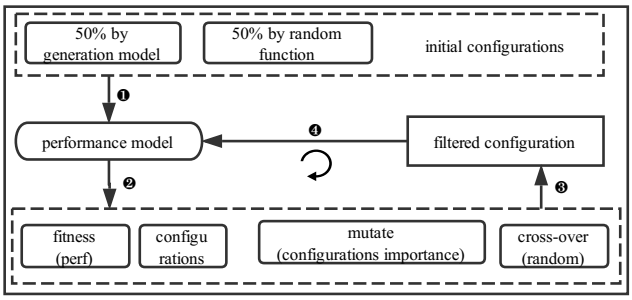



Fig. 4. Searching the optimum configuration.

the configuration parameters with respect to performance when building the model. Subsequently, we leverage the configuration parameter importance to guide the *mutation* operation of the GA. Concretely, we use the importance of the configuration parameters as the mutation probability and guides the GA to make mutation only occur on the important configuration parameters. In other words, this makes GA avoid performing the mutation operations on parameters that do not influence performance significantly, shortening the time used to find the optimum configuration parameter values. We will show the important configuration parameters of the experimented programs in Section 5.4.

In step ②, we pass configuration parameter values, the corresponding performance predicted by performance model, and the importance of the configuration parameters to the GA. Subsequently, the GA performs a number of operations such as *crossover* and *mutate* on the configuration parameter values, as shown in step ③. In step ④, these configuration parameter values obtained by the GA are passed to the performance model as inputs to get a predicted performance. Next, the new configuration parameter values and the corresponding performance *perf* are passed to the GA again. The step ② to ④ might be repeated for a number of times until the optimum configuration is found. In other words, if we find the performance produced by three configurations continuously generated by the GA can not be higher anymore, we stop the searching of GA, and treat one of the configurations as the optimal configuration.

## 4 Experimental Setup

In this section, we describe our experimental methodology.

### 4.1 Lab Cluster Platform

Our lab experimental cluster platform consists of 4 Linux servers. One serves as the master node which runs the Job Manager and the other three serve as slave nodes which execute the Task Managers. Each server is equipped with an Intel(R) Xeon(R) CPU E5-2407 2.20GHz 4-core processor and 32 GB DRAM. The OS of each node is SUSE Linux Enterprise Server 11. As for the Flink framework, we use Flink 1.4.2 which is a stable version.

To generate data streams, we employ a Kafka cluster [37]. Figure 5 shows the workflow of a typical Kafka and Flink cluster. Kafka firstly converts data stream from transactions, log, IoT (Internet of Things), and etc. into Kafka stream. The Flink program subsequently processes the Kafka stream

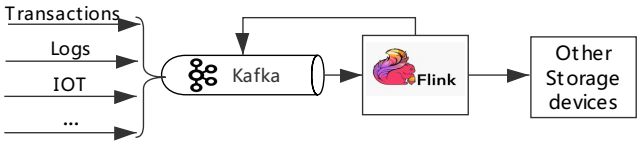



Fig. 5. The coordination between a Kafka cluster and a Flink cluster.

TABLE 1
Experimented applications in this study.

| Benchmark suite | Application | Abbr. |
|---|---|---|
| Hibench | FixWindow | FW |
| | WordCount | WC |
| | Repartition | RP |
| | Identity | ID |
| Yahoo Streaming benchmark | advertisement | AD |

and outputs the results to the Kafka cluster, or other storage devices, or other applications. In our experiments, our Kafka cluster consists of 4 servers and the Flink cluster has the same number of servers.

### 4.2 Benchmarks

We select all the programs from the Flink version of HiBench and a program from Yahoo! — the *advertisement* program [38] as our experimental programs, shown in Table 1. The Yahoo! Streaming Benchmark is widely used in industry to evaluate streaming systems. It is a simple advertisement application which attempts to probe some common operations performed on data streams like window operations and checkpoints.

The Flink version of HiBench contains four programs: *FixWindow*, *WordCount*, *Repartition*, and *Identity*. FixWindow is a window-based aggregation which evaluates the window operations in the stream frameworks. WordCount tests the performance of the stateful operators and the cost of checkpoints/Ackers. Repartition evaluates the shuffle efficiency and Identity tests the read/write efficiency of an external input source represented by a kafka cluster.

### 4.3 Real Production Cluster

We also evaluate GML in a real production cluster from a large retailer in China. In this experiment, we use two docker containers as Job Managers and 10 docker containers as Task Managers. The docker containers (virtual machines) are managed by a customized version of Kubernetes [27]. Each container is equipped with 4 CPU cores (Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz), 8 $GB$ memory, and 10 $GB/s$ ethernet network. All the containers share a network disk which is 100 $GB$. The OS is Centos 7 with Linux kernel 3.10.0-327.28.3.el7.x86_64 and the Apache Flink version is 1.4.

In this experiment, we run a type of real-time log analysis program written in Apache Flink. Its main function is to process the log data, to extract the execution time of each request from the log, to aggregate execute time from the raw data, and to calculate the percentile response time of each type of requests. We require the program can get the analysis results within 5 seconds since the program is a key component of the monitoring and alarm system. In addition,

TABLE 2
The 27 Flink configuration parameters. (jm - jobmanager, tm - taskmanager.)

| Configuration Parameters-Description | Abbr. | Range | Defaut |
|---|---|---|---|
| **parallelism.default**-The default parallelism for programs. | PD | 1-24 | 1 |
| **tm.numberOfTaskSlots** - The number of parallel tm operators. | slot | 1-8 | 1 |
| **jm.heap.mb** - JVM heap size (MB) for jm. | JH | 1024-4096 | 1024 |
| **tm.heap.mb** - JVM heap size (MB) for tm. | TH | 1024-10240 | 1024 |
| **tm.memory.fraction** - The relative amount of tm reserved memory. | MEMF | 0.5-0.8 | 0.7 |
| **tm.memory.segment-size** - The buffer size for mem & network stack(B). | SEGS | 32KB - 4M | 32KB |
| **tm.runtime.hashjoin-bloom-filters** - en/disable filters. | RTHS | false, true | false |
| **tm.runtime.sort-spilling-threshold** - spilling threshold. | RTTH | 0.6 - 0.9 | 0.8 |
| **tm.runtime.max-fan** - The max fan-in for external merge joins and fan-out for spilling hash tables. | RTMF | 120 - 200 | 128 |
| **tm.network.memory.fraction** - JVM mem fraction for network buffers. | NTMF | 0.1 - 0.6 | 0.1 |
| **tm.net.num-arenas** - the number of Netty arenas. | NTNA | 1 - 8 | #slot |
| **tm.net.server.numThreads** - The number of Netty server thread. | NTST | 1 - 8 | #slot |
| **tm.net.client.numThreads** - The number of Netty client threads. | NTCT | 1 - 8 | #slot |
| **blob.fetch.num-concurrent** - The number concurrent BLOB fetches (e.g., JAR file downloads) that the jm serves. | BONC | 40 - 100 | 50 |
| **blob.fetch.retries** - The number of retries for the tm to download BLOBs (such as JAR files) from the jm. | BOFR | 40 - 100 | 50 |
| **akka.framesize** - Max size of messages sent between the jm and tm. | AKFS | 6M - 20M | 10M |
| **akka.watch.threshold** - Threshold for DeathWatch failure detector. | AKWT | 4 - 22 | 12 |
| **tm.network.memory.min** - Min memory size for network buffers (B). | NTMI | 64M - 256M | 64M |
| **tm.network.memory.max** - Min memory size for network buffers (B). | NTMA | 1024M - 2048M | 1024M |
| **tm.net.sendReceiveBufferSize** - The Netty send and receive buffer size. | NTBS | 763659- 1527317 | system buffer sizes |
| **blob.fetch.backlog** - The max number of queued BLOB fetches (such as JAR file downloads) that the jm allows. | BOBL | 900 - 2000 | 1000 |
| **jm.tdd.offload.minsize** - Max size of the TaskDeploymentDescriptor's serialized task and job information to transmit them via RPC. | JMTO | 900 - 4096 | 1024 |
| **fs.overwrite-files** - Specifies whether file output writers should overwrite existing files by default. | FSOF | false, true | false |
| **fs.output.always-create-directory** - File writers running with a parallelism larger than one create a directory for the output file path and put the different result files (one per parallel writer task) into that directory. | FSOO | false, true | false |
| **compiler.delimited-informat.max-line-samples** - The max number of line samples taken by the compiler for delimited inputs. | CMAS | 9 - 20 | 10 |
| **compiler.delimited-informat.min-line-samples** - The min number of line samples taken by the compiler for delimited inputs. | CMIS | 2 - 8 | 2 |
| **compiler.delimited-informat.max-sample-len** - The max length of a line sample that the compiler takes for delimited inputs. | CMASL | 1M - 10M | 2M |

we require the monitor system can capture the exceptions within seconds.

### 4.4 Configuration Parameters

As discussed earlier, we choose a wide range of Flink configuration parameters that significantly influence performance, including *memory management*, *execution behavior*, *networking*, *parallelism*, and etc. Table 2 lists the 27 parameters experimented in this study.

The last column of Table 2 provides the default values of the parameters which are recommended by the Flink team and can be found at [11]. The third column shows the value range of each configuration parameter. This information is however not provided by the Flink team, and we therefore conducted experiments to determine the value range for each parameter. Note that the value ranges of these configuration parameters might be different for different clusters because some value ranges depend on cluster hardware configurations such as memory size.

### 4.5 Data Speed Paramemters

The data speed $sp$ for the Flink benchmarks in HiBench can be tuned by four parameters shown in Table 3. We employ *recordsPerSecond* to represent $sp$ and it is calculated as follows.

$$sp = \frac{recordsPerInterval * 1000 * producerNumber}{intervalSpan} \quad (5)$$

In our experiment, we tune $sp$ by changing the value of *recordsPerInterval* while keeping other parameters unchanged. Through this way, we can observe how the data speed affects the performance of a Flink program. Note that the unit of $sp$ represented by Equation ( 5) is $records/s$ and it can be converted to *mega-byte/s* by using $recordLength$.

## 5 Results and Analysis

In this section, we first determine several parameters of GML. We then present the experimental results and analysis.

### 5.1 Determining the Parameter $k$ of GML

As described in Section 3.1, it is important to determine a suitable $k$ which specifies the number of randomly generated configurations. They are used as the basis of the GANs to generate more configurations as training data to train performance models. We conduct the following experiments to determine $k$. First, we run a Flink program 150 times, each with a randomly generated configuration and we call this one experiment. We observe which time the configuration with the highest performance in the experiment (the 150 runs) occurs. Second, we perform the experiment 100 times for each Flink program.

Figure 6 shows the experiment results for benchmark $WC$. Each dot in the figure represents the configuration with the highest performance in one experiment. As can be seen, in 87% of the experiments, the configuration with the

TABLE 3
Parameters used to tune the data speed of a Flink program.

| Parameters | Short Name | value | Description |
|---|---|---|---|
| hibench.streambench.datagen.recordsPerInterval | recordsPerInterval | 5 | number of records to generate per interval span (default:5) |
| hibench.streambench.datagen.producerNumber | producerNumber | 3 | Number of KafkaProducer running on different thread (default:1) |
| hibench.streambench.datagen.intervalSpan | intervalSpan | 30 | Interval span in millisecond (default: 50) |
| hibench.streambench.datagen.recordLength | recordLength | 200 | fixed length of record (default: 200 bytes) |

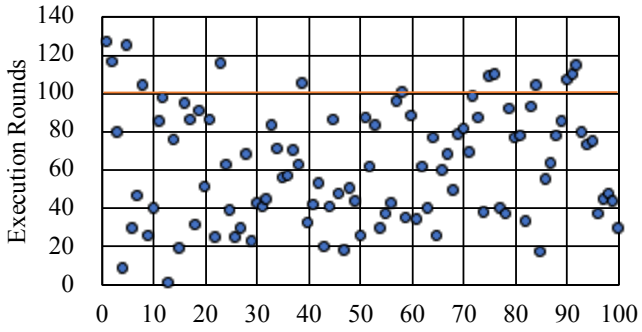



Fig. 6. Observing which time the configuration with the highest performance occurs in one experiment. We call 150 executions of a Flink program with a different randomly generated configuration each as one experiment. The X-axis means the $i^{th}$ experiment. The Y-axis represents the order of a Flink program's execution with a different configuration.

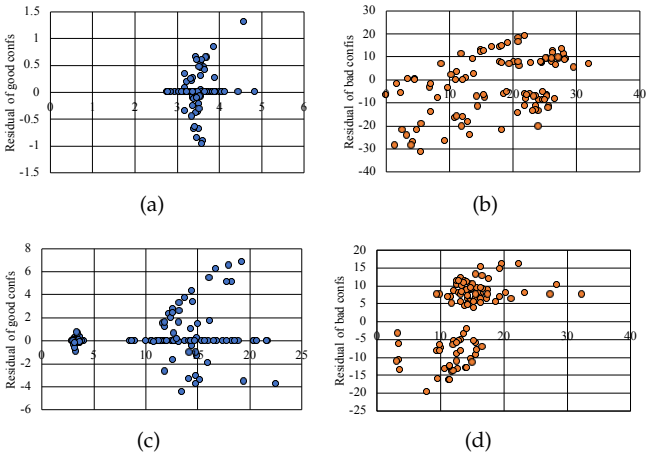



Fig. 7. The residual differences between guided configurations and random configurations for programs $ID$ and $WC$.(a) The program $ID$ with guided configurations. (b) The program $ID$ with random configurations.(c) The program $WC$ with guided configurations. (d) The program $WC$ with random configurations.

highest performance is generated before the $100^{th}$ round. This indicates that it is highly possible to generate configurations with high performance in 100 randomly generated configurations. We observe the similar results for other experimented programs and we therefore set $k$ to 100 for our GML approach.

### 5.2 Accuracy of the Performance Models

As aforementioned, we design *mean processing* to select configurations from 100 randomly generated configurations for GANs to generate more configurations. We put the 100 randomly generated configurations and the ones generated by GANs together to train performance models. Before

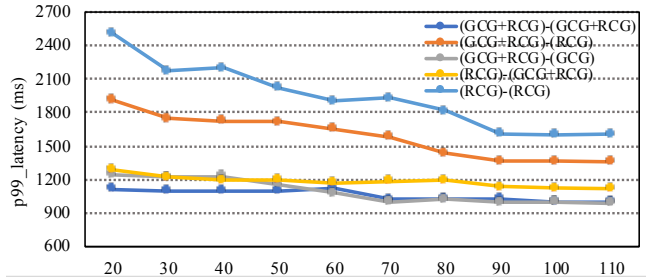



Fig. 8. The 99th percentile latency variation with the iteration numbers of the GA and different mixtures of configurations generated by RCG and GCG. For the (XX-YY) style legend, XX represents how the training data set used to train a performance model is generated and YY denotes how the initial data set of the GA is generated. For example, (GCG+RCG)-(GCG+RCG) means that a part of the training data is generated by GANs (GCG) and the others are generated randomly (RCG), and a part of the initial configurations of the GA is generated by GANs (GCG) and the others are generated randomly (RCG).

we evaluate the model accuracy, we define two concepts: *guided configuration* ( a configuration generated by GAN) and *random configuration*. To evaluate the accuracy of the performance model with different types of configurations, we collect 100 guided configurations and 100 random configurations, which are different from the training data. We input these 200 configurations to the performance model to predict the performance of five programs. We leverage residual plot to observe the accuracy of the performance models. That is, the accuracy is represented by the residual difference between the performance predicted by the performance model and that obtained by real measurement.

Figure 7 shows plots produced by 100 real performance measurements and 100 performance predictions by GML for programs $ID$ and $WC$ for 100 guided configurations and 100 random configurations. The X-axis represents the predicted performance values, and the Y-axis denotes the residuals. These figures clearly show that the models are accurate across the guided Flink configuration space and relatively inaccurate to the random Flink configuration space: all data points for each application are randomly located around the horizontal line of y=0, and the more accurate class has closer distance to the y=0, indicating that the performance model has a good orientation to guided configurations. For other programs, we also observe the similar results. Note that the more accurate performance models, the higher performance we can achieve by GML.

### 5.3 Why Mixed Configurations?

In Section 3.5, we analyzed that using a mixture of 50% of configurations generated by RCG and 50% of configurations generated by GCG as the initial data of GA is better than using configurations only generated by RCG. We now

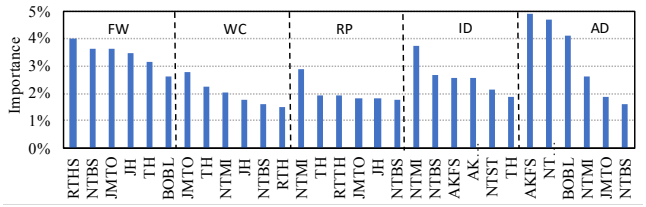



Fig. 9. The importance of the configuration parameters with respect to performance for the five Flink programs.

verify this assertion by experiments. Figure 8 shows how the 99th percentile latency (p99_latency) varies with the iteration numbers of the GA and the different mixtures of configurations generated by RCG and GCG as the initial configurations of the GA. For the (XX-YY) style legend in the figure, XX represents how the training data set used to train a performance model is generated and YY denotes how the initial data set of the GA is generated. For example, (GCG+RCG)-(GCG+RCG) means that a part of the training data is generated by GANs (GCG) and the others are generated randomly (RCG), and a part of the initial configurations of the GA is generated by GANs (GCG) and the others are generated randomly (RCG).

There are a couple of interesting observations to be made here. For one, if all the training data and all the initial configurations for the GA are only generated by RCG (the RCG-RCG curve), the GA takes the longest time (e.g., 90 iterations) to converge and the 99th percentile latency is the longest among all the approaches (curves) shown in Figure 8 (case A). Second, if we use the opposite extreme case (GCG+RCG)-(GCG+RCG), the GA takes the shortest time (e.g., 60 iterations) to converge and the 99th percentile latency is the shortest among all the approaches (case B). The GA convergence rates and 99th percentile latencies of other cases are in the ranges of those of the case A and case B. These results confirm our assertion that a mixture of configurations generated by RCG and the ones generated by GCG can speed up the searching process for the optimal configuration.

### 5.4 Configuration Parameter Importance

As a call-and-response for Section 3.5, Figure 9 shows the importance quantification of the configuration parameters for the five experimented Flink programs. Because of the limited space, we only show the six most important configuration parameters for each program. As can be seen, the importance quantification of the configuration parameters for a single Flink program are significantly different. Moreover, the importance ranks of different programs are different. This indicates that it is possible for the GA to speed up the search process if it makes the mutation operations only occur on the important configuration parameters. In other words, traditional GA mutates the configuration parameters randomly, which may waste time if the mutation occurs on the unimportant parameters with respect to performance.

### 5.5 Configurations Generated by GANs

The main purpose of the generative models of GANs is to generate configurations that conform with the distribution

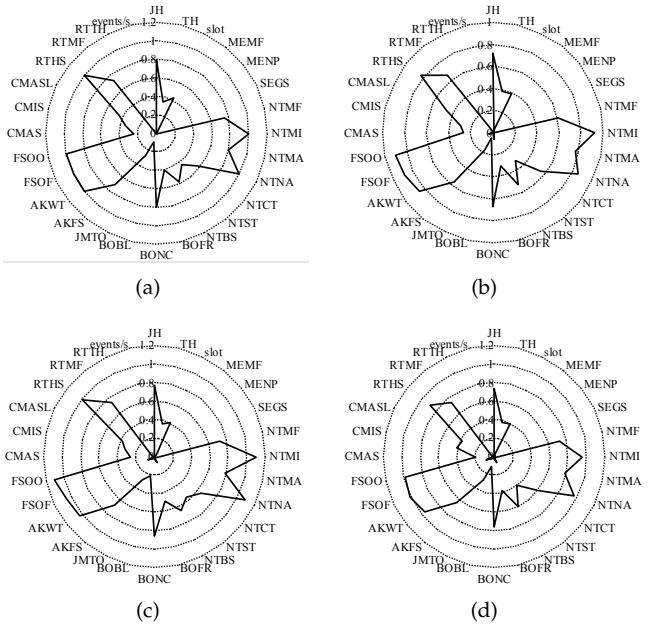



Fig. 10. The kiviat Plot represents the simulation generation capability of the generated model.(a) the real configurations. (b), (c), and (d) are the configurations generated by GANs.

of real configurations. We compare the distribution difference between the real configuration and generated ones using kiviat plots. Before drawing the plot, we normalize the data as fellows:

$$X_{nor} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

with $X_{nor}$ the normalized value of the $i^{th}$ configuration parameter, $x_i$ the value of the $i^{th}$ configuration parameter, $x_{max}$ and $x_{min}$ the maximum and minimum values of the $i^{th}$ configuration parameter, respectively.

Figure 10 shows the simulation generation capability of the generated model. (a) shows the kiviat plot of real configurations. (b), (c), and (d) show the configurations generated by the generative model of the GANs. As we can see, the shapes of (b) to (d) are quite similar to that of (a). This indicates that the GANs can capture the real configuration distribution well. The results can be seen in other experiments as well.

### 5.6 Iteration Number of the GA

We employ GA to iteratively search the huge configuration space to find the optimum configuration for a Flink program. The time (or the number of iterations) needed for convergence is our primary concern because longer time incurs higher cost to find the optimal configuration. Figure 11 shows how GA converges for all the experimented programs. We see that, a small number of Iterations, e.g. 20 to 50, are typically enough. Moreover, different programs may need different numbers of iterations. For example, $WC$, $FW$, and $ID$ need 24, 40, and 32 iterations to find the optimal configurations, respectively. This indicates that the combination of our proposed techniques such as the mixture of configurations generated by RCG and GCG, and the performance models can indeed speed up the search process for the optimal configuration for a Flink program.

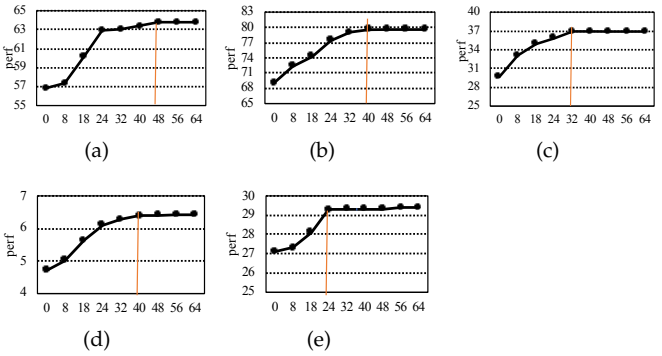



Fig. 11. The number of iterations for all programs, Advertisement(AD), Repartition(RP), Identity(ID), FixWindow(FW), WordCount(WC).

## 5.7 Speedup

We now evaluate the overall speedup obtained by the GML generated configurations over the default configuration parameter values and the state-of-the-art approach, DAC, generated configurations. For a fair play, we re-implement the DAC in our experimental environment. We define the speedup as fellows:

$$Speedup = perf/perf_{meth} \quad (6)$$

with $perf_{meth}$ the performance of an application with configurations generated by GML or DAC, and *perf* the performance with default configurations. The goal of comparing GML with DAC is to show whether GML outperforms the state-of-the-art approach for tuning configurations of big data systems. In addition, since thus far we have not found studies focusing on tuning the configurations of Flink programs, we also compare the configuration of a Flink program tuned by GML against its default configuration to let readers have an initial experience about how much performance improments can be obtained.

As aforementioned, latency and throughput are important performance metrics in stream processing systems. In general, when latency improves, the throughtput decreases, and vice versa. Thus, we consider throughput, p99_latency, and ratio of the two as the optimization metrics.

Figure 12 shows the speedups of 20 program-input pairs with the GML generated configurations over the default configurations and the DAC generated configurations. As shown in Figure 12(a), taking the ratio of throughput over p99_latency (higher is better) as the optimization metric, GML improves the ratios of the 20 programs-input pairs with default configurations by a factor of 2.2× on average and up to 3.7×. The minimum and maximum speedups of GML over DAC are 1.2× and 2.9×, respectively. On average, the speedup achieves 1.5×.

Figure 12(b) and Figure 12(c) are the speedups with GML over DAC and default configurations for considering throughput and p99_latency as optimization metrics. As can be seen, GML can improve the throughput, compared to default configurations by a factor of 1.3× on average and up to 2.4×. Moreover, GML outperforms DAC by a factor of 1.13× on average. As for the p99_latency, the average and maximum improvements of GML are1.7× and 2.3 ×, respectively, In addition, GML improves the p99_latency over DAC by a factor of 1.3× on average.

### 5.7.1 Discussion about Streaming Rate

For Flink programs, streaming rate significantly affects the program performance (e.g., tail latency or throughput). Theoretically, when the streaming rate changes, the optimal configuration of the program should be changed. This indicates that we should collect new profiling data to find a new optimal configuration for the program. However, practically this depends on how much the streaming rate changes. In our experiments, we observe that if the streaming rate variation is within [-15%, 15%] of a given rate which has been used to optimize performance based on a set of profiling data, it is unnecessary to re-collect new profiling data for a different streaming rate. In other words, the optimized configuration obtained by using the given streaming rate can be also the optimal one for other different streaming rates as long as they are in the range from *"the given rate – given rate * 15%"* to *"the given rate + given rate * 15%"* in our case. Nevertheless, if a different streaming rate is out of the range, we must collect new profiling data, incurring additional overhead. Fortunately, we find that one usually limits the streaming rate within a predefined range based on experience in industry, rather than lets it change wildly. In such a case, our approach can be easily used.

To observe how streaming rate affects performance of a Flink program, we have tried four significantly different rates:1,000,000 event/s, 2,000,000 events/s, 4,000,000 events/s, and 5,000,000 events/, and the results are shown in Figures 12 (a), (b), and (c). As can be seen, the performance improvements (e.g., tail latency, throughput, and throughput/latency) achieved by GML are different for significantly different streaming rates for all the experimented Flink programs. For example, GML only improves the throughput of *Repartition* by 4.8% when the streaming rate is 2,000,000 events/s while it improves that by 61.8% when the rate is 4,000,000 events/s. This indicates that we indeed need to collect new profiling data to find new optimal configruation for a Flink program for significantly different streaming rates.

## 5.8 Overhead

We now report the overhead of GML including the time used to collect training data, to train models, and to search optimal configurations. Table 4 shows the results. The unit for the time used for collecting data is hour, for model training is second, and for searching optimal configuration is minute. As can be seen, collecting data incurs the highest cost, 83.3 hours for GML while DAC needs 383.5 hours. The time needed by GML is shortened by $2.4\times$ compared to DAC! While it still seems long, it is a one-time cost and is still attractive compared to manually configuration. It is important to remember that the targets of GML are the stream processing applications which usually run in data centers for months or even longer. In this usage scenario, this high one-time cost is amortized with a very large number of runs. So, the additional cost per run is very low.

## 5.9 Results for the Real Production Cluster

Since it is difficult to get the throughput of the whole cluster in the company, we only show the results for p99_latency

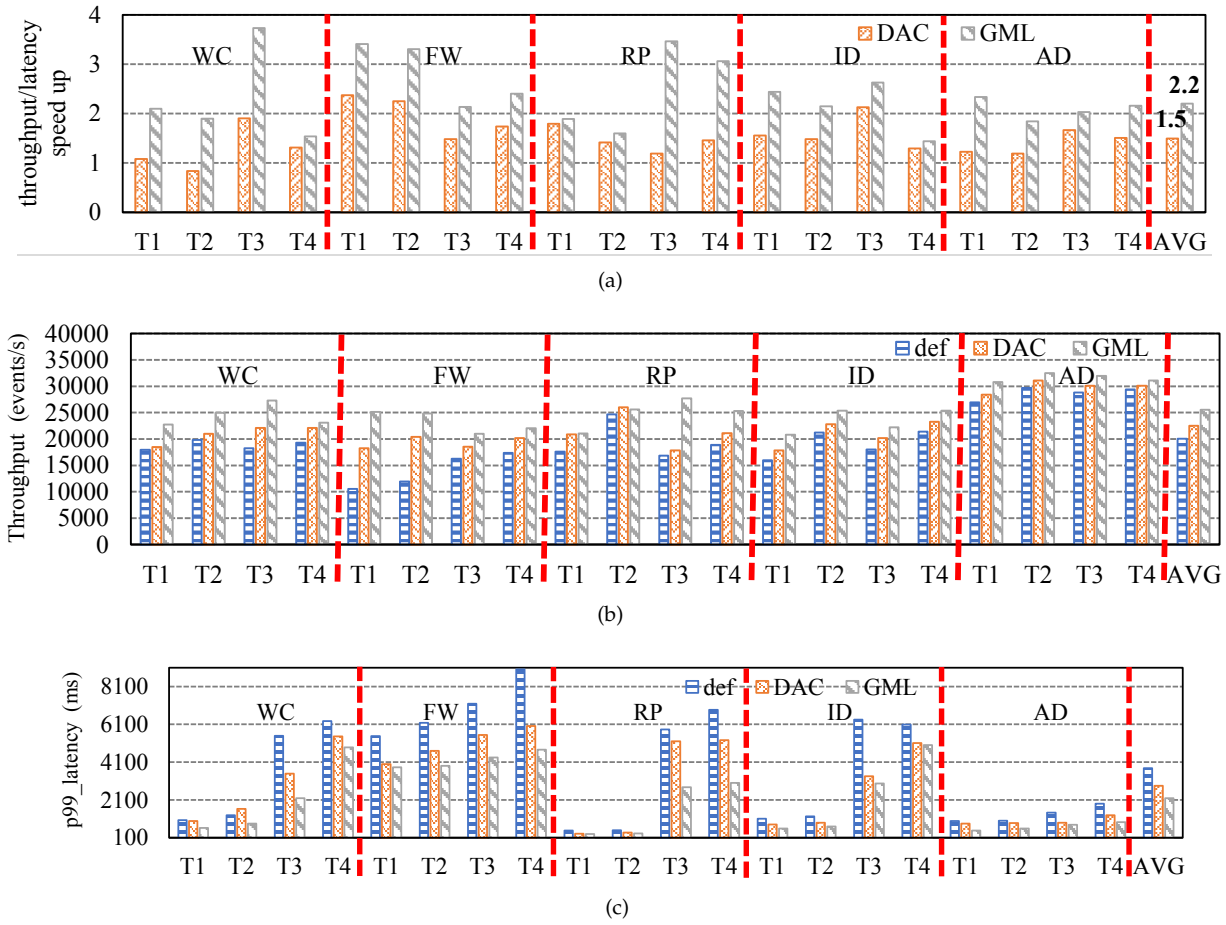



Fig. 12. Speedup for all programs WordCount(WC), FixWindow(FW), RePartition(RP), Identity(ID), Advertisement(AD) with GML generated configurations over with default configurations and DAC generated configurations . The T1,..,T4 of each program correspond to the input data speed from listed from 1,000,000 events/s, 2,000,000 events/s, 4,000,000 events/s, and 5,000,000 events/s. (a), (b), and (c) compare the ratio of throughput and p99_latency, throughput, and p99_latency, respectively.

TABLE 4
Time Cost

| Methodology | Collecting(h) | Modeling(s) | Searching(m) |
|---|---|---|---|
| GML | 83.3 | 42 | 4.5 |
| DAC | 283.5 | 23 | 9 |

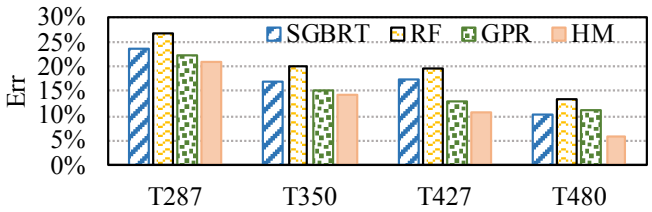



Fig. 13. The Model Accuracy for the real production cluster.

optimization. However, when we employ GML in a real production environment, we encounter a problem. It is difficult to build an accurate performance (p99_latency) model by using the SGBRT algorithm with a limited amount of training data even when we generate some data by using GANs. We have to try other ML algorithms and find the one, the HM model, proposed by DAC [1] is accurate enough when the number of training examples achieves 480 (e.g., the error $< 10\%$), as Figure 13 shows. We therefore choose the HM as the modeling technique for the real production environment while keep other parts of GML unchanged.

One may ask the reason why SGBRT works well for the lab cluster but not for the real production cluster. Generally, this is because of the data-dependent nature of machine learning. That is, a machine learning algorithm works well in one scenario may not work well in another scenario because the data being processed is changed. To further understand how the data is different. We conduct experiments to investigate the distribution of 99th percentile latencies produced by the lab cluster and the real production cluster. We employ kernel density estimation (KDE) [39] to observe the distribution of 99th percentile latencies. KDE is a really useful statistical tool to visualize the "shape" of data [39]. We use Gaussian (normal) function as the kernel function of KDE in this study.

Figure 14 shows the 99th percentil latency distribution of the lab cluster and Figure 15 shows that for the real production cluster. The X axes of the two figures represent the 99th percentile latency in millisecond and the Y axes denote the KDE of the 99th percentile latency. The calculation of KDE can be found in [39]. We observe that the distribution of 99th percentile latency of the lab experiments is similar to Gaussian distribution while that of the real production

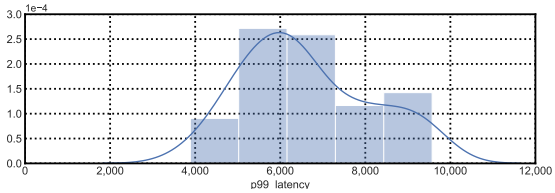



Fig. 14. The data distribution of 99th percentail latency observed on the lab cluster.The X axis represents the 99th percentile latency with a unit of millisecond. The Y axis denotes the kernel density estimation (KDE) of the 99th percentile latency.

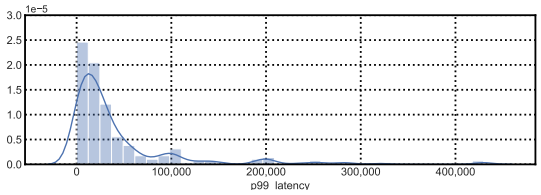



Fig. 15. The data distribution of 99th percentail latency observed on the real production cluster.The X axis represents the 99th percentile latency with a unit of millisecond. The Y axis denotes the kernel density estimation (KDE) of the 99th percentile latency.

experiments is similar to a long tail distribution. SGBRT is a stage-wise boosted decision tree, which is good to build models for data obeying Gaussian distribution but sensitive to outliers [40]. HM is an ensemble of traditional ensemble machine learning algorithms (e.g., SGBRT, random forest), which is more robust when building models for data obeying non-Gaussian distribution such as long-tail distribution with more outliers. We therefore choose HM for GML used in the real production cluster.

We compare the 99th percentile latency of the production Flink program configured by GML, by performance experts in the company, and by default configurations. In our experiments, GML finds two optimized configurations denoted by `conf1` and `conf2`. For the `conf1`, GML improves the p99_latency over the configuration made by the company by $3.7\times$ on average and up to $20.4\times$. Compared to the default configurations, the `conf1` provided by GML improves the p99_latency by $10.5\times$ on average and up to $57.8\times$. Compared to the configuration made by the experts in the company, the the `conf2` provided by GML improves the p99_latency by $4\times$ on average and up to $16.8\times$. In addition, the the `conf2` provided by GML improves the p99_latency over the default configurations by $11.5\times$ on average and up to $47.7\times$.

These results show that GML can be used in industry. However, the performance improvements on real production cluster do not show scientific achievements, which is not our aim either. We perform this comparison with two goals. First, the significant performance improvements acheived by GML on a real production cluster shows that GML can be used in signficantly different hardware environments and in turn has good adapatability. Second, the performance of the Flink program in the real production cluster has already been manually optimized by the performance experts in the company. The performance improvements made by GML demonstrates that GML outperforms the manual tuning approach used by performance experts in the practice of performance optimization.

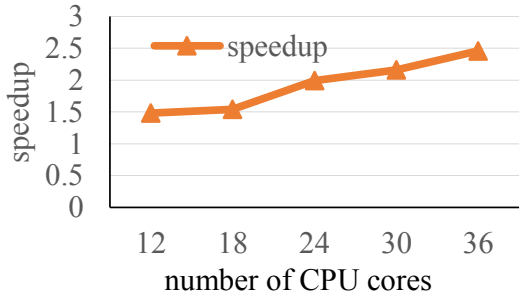



Fig. 16. The speedup of the 99th percentail latency of the "WordCount" observed on different scales of clusters. The X axis represents the number of ARM CPU cores.

### 5.10 Scalability

We now evauate the scalability of GML with respect to compute nodes by using mobile ARM processors because we do not have many standard servers. We create five mobile ARM clusters consisting of 2,3,4,5,and 6 ARM mobile processors. Each processor acts as a compute node and all the processors are connected by a 100MB/s ethernet network. Moreover, each processor has four little CPU cores, two big CPU cores, and 4GB memory. Hence, the five clusters consist of 12, 18, 24, 30, and 36 CPU cores, respectively.

Figure 16 shows the speedup of the 99th percentile latency of program *WordCount* tuned by GML over that by the default configurations. As can be seen, the speedup increases along with the incresing number of CPU cores. In the clusters consisting of 12, 18, 24, 30, and 36 CPU cores, GML accelerates *WordCount* over the default configuration by $1.48\times$, $1.54\times$, $1.99\times$, $2.16\times$, and $2.46\times$, respectively. This indicates that GML has a good scalability with respect to the number of CPU cores.

### 5.11 Improvments on a Heterogeneous Cluster

Our previous evalutions are all on homogeneous clusters. It would be interesting to know how GML performs on heterogeneous clusters. We therefore conduct an experiment to observe the 99th percentile latency of Flink programs on a cluster consisting of two servers equipped with X86 CPUs and two servers equipped with ARM CPUs. Figure 17 shows that the 99th percentile latencies of the experimented program-input pairs tuned by GML are signficantly shorter than those achieved by the default configurations on this deeply heterogenous cluster. The speedup achieved by GML over default configurations is $2.2\times$ on average and up to $4.1\times$ in this case. This indicates that GML can also adapt to heterogeneous clusters (hardware).

## 6 Related Work

Although there is no study having been done to tune the configurations for Flink program, a large class of related studies attempt to optimize the configurations of MapReduce/Hadoop applications. Herodotou *et al.* did a comprehensive survey about the configuration optimization of

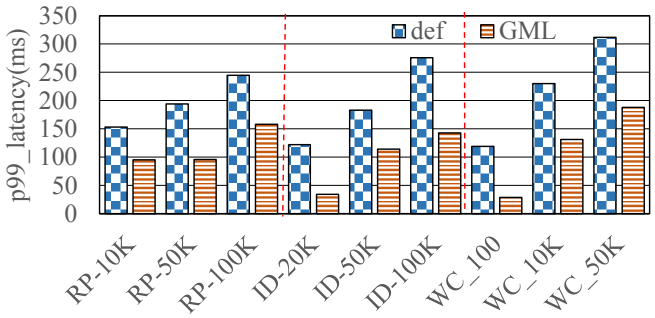



Fig. 17. The 99th percentail latency of Flink programs on a heterogeneous cluster.The X axis represents the program-input pairs. RP – Repartition, ID – Identity, WC - WordCount. 10K — 10,000 records/s, 50K – 50,000 records/s, and so on. These are the input data speed of Flink programs. *def* indicates the default configuration.

these big data systems [41]. They classify existing methods into six categories including rule-based, cost modeling, simulation-based, experiment-driven, machine learning, and adaptive approaches. Interested readers can read this paper for a comprehensive understanding about the parameter tuning of big data systems. We therefore do not repeat the related studies in this paper.

We instead describe the related work from the modelling approach perspective. Herodotou *et al.* [31], [32], [42] propose to build analytical performance models first and then leverage genetic algorithm to search the optimum configurations for Hadoop workloads. Adem *et al.* [33] suggest using a statistic reasoning technique named response surface (RS) to construct performance models for MapReduce/Hadoop programs and then implement the models in a MapReduce simulator. Zd.Bei *et al.* [12] propose a random forest based approach to automatically tune the configurations of Hadoop programs. These studies work well for Hadoop programs but not Spark. Therefore, Xuehai *et al.* propose a hierarchical model to build the performance model for Spark programs and in turn to tune the configurations of them [1]. Their work shows significant performance improvement for the Spark programs. In summary, the above mentioned approaches all use machine learning or statistic reasoning algorithms which need a large amount of training data by nature. Our approach is different from them, which aims to reduce the time needed to collect the training data while keep the same performance improvements.

Recently, GAN was proposed to help tune the configuration parameters of Apache Spark programs named ATCS [43], which is the closest work to ours. ATCS employes GAN to build performance models as functions of Spark configuration parameters. Our work is different. We leverage GAN to generate a part of training data for our performance models. Moreover, ATCS does not outperform DAC which is also design for optimizing the configurations of Spark programs in the aspect of speedup. In contrast, our GML does not only use less time for collecting training data but also improves more performance than DAC.

## 7 Conclusion

Apache Flink programs have more than 300 configuration parameters and a larger number of them are performance-critical. We propose an approach named guided machine learning (GML) to auto-tune the configurations of Flink programs. Compared to other ML based configuration auto-tuning approaches for big data engines, GML can significantly reduce the time needed for training data collection and optimal configuration searching by innovating a GANs based and a mean processing technique. Experimental results on a lab cluster and a production cluster show that GML significantly outperforms traditional approaches.

## 8 Acknowledgements

This work is supported by the Key R&D Program of Guangdong Province under No.2019B010155003 and NSFC under grant no 61672511, 61702495, and 61802384. This work is also supported by Shenzhen Institute of Artificial Intelligence and Robotics for Society (AIRS) at the Chinese University of Hong Kong (Shenzhen).

## References

[1] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 564–577.
[2] YARN, https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, accessed January 4, 2019.
[3] Flink, https://flink.apache.org/flink-architecture.html, accessed May 28, 2020.
[4] Hadoop, http://hadoop.apache.org/, accessed May 28, 2020.
[5] Spark, https://spark.apache.org/, accessed May 28, 2020.
[6] Storm, http://storm.apache.org/, accessed May 28, 2020.
[7] Ververica, https://www.ververica.com/, accessed May 28, 2020.
[8] N. Rivetti, Y. Busnel, and A. Gal, "Flinkman: Anomaly detection in manufacturing equipment with apache flink: Grand challenge," in *ACM International Conference on Distributed and Event-Based Systems*, 2017, pp. 274–279.
[9] M. Schwarzer, C. Breitinger, M. Schubotz, N. Meuschke, and B. Gipp, "Citolytics: A link-based recommender system for wikipedia," in *Eleventh ACM Conference on Recommender Systems*, 2017, pp. 360–361.
[10] Ververica, https://www.ververica.com/blog/apache-flink-user-survey-2017-recap, accessed May 28, 2020.
[11] A. F. developers, https://ci.apache.org/projects/flink/flink-docs-release-1.5/, accessed May 28, 2020.
[12] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng, "Rfhoc: A random-forest approach to auto-tuning hadoop's configuration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1470–1483, 2016.
[13] Z. Bei, Z. Yu, Q. Liu, C. Xu, S. Feng, and S. Song, "Mest: A model-driven efficient searching approach for mapreduce self-tuning," *IEEE Access*, vol. 5, pp. 3580–3593, 2017.
[14] Z. Bei, Z. Yu, N. Luo, C. Jiang, C. Xu, and S. Feng, "Configuring in-memory cluster computing using random forest," *Future Generation Computer Systems*, vol. 79, pp. 1–15, 2018.
[15] 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*. ACM, 2014, pp. 97–108.
[16] A. E. Gencer, D. Bindel, E. G. Sirer, and R. van Renesse, "Configuring distributed computations using response surfaces," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 235–246.
[17] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based autotuning of mapreduce," in *European Conference on Parallel Processing*. Springer, 2013, pp. 406–419.
[18] M. Wasi-Ur-Rahman, N. S. Islam, X. Lu, D. Shankar, and D. K. Panda, "Mr-advisor: A comprehensive tuning tool for advising hpc users to accelerate mapreduce applications on supercomputers," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on*. IEEE, 2016, pp. 198–205.

[19] S. Pellegrini, R. Prodan, and T. Fahringer, "Tuning mpi runtime parameter setting for high performance computing," in *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*. IEEE, 2012, pp. 213–221.

[20] A. Saboori, G. Jiang, and H. Chen, "Autotuning configurations in distributed systems for performance improvements using evolutionary strategies," in *The 28th International Conference on Distributed Computing Systems*. IEEE, 2008, pp. 769–776.

[21] HiBench, https://github.com/intel-hadoop/hibench, accessed May 28, 2020.

[22] Yahoo!Benchmark, https://github.com/yahoo/streaming-benchmarks, accessed May 28, 2020.

[23] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[24] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernznzez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, 2015.

[25] D. Warneke and O. Kao, "Nephele: efficient parallel data processing in the cloud," in *Proceedings of the 2nd workshop on many-task computing on grids and supercomputers*. ACM, 2009, p. 8.

[26] Mesos, http://mesos.apache.org/, accessed May 28, 2020.

[27] Kunbernetes, https://kubernetes.io/, accessed May 28, 2020.

[28] Flink, https://ci.apache.org/projects/flink/flink-docs-release-1.7/ops/config.html, accessed May 28, 2020.

[29] R. Grob, Y. Gu, W. Li, and M. Gauci, "Generalizing gnas: A turing perspective," in *The 31st Conference on Neural Information Processing Systems (NIPS 2017)*, 2017, pp. 1–11.

[30] I. J. Goodfellow, J. Pougetabadie, M. Mirza, B. Xu, D. Wardefarley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," pp. 2672–2680, 2014.

[31] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics." in *Proceedings of the Biennial International Conference on Innovative Data Systems Research (CIDR)*, Jan. 2011, pp. 261–272.

[32] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.

[33] A. E. Gencer, D. Bindel, E. G. Sirer, and R. van Renesse, "Configuring distributed computations using response surfaces," in *Proceedings of the Annual ACM/IFIP/USENIX Middleware Conference*, Dec. 2015, pp. 235–246.

[34] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for large-scale network parameter configuration," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 196–205, 2003.

[35] J. W. Escobar, R. Linfati, P. Toth, and M. G. Baldoquin, "A hybrid granular tabu search algorithm for the multi-depot vehicle routing problem," *Journal of heuristics*, vol. 20, no. 5, pp. 483–509, 2014.

[36] M. Albayrak and N. Allahverdi, "Development a new mutation operator to solve the traveling salesman problem by aid of genetic algorithms," *Expert Systems with Applications*, vol. 38, no. 3, pp. 1313–1320, 2011.

[37] Kafka, https://kafka.apache.org/, accessed May 28, 2020.

[38] S. Chintapalli, B. J. Peng, P. Poulosky, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, and K. Nusbaum, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, 2016, pp. 1789–1792.

[39] M. Conlen, "Kernel density estimation," https://mathisonian.github.io/kde/, accessed October 11, 2020.

[40] A. H. Li and J. Bradic, "Boosting in the presence of outliers: Adaptive classification with nonconvex loss functions," *Journal of the American Statistical Association*, vol. 113, no. 522, 2017.

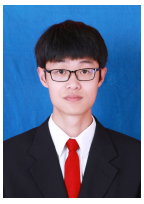
[41] H. Herodotou, Y. Chen, and J. Lu, "A survey on automatic parameter tuning for big data processing systems," *ACM Computing Surveys*, vol. 53, no. 2, pp. 43:1–43:37, 2020.

[42] H. Herodotou, "Hadoop performance models," Duke University, Tech. Rep., 2011.

[43] M. Li, Z. Liu, X. Shi, and H. Jin, "Atcs: Auto-tuning configurations of big data frameworks based on generative adversarial nets," *IEEE Access*, vol. 8, pp. 50 485–50 496, 2020.


**Yijin Guo** got her bachelor degree from Zhengzhou University, majored in software engineering. Now she is a research assistant working on optimizing the performance of cloud computing systems, edge computing systems, and big data systems.

**Huasong Shan** received the PhD degree in computer engineering from Louisiana State University-Baton Rouge, in 2017. He received the MS and BS degree in computer science and technology, Huazhong University of Science and Technology, China, in 2003 and 2006, respectively. He has been working with JD.com American Technologies Corporation, Mountain View, California as staff scientist. His research interests include system security, network security, web security, application of AI on system and security, cloud computing, AIOps, NLP, etc. He has served on the program committee of several international conferences, e.g., USENIX Security'20, EuroSys'20, SoCC'19, WWW'19, ICDCS'20, etc.



**Shixin Huang** got his bachelor degree from Zhengzhou University, majored in software engineering. Now he is a master student working on optimizing the performance of big data systems, especially Flink programs.

**Kai Hwang** is presently a Presidential Chair Professor in Computer Science and Engineering at the Chinese University of Hong Kong (CUHK), Shenzhen, China. He also serves as a Chief Scientist at the Cloud Computing Center, Chinese Academy of Sciences. He has taught at the University of Southern California and at Purdue University for 46 years prior joining CUHK. He received the Ph.D. in Electrical Engineering and Computer Science from UC Berkeley. Dr. Hwang has published extensively in the fields of parallel processing, cloud computing, and network security.

**Jianping Fan** received his PhD degree in computer science from Institute of Software, Chinese Academy of Science. Now he is a professor in Shenzhen Institute of Advanced Technology (SIAT), Chinese Academy of Science (CAS). He is also the dean of SIAT. His research interests include high performance computing, computer architecture, architecture supported big data, cloud, artificial intelligence systems.

**Zhibin Yu** received his PhD degree in computer science from Huazhong University of Science and Technology (HUST) in 2008. He visited the Laboratory of Computer Architecture (LCA) of ECE of the University of Texas at Austin for one year and he worked in Ghent University as a postdoctoral researcher for half of a year. Now he is a professor in SIAT. His research interests are micro-architecture simulation, computer architecture, workload characterization and generation, performance evaluation, multi-core architecture, GPGPU architecture, virtualization technologies, big data processing and so forth. He won the outstanding technical talent program of Chinese Academy of Science (CAS) in 2014 and the 'peacock talent' program of Shenzhen City in 2013. He is a member of IEEE and ACM. He serves for ISCA 2013, 2015, 2020, MICRO 2014, HPCA 2015, 2018, PACT 2016, and ICS2018, 2019.