Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing

Zhibin Yu
Shenzhen Institute of Advanced Technology, CAS
Shenzhen, China
zb.yu@siat.ac.cn

Zhendong Bei
Shenzhen Institute of Advanced Technology, CAS
Shenzhen, China
zd.bei@siat.ac.cn

Xuehai Qian
University of Southern California
Los Angeles, California
xuehai.qian@usc.edu

Abstract

In-Memory cluster Computing (IMC) frameworks (e.g., Spark) have become increasingly important because they typically achieve more than 10× speedups over the traditional On-Disk cluster Computing (ODC) frameworks for iterative and interactive applications. Like ODC, IMC frameworks typically run the same given programs repeatedly on a given cluster with similar input dataset size each time. It is challenging to build performance model for IMC program because: *1)* the performance of IMC programs is more sensitive to the size of input dataset, which is known to be difficult to be incorporated into a performance model due to its complex effects on performance; *2)* the number of performance-critical configuration parameters in IMC is much larger than ODC (more than 40 vs. around 10), the high dimensionality requires more sophisticated models to achieve high accuracy.

To address this challenge, we propose DAC, a datasize-aware auto-tuning approach to efficiently identify the high dimensional configuration for a given IMC program to achieve optimal performance on a given cluster. DAC is a significant advance over the state-of-the-art because it can take the size of input dataset and 41 configuration parameters as the parameters of the performance model for a given IMC program, — unprecedented in previous work. It is made possible by two key techniques: *1)* Hierarchical Modeling (HM), which combines a number of individual sub-models in a hierarchical manner; *2)* Genetic Algorithm (GA) is employed to search the optimal configuration. To evaluate DAC, we use six typical Spark programs, each with five different input dataset sizes. The evaluation results show that DAC improves the performance of six typical Spark programs, each with five different input dataset sizes compared to default configurations by a factor of 30.4× on average and up to 89×. We also report that the geometric mean speedups of DAC over configurations by default, expert, and RFHOC are 15.4×, 2.3×, and 1.5×, respectively.

CCS Concepts • **Computer systems organization** → **Cluster architectures**; • **Software** → *In-memory computing*;

Keywords Big data, In-memory computing, Performance tuning

ACM Reference Format:
Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/http://dx.doi.org/10.1145/3173162.3173187

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/http://dx.doi.org/10.1145/3173162.3173187

1 Introduction

Recently, In-Memory cluster Computing (IMC) frameworks such as Spark [41, 59] achieve much higher performance than traditional On-Disk cluster Computing (ODC), e.g., MapReduce/Hadoop [7, 45] and Dryad [17], for iterative and interactive applications. As a result, IMC has become increasingly popular in the past few years. A survey from Typesafe Inc. [47] shows that more than 500 organizations, including big companies such as Google and many more small companies, use Spark in production as of early 2015. Spark has been used in a wide range of domains including machine learning [30], graph computing [51], streaming computing [44], and database management [3].

Typically, an IMC program runs repeatedly with similar input dataset sizes. For example, the e-companies on Taobao need to sort their products according to the saleroom every day or every week [36]. In this scenario, the input dataset size of different runs of the same program is almost the same because it is determined by the total number of the sorted products of an e-company. Depending on e-companies, the data content may be significantly different. We call this kind of programs as *periodic (daily, weekly, and etc.) long jobs* [36].

To achieve optimal performance of periodic long jobs, the *end users* are required to determine a large number of performance-critical configuration parameters. We consider

the key challenge as the *high dimensional configuration issue*. For example, the performance of a Spark program can be determined by more than 40 configuration parameters such as `spark.executor.cores`, which specifies the number of cores, and `spark.executor.memory`, which specifies the amount of memory used by the job, with nonlinear interactions. As a result, manually tuning the configuration parameters of a given IMC program to achieve the optimal performance on a given cluster is extremely difficult. It strongly motivates an auto-tuning approach to automatically generate configuration parameters.

Due to the number of configuration parameters, naively running a program with all configurations and choosing the one with the best performance is not feasible because of: *1)* the huge number of parameter combinations; and *2)* the long accumulative running time, — the execution of each IMC program may take several minitues to hours. To overcome the challenge, performance models are investigated to predict the execution time of a program with a given configuration, making the configuration search dramatically faster than the naive approach. Currently, this approach is widely used to tune the performance of ODC programs [4, 5, 10, 12–14, 19, 25, 48], and traditional distributed systems [35, 37]. The key is that the performance models must be highly accurate with tolerable overhead in a certain environment such as ODC or HPC. Otherwise, the optimal performance can not be achieved.

To tune the performance of IMC program, a straightforward solution is to follow the same approach and apply the existing performance models. However, two factors make it not as trivial as expected: *1)* the performance of IMC programs is more sensitive to the size of input dataset, which is known to be difficult to be incorporated into a performance model due to its complex effects on performance [52]; *2)* the number of performance-critical configuration parameters in IMC is much larger than ODC (more than 40 vs. around 10) [13], the high dimensionality requires more sophisticated models to achieve high accuracy. Due to the *combination of the two factors*, building an accurate performance model for IMC programs is quite challenging. This observation is supported by recent works. For example, [35] builds its model with a large number of parameters (e.g., 27) but without considering input dataset size. In contrast, [49][57] take input dataset size as a parameter of their performance model but they only consider 15 and 4 configuration parameters, respectively.

To overcome the challenge, this paper proposes *DAC*, a datasize-aware auto-tuning approach, that efficiently identifies the optimal high dimensional configuration parameters for IMC programs. DAC is a significant advance over the state-of-the-art because it can take *the size of input dataset and 41 configuration parameters* as the parameters of the performance model for a given IMC program, — *unprecedented* in previous work. It is made possible by two key techniques: *1)* Hierarchical Modeling (HM), which combines a number of individual sub-models in a hierarchical manner. It is more practical to construct an accurate model by combining multiple simpler sub-models than building an accurate sophisticated large model. HM is performed recursively, producing *first-*, *second-*, or *higher-order* hierarchical models. *2)* Genetic Algorithm (GA) is employed to search the optimal configuration. While data skew may affect the performance of an IMC program, we do not consider it in this paper for a practical reason: the configuration parameters of current IMC frameworks can only specify *the same settings for all tasks*. In another word, a parameter related to data skew cannot be conveyed in the configuration to affect execution.

To evaluate DAC, we use six typical Spark programs, each with five different input dataset sizes. The results show that DAC improves the performance of *all* 30 executions compared to default configurations by a factor of 30.4× on average and up to 89×. To compare with the configurations generated by DAC to auto-tuning approaches for ODC programs, we reimplement RFHOC [4], the state-of-the-art approach, in the context of Spark. We demonstrate that DAC significantly outperforms RFHOC by a geometric mean speedup of 1.5×, and even configurations determined by an expert by 2.3× (geometric mean speedup).

The rest of the paper is organized as follows. Section 2 discusses the background and motivation. Section 3 explains the DAC methodology. Section 4 describes our experimental setup. Section 5 presents the results and analysis. Section 6 discusses the related work and Section 7 concludes the paper.

2 Background and Motivation

2.1 The Spark Framework

Spark [41] is a MapReduce-like IMC framework with APIs in Scala, Java, and libraries for streaming, graph processing, and machine learning [41], which can not be efficiently processed by the traditional MapReduce ODC frameworks such as Hadoop [45]. The key is that Spark employs a data structure called Resilient Distributed Datasets (RDDs) [59] to keep reusable intermediate results in memory. In fact, RDD is an abstraction of cluster memory and allows users to explicitly control the partitioning to optimize data placement, as well as manipulate it by a set of operators such as `map` and `hash-join`.

As shown in Figure 1, the Spark framework generates a DAG (directed acyclic graph) based on the codes of a Spark application (job) when it is submitted. Subsequently, the DAG is split into a collection of stages (e.g., `stage1` and `stage2`) with each containing a set of parallel tasks. Each task, one per RDD partition (shown as `p` in the figure), computes partial results of a Spark job. One Spark job may have a number of stages, each of which may depend on other stages [20]. This dependency is called *lineage* stored in a

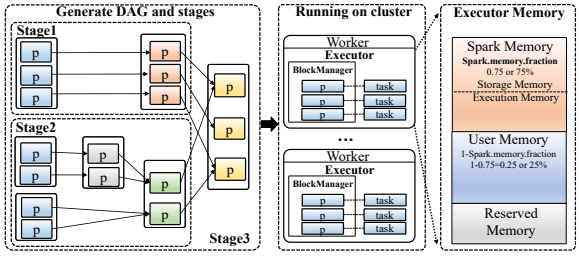

Figure 1. An Overview of Spark Workflow.

RDD. Next, the DAG scheduler schedules the stages to execute by a number of executors, as shown in Figure 1. The resources can be used by an executor are specified by configuration parameters. For example, `spark.executor.memory` specifies the memory size allocated for an executor. This block of memory is further divided into *execution memory*, *user memory*, and *reserved memory* (e.g., 300 MB) and their sizes are controlled by `spark.memory.fraction` [32].

In summary, a Spark job is controlled by up to 160 configuration parameters. They specify fourteen aspects including *application*, *runtime environment*, *shuffle behavior*, *data serialization*, *memory management*, *execution behavior*, *networking*, *Spark UI*, *scheduling*, *dynamic allocation*, *security*, *encryption*, *sparkstreaming*, and *sparkR*. Some parameters such as `spark.application.name` do not affect performance at all while others such as `spark.executor.cores` and `spark.executor.memory` significantly do. We find that, there are 41 parameters that can be easily tuned *and* significantly affect performance. We therefore focus on tuning them in this paper. Note that Spark framework is just used as an example of IMC to evaluate DAC, the principles of DAC can be easily applied to other computing systems such as HBase [40] which also requires end users to set a large number of configuration parameters.

2.2 Motivation

As mentioned above, Spark has 41 configuration parameters to be tuned, similarly, Hadoop, a traditional ODC framework also has a number of parameters. In the following, we attempt to answer two questions: *1)* whether enough parameters have been taken into account for Spark (IMC)? and *2)* whether the current performance modeling techniques for ODC can be applied to IMC?

2.2.1 Input Dataset Size Sensitivity

We study two programs with two implementations for each: Spark `KMeans` (`Spark-KM`), Hadoop `KMeans` (`Hadoop-KM`), Spark `PageRank` (`Spark-PR`), and Hadoop `PageRank` (`Hadoop-PR`). We run these four programs, with two input datasets (`input-1` and `input-2`) for 200 times with a different configurations on the same cluster. The goal is observe the *execution time variation*. For `KMeans`, `input-1` and `input-2` contain 40 and 80 millions of records (around 18 GB), respectively. For `PageRank`, `input-1` and `input-2` are two page collections with 500

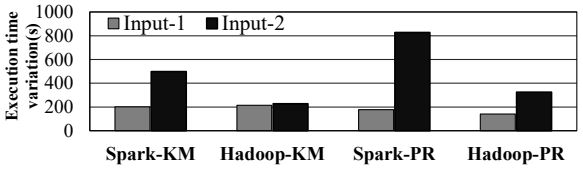

Figure 2. Execution time variation between two implementations (Spark and Hadoop) of two programs: (*KMeans* and *PageRank*) with the same two input datasets when the configurations are randomly changed by 200 times. The Y-axis represents the execution time variation which is equal to the difference between the maximum and minimum execution time observed during the 200 experiments. KM — *KMeans*, PR — *PageRank*.

thousands and 1 million pages (around 100 GB), respectively. To generate each configuration, we randomly generate a value for each configuration parameter within its value range, these values form a configuration vector ($\{c_1, c_2, ..., c_{41}\}$). For all experiments, each with a different random configuration, we observe the maximum and normal execution time (T_{max} and T_i) for each program-input pair. We then use T_{var} defined by Equation (1) to represent the execution time variation caused by different configurations.

$$T_{var} = (\sum_{i=1}^{n} T_{max} - T_i)/n, \quad (1)$$

with n the total number of different configurations.

Figure 2 shows the execution time variation. We clearly see that the T_{var} for program-input pair (`Spark-KM`, `input-2`) is 2.6× of that for (`Spark-KM`, `input-1`); similarly, the T_{var} for (`Spark-PR`,`input-2`) is 4.3× of that for (`Spark-PR`,`input-1`). In contrast, the T_{var} for program-input pair (`Hadoop-KM`, `input-2`) is only 0.97× of that for (`Hadoop-KM`, `input-1`). Similarly, the T_{var} for (`Hadoop-PR`, `input-2`) is 1.76× of that for (`Hadoop-PR`, `input-1`). The results indicate that the execution time of Spark programs with different configurations is more sensitive to input dataset size compared to Hadoop programs. This key difference can be explained by the natural difference between IMC and ODC: by placing as much as possible data in memory, the performance of IMC is higher but more sensitive to the slight perturbations. In contrast, ODC programs typically involve more slow I/O operations, which is more stable with perturbations. This study motivates the consideration of dataset size in performance modeling.

2.2.2 Limitations of ODC Modeling Techniques

Analytical modeling [12–14], statistic reasoning [10, 35], and machine learning techniques [4, 19], have been used to construct the performance model as a function of configuration parameters for a Hadoop program or a high performance computing program. Our goal is to investigate whether the models built by these techniques are accurate enough for Spark programs when the size of input dataset is taken into account and the number of configuration parameters is as

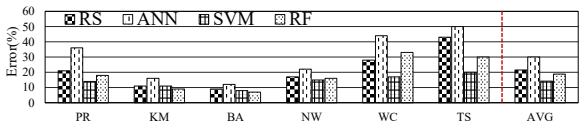

Figure 3. The prediction errors of the models constructed by response surface (RS), artificial neural network (ANN), support vector machine (SVM), and random forest (RF).

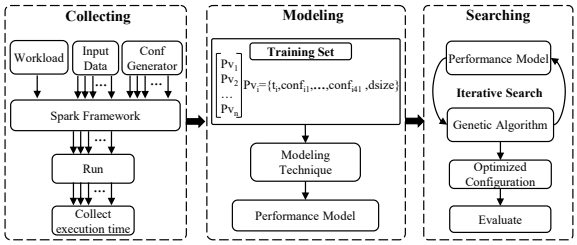

Figure 4. Block Diagram of DAC.

large as 41. To this end, we employ the statistic reasoning technique—response surface (RS) used in [10] and the three machine learning techniques — artificial neural network (ANN) used in [21], support vector machine (SVM) used in [19], and random forest (RF) used in Hadoop configuration auto-tuning [4] to construct performance models for six Spark programs (the experimental methodology is discussed in Section 4). We do not try analytical modeling techniques [12–14] because it is already known that they suffer from low accuracy caused by over-simplified assumptions [4]. To compare the model accuracy, we define prediction error as follows.

$$err = \frac{|t_{pre} - t_{mea}|}{t_{mea}} \times 100\%, \tag{2}$$

in which t_{pre} is the predicted execution time of a Spark program-input pair and t_{mea} is the real execution time. Therefore, err reflects the relative difference between the prediction and real measurement of a Spark program-input pair's execution time, and lower is better.

Figure 3 shows the prediction error comparison between the models constructed by the four modeling techniques. We see that the average errors of models built by RS, ANN, SVM, and RF are 23%, 27%, 14%, and 18%, respectively. We believe that performance models with such high errors can not accurately identify the optimal configurations. Therefore, our experiments demonstrate that existing modeling techniques fail to build accurate performance models when the input dataset size and 41 configuration parameters are considered for Spark programs. This motivates the seek of new modeling techniques.

3 DAC Methodology

DAC is a configuration tuning approach that automatically adjusts the values of configuration parameters to optimize performance for a given Spark program on a given cluster. It is designed for Spark in a popular usage scenario in industry: a Spark program *repeatedly runs many times with similar sizes of input datasets*, while the data contents are different.

Figure 4 illustrates the block diagram of DAC, which consists of three components: *collecting*, *modeling*, and *searching*. The *collecting* component drives the experiments: it generates a number of configurations, automatically runs IMC programs with the generated configurations, and collects the execution times of the experiments. The *modeling* component constructs a performance model as a function of the *high dimensional* configuration parameters and the *size of input datasets* for a given Spark program. The key innovations in this component enable DAC to generate performance model with such a large number of parameters, — unprecedented in previous auto-tuning approaches.

The *searching* component automatically searches the configuration that produces optimal performance. Overall, the *modeling* component relies on the results of the *collecting* component, and the *searching* component selects the best configuration from the outcome of *modeling* component.

3.1 Collecting Data

The goal of collecting performance data is to observe the execution behavior of Spark programs with different configurations. Then, the observed behavior will be used to build accurate performance models in other components. For a given Spark program, we collect performance data as follows.

1) We develop a configuration generator (CG) to generate a configuration which is a vector containing n configuration parameter values each time.

$$conf_i = \{c_{i1}, c_{i2}, ..., c_{ij}, ..., c_{in}\} \tag{3}$$

where $conf_i$ is the i^{th} configuration and c_{ij} is the value of the j^{th} configuration parameter in the i^{th} configuration. c_{ij} is randomly generated within its value range by CG. n is 41 corresponding to the 41 configuration parameters of Spark (shown in Table 2) that can be easily adjusted and significantly affect performance.

(2) We use the input dataset generator (DG) of each program to generate m input datasets with significantly different sizes. The size difference between any two datasets is at least 10%:

$$\frac{|DS_p - DS_q|}{min(DS_p, DS_q)} \times 100\% \geq 10\% \tag{4}$$

where DS_p and DS_q are the sizes of the p^{th} and q^{th} input datasets, respectively ($p < m$ and $q < m$). We set m to 10 to achieve a good trade-off between the size diversity of the input datasets and the time to collect the performance data. A larger m increases the time needed to collect performance data, whereas a smaller m decreases the size diversity of input datasets and in turn fails to reflect the influence of the size of input datasets on the configuration for optimized performance.

(3) We call a program and its input dataset a *program-input pair*, therefore, we have 10 program-pairs for a given

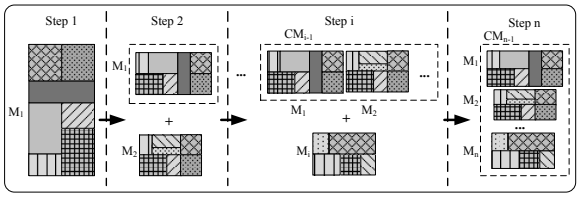

Figure 5. Overview of Hierarchical Modeling.

program. Subsequently, we run each program-input pair k times with k different configurations. After the execution of a program-input pair finishes, we construct a vector to store the execution time and the corresponding configuration:

$$Pv_i = \{t_i, c_{i1}, c_{i2}, ..., c_{ij}, ..., c_{in}, dsize_i\}, i = 1, ..., k \quad (5)$$

where Pv_i is the performance vector of the i^{th} execution of the program-input pair, t_i is the execution time of the i^{th} execution, c_{ij} is the value of the j^{th} configuration parameter in the i^{th} execution, $dsize_i$ is the size of the input dataset used in the i^{th} execution, and k is the number of times that the program-input pair is executed. Again, the value of k needs to be determined upfront and to be balanced: a large k increases the time needed to collect performance data, whereas a small k fails to collect enough amount of data for training accurate models.

3.2 Modeling Performance

As mentioned in Section 2.2.2, response surface (RS), artificial neural network (ANN), support vector machine (SVM), and random forest (RF) algorithms fail to build accurate enough performance models for Spark programs. It implies that the performance influenced by the *combination* of the *size of input datasets* and *high dimensional configuration parameters* of a Spark program shows much more complex behavior than what is only influenced by low dimensional configuration parameters. This is why existing approaches cannot accurately capture it.

Considering the principles of the four existing modeling techniques, they all try to build highly accurate and sophisticated *individual* models based on training data. This typically leads to "over-fitting" problem that pervasively exists in statistic reasoning and machine learning algorithms. The problem is exaggerated with larger number of parameters. Based on this insight, we propose a Hierarchical Modeling (HM) approach. The key idea is to predict the performance by the cooperation of *multiple* simpler models, rather than a *single* sophisticated model.

Figure 5 illustrates HM mechanisms. In the first step, we build an individual model (i.e., sub-model), M_1, as a function of the size of input datasets and configuration parameters. The sub-model can be built by different modeling techniques such as *ANN* and *SVM* but not by analytical modeling because of its over-simplified assumption. For simplicity, we employ regression tree [22] to build the sub-models. We thereby need to determine the tree complexity or tree size (the number of nodes in a tree) to minimize prediction error, e.g., execution time prediction error in the context of Spark configuration space. In the second step, a different regression tree model (M_2) is built to reflect the variation in the execution time of a Spark job that is not captured by M_1.

At this point, an initial combined model, CM_1, is created by combining the first two sub-models: $\alpha_1 M_1 + \alpha_2 M_2$, where M_1 and M_2 are the predicted the execution time by the two sub-models, and α_1 and α_2 are the respective coefficients corresponding to learning rate. This procedure is performed a number of times, and more sub-models are added to the combined model. The number of times is determined by the convergence of the model and the target accuracy such as 90%. If the target accuracy is met before the convergence, then we have obtained the final model or *first-order* model. If the target accuracy can not be satisfied after the combined model (e.g., TM_1) converges, we repeat the above procedure to build another combined model TM_2. Then, we perform the second level combination: $\beta_1 TM_1 + \beta_2 TM_2$, where β_1 and β_2 are the corresponding coefficients of TM_1 and TM_2. This combined model is called *second-order* model. This procedure can be performed recursively, e.g., more different levels of models are hierarchically added to the final combined model, until the target accuracy is satisfied.

Essentially, HM is a hierarchical sequential process in which the original model remains unchanged at each step. However, the execution time variation gradually reduces as the HM proceeds and the model becomes more accurate. Moreover, we introduce randomness into the HM process to improve accuracy and convergence speed. It also helps to mitigate the "over-fitting" problem. We will determine several model parameters, including *tree complexity* and *learning rate* in Section 5.

To build a performance model using HM, the first step is to construct a training set S which is a matrix as follows.

$$S = (Pv_j), j = 1, 2, ..., 10 \times k \quad (6)$$

where Pv_j is the j^{th} performance vector obtained by the *collecting* component. 10 corresponds to the number of input datasets for each program and k represents the number of different configurations used to execute a program-input pair. Subsequently, we input the matrix S to HM to build a performance model which can be described by:

$$t = f(c_1, c_2, ..., c_i, ..., c_n, dsize) \quad (7)$$

where t is the execution time of a program with input size of $dsize$ and the configuration is $\{c_1, c_2, ..., c_i, ...c_n\}$. Note that $f(c_1, c_2, ..., c_i, ...c_n, dsize)$ is a data model which means there is no formula for it. The formal description of constructing models by HM is illustrated by Algorithm 1. Note that although TM1 and TM2 (line 5 and 6) in *HigherOrderProcedure* (order) call the same function, they are different because we introduce randomness in the procedure.

Finally, we use the *collecting* component to collect a number (*num*) of performance vectors (shown in equation (5)), which are different from those in the matrix S to cross-validate the accuracy of the performance model. According to the accepted/standard practice in statistical reasoning and machine learning, we set *num* to a quarter of the size of the training set S, which is $(10 \times k)/4$.

Algorithm 1 The procedure of HM algorithm.

Input: S, tc(tree complexity), nt(number of trees), lr(learning rate)
Output: FM (the function defined by Equation (7))

Set the $accuracy$ of TM to 0 and order to 1
WHILE (the $accuracy$ of TM is lower than the target accuracy)
if (order == 1) **then**
    TM = **FirstOrderProcedure**(S)
else
    $TM = TM \times lr +$ **HigherOrderProcedure**$(order - 1) \times lr$
end if
order = order + 1
ENDWHILE
Set FM to TM and return FM
END the procedure of HM algorithm

HigherOrderProcedure(order)
if (order == 1) **then**
    TM = **FirstOrderProcedure**(S)
else
    TM_1 = **HigherOrderProcedure**(order-1)
    TM_2 = **HigherOrderProcedure**(order-1)
    $TM = TM_1 \times lr + TM_2 \times lr$
end if
if (the $accuracy$ of TM is higher than the target accuracy) **then**
    Set FM to TM and return FM
else
    Return TM
end if
END HigherOrderProcedure

FirstOrderProcedure(S)
Building a regression tree M_1 with tc nodes on a Bootstrap sample from S
for (i from 1 to nt) **do**
    Building a regression tree M_i with tc nodes on a Bootstrap sample from S
    $TM = TM + M_i \times lr$
    if (the $accuracy$ of TM is more than target accuracy) **then**
        Set FM to TM and return FM
    end if
    if (converge) **then**
        Break
    end if
end for
Return TM
END FirstOrderProcedure

3.3 Searching Optimal Configuration

In this section, we will describe how to search the optimum configuration for a Spark program. There exist many algorithms to search complex configuration spaces, e.g., recursive random search [56], pattern search [46], and genetic algorithm (GA) [18, 26]. Random recursive search is sensitive to getting stuck in local optima; pattern search typically suffers from slow local (asymptotic) convergence rates [46]. GA is a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover [26]. In particular, GA is well-known for being robust against local optima [18]. Our goal is to find the configuration for optimized performance of a Spark program-input pair from the global space of configuration parameters, which is a complex space to explore with many local optima. We therefore employ GA.

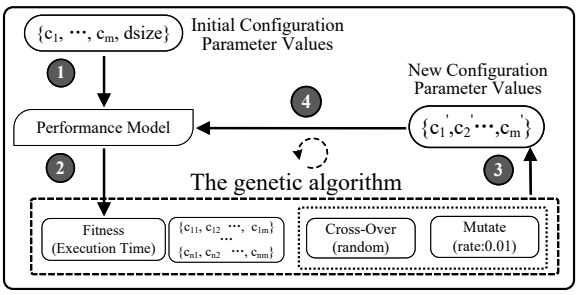

Figure 6. Searching the optimum configuration.

Figure 6 elaborates the searching procedure which consists of four steps. In step ①, we input a set of initial values of the configuration parameters and the size of an input dataset to the performance model of a Spark program, and the performance model outputs an execution time. In step ②, we pass the execution time and the configuration parameter values to the GA. Note that the configuration parameter values are *popSize* (which is a GA parameter) vectors randomly selected from S but the element t_i of each vector is removed. The GA then performs a number of operations such as *cross-over* and *mutate* on the configuration parameter values, and generates a new set of configuration parameter values, as shown in ③. These configuration parameter values are injected to the performance model to get another execution time, as illustrated in ④. Next, the execution time is passed to the GA again. The step ② to ④ might be repeated for a number of times until the optimum configuration is found.

3.4 Implementation

We implement DAC on top of Spark 1.6 by using R [11] which is a programming language and environment for statistical computing. First, we use R to implement the CG (configuration generator) of the *collecting* component of DAC. CG generates a random value within the value range of each configuration parameter, and writes the values of all the 41 configuration parameters into the configuration file of Spark named `spark-dac.conf`. Subsequently, we call the Spark submitter command, `spark-submit`, through R to submit a program to run with the configuration specified by `spark-dac.conf`. When the execution completes, we collect the execution time of the program, and store it with the configuration parameter values and the size of the input dataset, forming a vector defined by Equation (5). We

Table 1. Experimented applications in this study.

Application	Abbr.	input data size
PageRank	PR	1.2, 1.4, 1.6, 1.8, 2 (million pages)
KMeans	KM	160, 192, 224, 256, 288 (million points)
Bayes	BA	1.2, 1.4, 1.6, 1.8, 2 (million pages)
NWeight	NW	10.5, 11.5, 12.5, 13.5, 14.5 (million edges)
WordCount	WC	80, 100, 120, 140, 160 (GB)
TeraSort	TS	10, 20, 30, 40, 50 (GB)

repeat this procedure a number of times, e.g., 2000, to collect a training set S which is stored in a CSV file. Next, we implement Algorithm 1 in R to construct a performance model which produces a R object to represent the model. Finally, we pass a set of configuration parameter values and the model (R object) to the GA implemented in R to search the configuration for optimal performance of that program. When the optimal configuration is found, we write it back to `spark-dac.conf` to configure Spark for the program with optimized performance.

4 Experimental Methodology

Our experimental platform consists of 6 DELL servers, one serves as the master node and the other five serve as slave nodes. Each server is equipped with 12 Intel(R) Xeon(R) CPU E5-2609 1.90GHz six-core processor and 64GB PC3 memory. There are in total 432 cores and 384 GB memory in the cluster. The OS of each node is SUSE Linux Enterprise Server 12. We use Spark 1.6 as our experimental IMC framework. Although newer versions of Spark have been released, the 1.6 version is a main milestone and very popular in industry.

4.1 Representative Programs

We select representative programs from the Spark version of Hibench which is widely used to evaluate the Spark framework. HiBench has different kinds of real-world programs including machine learning and web search. We choose 6 programs to evaluate DAC, shown in Table 1. These programs represent a sufficiently broad set of typical Spark program behaviors. *KMeans* has good instruction locality but poor data locality while *Bayes* is the opposite. While both performing selective shuffling, the iteration selectivity of *PageRank* is much higher compared to *KMeans*. *NWeight* is an iterative graph-parallel algorithm implemented by Spark GraphX which computes associations between two vertices that are n-hop away. It consumes a lot of memory that it stores the whole graph in memory and iterates over the vertices. Compared to *TeraSort* and *WordCount*, these four programs contain much larger number of iterations. Finally, *WordCount* is CPU-intensive and *TeraSort* is both CPU and memory-intensive. To evaluate DAC, we employ five different sizes of input datasets (Table 1) for each program.

4.2 Configuration Parameters

As discussed earlier, we choose a wide range of Spark configuration parameters that significantly influence performance, including *shuffle behavior*, *data serialization*, *memory management*, *execution behavior*, *networking*, etc. Table 2 shows the 41 parameters in detail.

The last column of Table 2 provides the default values of the parameters which are recommended by the Spark team and can be found at [42]. The third column shows the value range of each configuration parameter. This information is not provided by the Spark team, and therefore we conducted experiments to determine the value range for each parameter. Note that the value ranges of these parameters might be different for different clusters because some ranges depend on cluster hardware configurations such as memory size.

5 Results and Analysis

In this section, we first determine the model parameters, such as the number of training examples (*ntrain*), then the results are presented and analyzed.

5.1 Determining *ntrain*

The amount of training data is the most important model parameter that affects model accuracy and the cost. In our evaluation, *ntrain* is equal to $10 \times k$ because we have 10 different data sizes. The larger *ntrain* generally increases the accuracy of the model constructed while the smaller *ntrain* reduces the cost (e.g., the time needed to collect the training data and train the model). To minimize the modeling cost and achieve high accuracy simultaneously, we need to carefully determine *ntrain* considering the trade-off. Since there is no theoretical guidance to determine *ntrain*, we conduct the following experiments.

We start by training the performance models for a Spark program using 200 Spark configurations, and increase the training set S by 200 each time. Figure 7 quantifies the relationship between accuracy and the number of training examples. For simplicity, we only show the maximum (*Max*), minimum (*Min*), and mean (*Mean*) errors for the models of all the experimented program-input pairs in the figure. The general trend is that, the errors decrease when the number of training examples increase. When *ntrain* reaches 2000, the curves for the three errors become flat thereafter, indicating diminishing return with more training data. Therefore, we choose 2000 training examples as *ntrain* to train a performance model for each Spark program.

5.2 Determining the *lr*, *nt*, and *tc*

To achieve an optimized performance model, we need to determine three other important parameters for the *first-order* HM model: learning rate (lr), the number of trees (nt), and tree complexity (tc). The lr controls the contribution of a sub-model when it is added to the first-order model. nt represents the number of sub-models needed to construct the first-order model. For the same accuracy, decreasing lr will increase the number of sub-models (nt) required to build the first-order model. Tree complexity (tc) specifies the number

Table 2. Description of the 41 Spark configuration parameters.

Configuration Parameters—Description	Range	Default
spark.reducer.maxSizeInFlight—Maximum size of map outputs to fetch simultaneously from each reduce task, in MB.	2–128	48
spark.shuffle.file.buffer—Size of the in-memory buffer for each shuffle file output stream, in KB.	2–128	32
spark.shuffle.sort.bypassMergeThreshold—Avoid merge-sorting data if there is no map-side aggregation.	100–1000	200
spark.speculation.interval—How often Spark will check for tasks to speculate, in millisecond.	10–1000	100
spark.speculation.multiplier—How many times slower a task is than the median to be considered for speculation.	1–5	1.5
spark.speculation.quantile—Percentage of tasks which must be complete before speculation is enabled.	0–1	0.75
spark.broadcast.blockSize—Size of each piece of a block for TorrentBroadcastFactory, in MB.	2–128	4
spark.io.compression.codec—The codec used to compress internal data such as RDD partitions, and so on.	snappy, lzf,lz4	snappy
spark.io.compression.lz4.blockSize—Block size used in LZ4 compression, in KB.	2–128	32
spark.io.compression.snappy.blockSize—Block size used in snappy, in KB.	2–128	32
spark.kryo.referenceTracking—Whether to track references to the same object when serializing data with Kryo	true,false	true
spark.kryoserializer.buffer.max—Maximum allowable size of Kryo serialization buffer, in MB.	8–128	64
spark.kryoserializer.buffer—Initial size of Kryo's serialization buffer, in KB.	2–128	64
spark.driver.cores—Number of cores to use for the driver process.	1–12	1
spark.executor.cores—The number of cores to use on each executor.	1–12	core #
spark.driver.memory—Amount of memory to use for the driver process, in MB.	1024–12288	1024
spark.executor.memory—Amount of memory to use per executor process, in MB.	1024–12288	1024
spark.storage.memoryMapThreshold—Size of a block above which Spark maps when reading a block from disk, in MB.	50–500	2
spark.akka.failure.detector.threshold—Set to a larger value to disable failure detector in Akka.	100–500	300
spark.akka.heartbeat.pauses—Heart beat pause for Akka, in second.	1000–10000	6000
spark.akka.heartbeat.interval—Heart beat interval for Akka, in second.	200–5000	1000
spark.akka.threads—Number of actor threads to use for communication.	1–8	4
spark.network.timeout—Default timeout for all network interactions, in second.	20–500	120
spark.locality.wait—How long to launch a data-local task before giving up, in second.	1–10	3
spark.scheduler.revive.interval—The interval length for the scheduler to revive the worker resource, in second.	2–50	1
spark.task.maxFailures—Number of task failures before giving up on the job.	1–8	4
spark.shuffle.compress—Whether to compress map output files.	true,false	true
spark.shuffle.consolidateFiles—If set to "true", consolidates intermediate files created during a shuffle.	true,false	false
spark.memory.fraction—Fraction of (heap space - 300 MB) used for execution and storage.	0.5–1	0.75
spark.shuffle.spill— Responsible for enabling/disabling spilling.	true,false	true
spark.shuffle.spill.compress—Whether to compress data spilled during shuffles.	true,false	true
spark.speculation—If set to "true", performs speculative execution of tasks.	true,false	false
spark.broadcast.compress—Whether to compress broadcast variables before sending them. Generally a good idea.	true,false	true
spark.rdd.compress—Whether to compress serialized RDD partitions.	true,false	false
spark.serializer—Class to use for serializing objects that are sent over the network or need to be cached in serialized form.	java,kryo	java
spark.memory.storageFraction—Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by *spark.memory.fraction*.	0.5–1	0.5
spark.localExecution.enabled—Enables Spark to run certain jobs on the driver, without sending tasks to the cluster.	true,false	false
spark.default.parallelism—The largest number of partitions in a parent RDD for distributed shuffle operations.	8–50	#
spark.memory.offHeap.enabled—If true, Spark will attempt to use off-heap memory for certain operations.	true,false	false
spark.shuffle.manager—Implementation to use for shuffling data.	sort,hash	sort
spark.memory.offHeap.size—The absolute amount of memory which can be used for off-heap allocation, in MB.	10–1000	0

of nodes in a tree, which also influences the optimal nt. For a given lr, fitting more complex trees leads to fewer sub-models required for minimum error. Since these general rules do not tell us the values of lr, nt, and tc, we again determine the values of them by experiments. We try two values (1 and 5) for tc, five values (0.05,0.01,0.005, 0.001,0.0005) for lr, and many values for nt.

Figure 8 shows the relationship between the nt, lr, and tc for PageRank. As can be seen, when tc is equal to 1, no matter how we vary lr and nt, the minimum error is always equal to or larger than 10%. This implies that the trees with one nodes are too simple to achieve high accuracy. In contrast, the minimum error decreases to 7.6% when tc is equal to 5. For a given lr, the error decreases with the increase of nt and finally converges to the minimum error. However, the convergence speed is slower with a small lr value. As shown in Figure 8 (b), the curve for $lr = 0.05$ converges

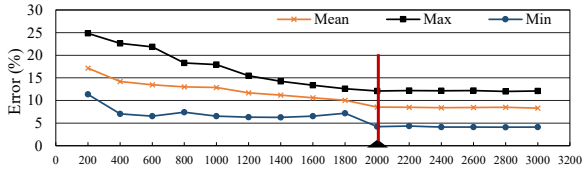

Figure 7. Performance model error as a function of the number of training examples ($ntrain$).

to the minimum error most quickly, and the convergence happens at a point where nt equals 3600. We observe similar curves for other experimented programs. Therefore, we set tc, lr, and nt to 5, 0.05, and 3600, respectively.

5.3 Model Accuracy

With the model parameters determined for the first-order HM model, we evaluate the accuracy of the generated performance predication. If it is the case, we stop the model

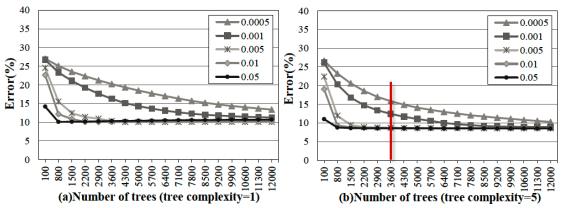

Figure 8. The relationship between number of trees (nt) and errors for models fitted with five learning rates(lr) and two levels of tree complexity (tc) for PageRank. (a) $tc = 1$; (b) $tc = 5$.

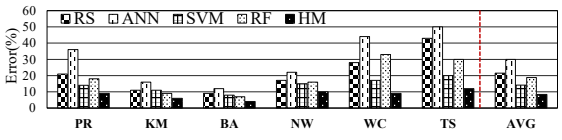

Figure 9. The average prediction errors of models built by RS, ANN, SVM, RF, and HM.

building process, otherwise, a second-order model needs to be further constructed. As described in Section 3.2, this process is recursive. We compare the accuracy of models built by RS, ANN, SVM, RF, and HM. We firstly use each modeling technique to construct 6 performance models for the 6 programs based on the same training data set S. Subsequently, we collect other 500 performance vectors which are different from the 2000 vectors for S to form a testing set T. We plug in the values of configuration parameters and the size of input dataset of each vector in T to the performance model to predict the execution time of a program. Then, equation (2) is used to calculate the error between the predictions and the real measurements. To test each model, we have 500 such errors and we use the average of them to represent the accuracy of each model for each program.

Figure 9 compares the prediction errors of performance models built by RS, ANN, SVM, RF, and HM. We see that the prediction errors of the models built by HM are dramatically lower than those of models constructed by other modeling techniques. For the experimented programs, only the error for TS slightly exceeds 10%. The average error of models built by HM for all applications is only 7.6%. In contrast, the average errors of models built by RS, ANN, SVM, and RF are 22%, 30%, 15%, and 19%, respectively. The results demonstrate the effectiveness of the first-order performance model constructed by HM: taking account of the size of input datasets and high dimensional configuration parameters, our method is accurate enough for searching the optimum configurations for Spark programs. In contrast, all other existing modeling approaches incur large errors.

5.4 Error Distribution

Our model accuracy measures the average error of the testing models. While the average error is a good metric to evaluate a model's accuracy statistically, it might hide large errors

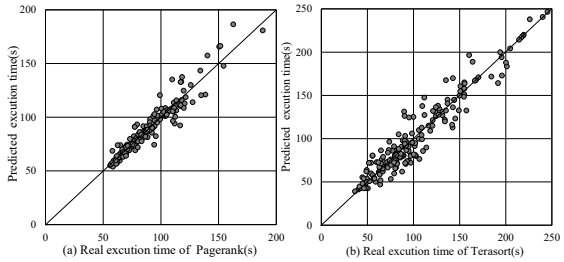

Figure 10. Error distribution illustrating prediction versus real measurement for 200 randomly selected Spark configurations. (a) shows the error distribution for PR. (b) shows the error distribution for TS.

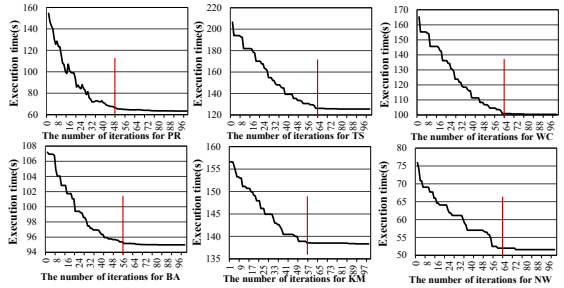

Figure 11. The number of iterations for all the programs *PageRank*(PR), *Bayes*(BA), *WordCount*(WC), *TeraSort*(TS), *KMeans*(KM), and *NWeight*(NW).

for particular predictions due to outliers. To address this issue, we now present the error distribution of our prediction models using scatter plots.

Figure 10 shows two scatter plots produced by 200 real measurements and 200 DAC predictions for programs PR and TS for 200 randomly selected Spark configurations. The X axis represents the real measurements and the Y axis denotes the execution times of the two programs predicted by DAC. This figure clearly shows that the models are fairly accurate across the entire Spark configuration space: all 200 data points for each application are located around the corresponding bisector, indicating that the predictions are close to the real measurements. For other experiments, we also observe similar results. Overall, we find that there are not many outliers in our performance predictions, which is good for optimizing the configurations for Spark.

5.5 Iteration Number of the GA

We employ GA to iteratively search the huge configuration space to find the optimum configuration for a Spark 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., 50 to 70, is typically enough. Moreover, different programs may need different number of iterations. For example, PR, BA, and KM

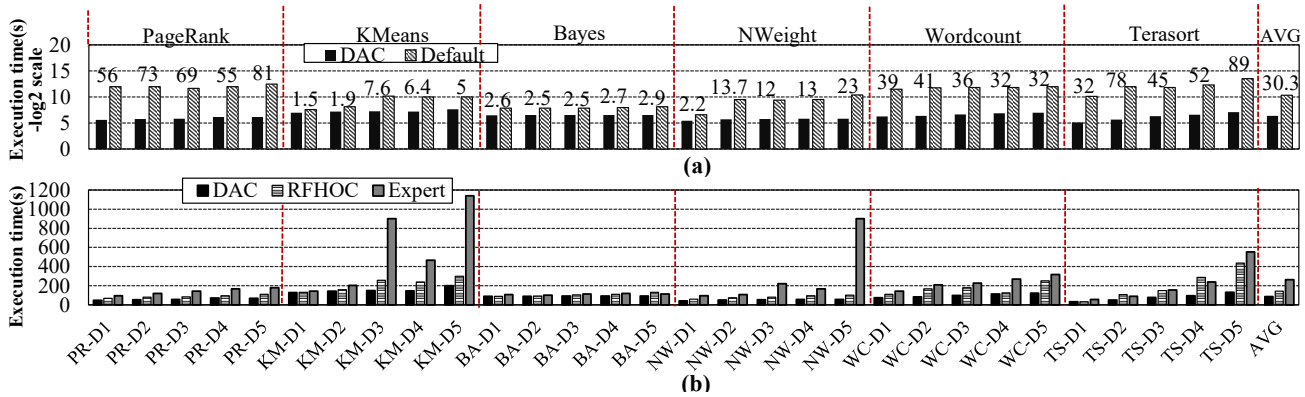

Figure 12. Speedup of DAC over default configurations, RFHOC and the expert approach for all programs *PageRank*(PR), *Bayes*(BA), *WordCount*(WC), *TeraSort*(TS), *KMeans*(KM), and *NWeight*(NW). The D1,..,D5 of each program correspond to the input datasets listed from the left to the right by Table 1 for each program.

need 48, 56, and 57 iterations respectively to find the best optimizations while other programs need 64 iterations. This implies different optimization costs for different programs.

One may think that it is unnecessary to employ performance models but instead to use real executions to search the optimal configuration for a Spark program by GA because it converges at most 64 iterations. Unfortunately, this is impractical for programs with relatively large input datasets because the real execution may take long time to finish. For example, TS with 50 GB of input dataset and a randomly generated configuration takes 16 minutes to complete its execution on our experimental cluster. In contrast, our performance model only takes several milliseconds to predict the execution time of TS with the same input dataset and configuration.

5.6 Speedup

Figure 12 shows the speedups of the 30 program-input pairs with DAC over default configurations and the RFHOC approach [4] . We reimplement RFHOC, the state-of-the-art technique for Hadoop in the context of Spark for a fair comparison. As shown in Figure 12(a), DAC dramatically improves the performance of the 30 program-input pairs with default configurations by a factor of 30.4× on average and up to 89×. This indicates that the default configurations of Spark programs make them far from optimal. The reason is that the default configurations do not consider the characteristics of programs, especially when the size of input datasets is very large. For example, the default value of `spark.executor.memory` is 1024 MB. This may work well for programs with small input datasets such as WC with 5 GB of input dataset. However, when the input dataset becomes large(e.g., 160 GB), this default value causes a lot of *out-of-memory* failures, forcing Spark to rerun some tasks many times and in turn take a long time to complete the program execution.

As shown in Figure 12(b), DAC also significantly outperforms RFHOC. The speedup of DAC over RFHOC is 1.6× on average and up to 3.3×. The reason is that the RFHOC does not consider the significant impact of the size of input datasets on finding the optimal configurations for IMC programs. Moreover, RFHOC employs random forest (RF) to build the performance models, which we show the models are not accurate in Section 2.2.2.

We further compare the performance tuned by DAC against that tuned by an expert. To perform this study, we manually tune the performance of the experimented Spark programs according to the tuning guide released by the Spark and Cloudera team [16, 43], which we call expert approach. The speedup of DAC over the expert approach is 2.99× on average and up to 16× as shown in Figure 12 (b). Therefore, the manual tuning indeed improves the default configuration but is still less effective than DAC. We believe it is due to two key limitations of a manual expert approach. First, the recommendations [16, 43] can not adapt to different programs. For example, a recommendation suggests 2-3 tasks per CPU cores in a cluster but this is not always true, especially for CPU-intensive workloads like WC, since multiple tasks share the same core will increase the contention for shared resources. Second, some recommendations are qualitative rather than quantitative. For example, if the Old Generation memory of JVM, that contains the objects that are long lived and survived after many rounds of minor garbage collection [34], is close to be full, the recommendation suggests lowering `spark.memory.fraction`, but does not suggest how much.

We also report that the geometric mean speedups of DAC over configurations by default, expert, and RFHOC are 15.4×, 2.3×, and 1.5×, respectively.

5.7 Overhead

We now report the overhead of DAC including the times used to collect training data, train performance models, and search optimum configurations. Table 3 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, — 70.3 hours on average and up to 92 hours. While it 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 DAC are the iterative applications which usually *repeatedly 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. Therefore, the additional cover per run is very low.

5.8 Detailed Analysis: KM & TS

Finally, We provide a detailed analysis for KM and TS to gain deeper insights. Figure 13 shows the execution times of the five stages of KM. From the results, we see a number of interesting observations. First, DAC and RFHOC both significantly reduce the execution time of KM. The performance gain is larger with larger input dataset. Second, DAC does not show significant improvement over RFHOC when the size of input datasets is small, as shown Figure 13 (a). However, when the input dataset is getting large, DAC outperforms RFHOC significantly, see Figure 13 (b) and (c). This is because DAC takes the effects of the size of input datasets into account when optimizing configurations while RFHOC does not. Thus, DAC could choose the proper configuration parameters with different data sizes. Third, both DAC and RFHOC significantly reduce the execution time of StageC that iteratively performs data aggregation and collection but DAC achieves more reduction than RFHOC, which is the primary reason why DAC significantly improves the performance of KM. In addition, DAC also reduces more time for the StageA and StageD than RFHOC but not that much for StageC. This is because the execution time of StageA and StageD is much shorter than that of StageC. Actually, we observe the similar ratios of reduced times to the total execution times of StageA, StageD, and StageC, respectively. This indicates that adjusting the configurations according to the size of input datasets significantly helps one to reduce the execution time of a program.

Figure 13 (d) and (e) shows that why DAC can reduce the execution time of KM with default configuration and RFHOC. As can be seen, DAC significantly reduces the garbage collection time no matter when KM runs with default configuration or with the RFHOC produced configuration. The larger input dataset, the more time can be saved by DAC.

Figure 14 shows the execution time comparison of Stage2 of TS which represents another application pattern: one stage

Table 3. Time Cost.

Workload	Collecting(h)	Modeling(s)	Searching(m)
KMeans	92	10	7
Bayes	60	11	9
PageRank	67	9	10
TeraSort	68	11	8
WordCount	82	12	7
NWeight	53	12	9

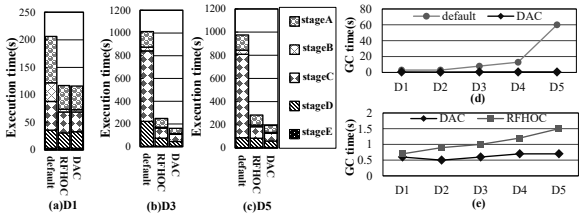

Figure 13. Performance of stages for KM with default configuration, RFHOC, and DAC. StageA - reading input data; StageB - taking samples; StageC - iteratively aggregating and collecting data; StageD - collecting results; StageE - summarizing results. D1,...,D5 represent the 5 input datasets from the smallest to the largest. GC - garbage collection.

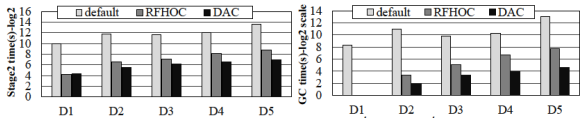

Figure 14. Performance of Stage2 for TS with default configuration, RFHOC, and DAC. GC — garbage collection.

dominates the execution time of the whole application. TS only contains two stages: Stage1 and Stage2. Stage1 takes 10% of the whole execution time while Stage2 takes 90% of that. We can see that, DAC also significantly outperforms RFHOC and the default configuration for TS (the left of Figure 14). Similar to KM, the time reduction for the garbage collection is the main reason for the performance improvement of TS. In addition, when the size of input dataset increases, the garbage collection time of applications with DAC increases more slowly than that with RFHOC and default configurations.

6 Related Work

In this section, we describe the configuration auto-tuning studies related to Spark. The Spark official web site provides a performance tuning guide for Spark programs [43]. Although this guide is helpful, it is a manual approach which requires developers have a deep understanding about Spark. In contrast, DAC is an automated tool. Juwei *et al.* [38] compared the MapReduce with Spark and provided interesting insights. However, this work does not propose a solution to improve the performance of Spark programs while our DAC does. Kay *et al.* [33] propose a block analysis approach to systematically analyze the bottleneck of Spark programs using two SQL workloads. While this work provides a good performance analysis tool and reveal some bottlenecks of the SQL-like programs, it does not optimize general Spark programs such as those implementing machine learning algorithms. In contrast, DAC provides an approach to automatically optimize the performance of general Spark programs.

Yao *et al.* [61] tried to improve the performance of Spark programs by adaptively tuning the serialization techniques.

DAC is different from this work since it improves performance by automatically adjusting a large set of Spark configuration parameters, instead of only the sterilization aspect. Guolu *et al.* [49] propose to tune Spark configurations by using regression tree, which is close to our work. However, simply using regression tree model can not tackle the high dimensional configuration issue of Spark. Their work only uses 16 configuration parameters and improves performance only by 36%. Tatsuhiro *et al.* [6] carefully characterize the memory, network, JVM, and garbage collection behavior and in turn optimize the performance of Spark workloads. However, this work only focuses on TPC-H workloads while our DAC focuses on general-purpose Spark applications.

Another class of related works attempt to optimize the configurations of MapReduce/Hadoop applications. Herodotou *et al.* [12–14] propose to build analytical performance models first and then leverage genetic algorithm to search the optimum configurations for Hadoop workloads. Adem *et al.* [10] 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.* [4] propose a random forest based approach to automatically tune the configurations of Hadoop programs. These studies work well for Hadoop programs but not Spark. We have demonstrated that the optimal configurations of Spark applications are more sensitive to input data size compared to Hadoop programs, implying that the configuration auto-tuning techniques for Hadoop workloads can not be easily extended to Spark applications.

Moreover, there are many studies related to optimize MapReduce/Hadoop applications from Hadoop runtime optimization [23, 28, 29, 31, 39, 50] and job scheduling [1, 24] aspects. Our work does not optimize programs from these aspects but focuses on tuning configurations, which can be complementary to these approaches.

System configuration may also cause errors and existing studies focus on eliminating or reducing configuration errors. Xu *et al.* [55] present a comprehensive survey on how to handle the system configuration errors. Yin *et al.* [58] perform a characterization of many configuration errors in commercial systems and they reveal that the parameter-related configuration is very important. Xu *et al.* [54] argue that software developers should take an active role in handling misconfigurations as end users do. They also provide a tool named SPEX to automatically infer configuration requirements from software source code. Zhang *et al.* [60] develop a framework and tool called EnCore to automatically detect software misconfigurations. Huang *et al.* [15] propose a framework named ConfValley to easily, systematically, and efficiently validate the configuration of cloud-scale systems. Finally, Xu *et al.* [53] investigate thousands of customers of one commercial storage system and two open-source systems about system configurations. They reveal a very interesting insight: usually a number of configuration parameters are not mandatory and can be safely removed. Unlike these studies focusing on configuration errors, DAC focuses on improving performance via configuration tuning.

Resource management related configuration optimization of cloud computing platforms is yet another class of related work. For instance, Paragon [8] is a machine learning based approach to schedule the tasks in heterogeneous datacenters. Later, Quasar [9] is proposed to manage a cluster to achieve resource efficiency and to guarantee QoS. Liu *et al.* [27] employ a generic algorithm based parameter tuning to improve the performance of interactive mobile applications in cloud. Ansel *et al.* [2] propose to build an independent framework to auto-tune performance for different programs. DAC differs from these studies that DAC focuses on tuning configuration parameters.

7 Conclusion

This paper proposes DAC, a datasize-aware auto-tuning approach to efficiently identify the high dimensional configuration for a given IMC program to achieve optimal performance on a given cluster. DAC is a significant advance over the state-of-the-art because it can take the size of input dataset and 41 configuration parameters as the parameters of the performance model for a given IMC program, — unprecedented in previous work. To evaluate DAC, we use six typical Spark programs, each with five different input dataset sizes. The evaluation results show that DAC improves the performance of six typical Spark programs, each with five different input dataset sizes compared to default configurations by a factor of 30.4× on average and up to 89×. We demonstrate that DAC significantly outperforms RFHOC by a geometric mean speedup of 1.5×, and even configurations determined by an expert by 2.3× (geometric mean speedup).

Acknowledgments

We thank the reviewers for their thoughtful comments and suggestions. This work is supported by the National Key Research and Development Program of China under no. 2016YFB1000204; NSFC under grants no. 61672511, 61702495; NSF-CCF-1657333, NSF-CCF-1717754, NSF-CNS-1717984, and NSF-CCF-1750656. Outstanding technical talent program of CAS. Additional support is provided by the major scientific and technological project of Guangdong province (2014B010115003), Shenzhen Technology Research Project (JSGG20160510154–636747), and Key technique research on Haiyun Data System of NICT, CAS with XDA06010500. Zhendong Bei is the corresponding author. Contact: Zhibin Yu (zb.yu@siat.ac.cn), Zhendong Bei (zd.bei@siat.ac.cn).

References

[1] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. 2014. ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *Proceedings of USENIX Annual Technical Conference (ATC) (ATC'14)*. USENIX Association, Philadelphia, PA, 1–12.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT) (PACT'14)*. ACM Press, Edmonton, Canada, 303–316.

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1383–1394.

[4] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2016. RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (June 2016), 1470–1483. https://doi.org/10.1109/TPDS.2015.2449299

[5] Dazhao Cheng, Jia Rao, Yanfei Guo, and Xiaobo Zhou. 2014. Improving MapReduce Performance in Heterogeneous Environments with Adaptive Task Tuning. In *Proceedings of the 15th International Middleware Conference (Middleware) (Middleware'14)*. USENIX Association, Bordeaux, France, 97–108.

[6] Tatsuhiro Chiba and Tamiya Onodera. 2015. *Workload Characterization and Optimization of TPC-H Queries on Apache Spark*. Technical Report. IBM Research - Tokyo, IBM Japan, Ltd.

[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI) (OSDI'12)*. USENIX Association, San Francisco, CA, 137–150.

[8] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS'13)*. ACM Press, Houston, TX, 77–88.

[9] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS'14)*. ACM Press, Salt Lake City, UT, 1–12.

[10] Adem Efe Gencer, David Bindel, Emin Gun Sirer, and Robbert van Renesse. 2015. Configuring Distributed Computations Using Response Surfaces. In *Proceedings of the annual ACM/IFIP/USENIX Middleware conference (Middleware) (Middleware'15)*. USENIX Association, Vancouver, Canada, 235–246.

[11] Robert Gentleman and Ross Ihaka. 2016. The R Project for Statistical Computing. (Sept. 2016). Retrieved Januray 20, 2018 from https://www.r-project.org/

[12] Herodotos Herodotou. 2011. *Hadoop Performance Models*. Technical Report CS-2011-05. Duke University, Durham, NC.

[13] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, What-If Analysis, and Cost-Based Optimization of MapReduce programs. *Journal of VLDB Endowment* 4, 11 (Jan. 2011), 1111–1122. https://doi.org/10.1.1.222.8262

[14] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of the Biennial International Conference on Innovative Data Systems Research (CIDR'11)*. CIDRDB, 261–272.

[15] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys) (EuroSys'15)*. USENIX Association, Bordeaux, France, 1–16.

[16] Cloudera Inc. 2016. Tuning Spark Applications. (June 2016). Retrieved Januray 20, 2018 from https://www.cloudera.com/documentation/enterpr-ise/5-4-x/topics/admin_spark_tuning.html

[17] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs form Sequential Building Blocks. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys) (EuroSys'07)*. USENIX Association, Lisbon, Portugal, 59–72.

[18] Manoj Kumar, Mohammad Husian, Naveen Upreti, and Deepti Gupta. 2010. Genetic algorithm: Review and Application. *International Journal of Information Technology and Knowledge Management* 2, 2 (Jan. 2010), 451–454.

[19] Palden Lama and Xiaobo Zhou. 2012. AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud. In *Proceedings of the 9th ACM International Conference on Autonomic Computing (ICAC) (ICAC'12)*. ACM Press, San Jose, CA, 63–72.

[20] Jacek Laskowski. 2016. Mastering Apache Spark. (Jan. 2016). Retrieved Januray 20, 2018 from https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-dagscheduler-stages.html

[21] Benjamin C. Lee and David Brooks. 2010. Applied Inference: Case Studies in Micro-architectural Design. *ACM Transactions on Architecture and Code Optimization (TACO)* 7, 2 (Sept. 2010), 8:1–8:35.

[22] Roger J Lewis. 2000. An introduction to classification and regression tree (CART) analysis. In *Proceedings of Annual Meeting of the Society for Academic Emergency Medicine*. San Francisco, CA, 1–14.

[23] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC) (SoCC'14)*. ACM Press, Seattle, WA, 1–15.

[24] Shen Li, Shaohan Hu, Shiguang Wang, Lu Su, Tarek Abdelzaher, Indranil Gupta, and Richard Pace. 2014. Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS) (ICDCS'14)*. IEEE, Madrid, Spain, 93–103.

[25] Guangdeng Liao, Kushal Datta, and Theodore L Willke. 2013. Gunther: Search-Based Auto-Tuning of MapReduce. In *Proceedings of Euro-Par 2013 Parallel Processing (EuroPar'13)*. Springer, Berlin, Heidelberg, 406–419.

[26] Luo Lie. 2010. Heuristic Artificial Intelligent Algorithm for Genetic Algorithm. *Key Engineering Materials* 439 (May 2010), 516–521.

[27] Weiqing Liu, Jiannong Cao, Lei Yang, Lin Xu, Xuanjia Qiu, and Jing Li. 2017. AppBooster: Boosting the Performance of Interactive Mobile Applications with Computation Offloading and Parameter Tuning. *IEEE Transactions on Parallel and Distributed Systems* 28, 6 (June 2017), 1593–1606.

[28] Zhaolei Liu and TS Eugene Ng. 2017. Leaky Buffer: A Novel Abstraction for Relieving Memory Pressure from Cluster Data Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (March 2017), 128–140. https://doi.org/10.1109/TPDS.2016.2546909

[29] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *Proceedings of the 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland, 1–6.

[30] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apach Spark. *The Journal of Machine Learning Research* 17, 1 (Jan.

2016), 1–7.

[31] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI) (OSDI'16)*. USENIX Association, Savannah, GA, 349–365.

[32] Andrew Or and Josh Rosen. 2016. Unified Memory Management in Spark 1.6. (Jan. 2016). Retrieved Januray 20, 2018 from https://issues.apache.org/jira/secure/attachment/12765646/unified-memory-management-spark-10000.pdf

[33] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12nd USENIX Symposium on Networked Systems Design and Implementation (NSDI) (NSDI'15)*. USENIX Association, Oakland, CA, 293–307.

[34] Pankaj. 2017. Java (JVM) Memory Model - Memory Management in Java. (March 2017). Retrieved Januray 20, 2018 from http://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java

[35] Simone Pellegrini, Radu Prodan, and Thomas Fahringer. 2012. Tuning MPI Runtime Parameter Setting for High Performance Computing. In *Proceedings of IEEE International Conference on Cluster Computing Workshops*. IEEE Computer Society, Washington, DC, 213–221.

[36] Zujie Ren, Xianghua Xu, Jian Wan, Weisong Shi, and Min Zhou. 2012. Workload Characterization on a Production Hadoop Cluster: A Case Study on Taobao. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC) (IISWC'12)*. IEEE Computer Society, San Diego, CA, 1–11.

[37] Anooshiravan Saboori, Guofei Jiang, and Haifeng Chen. 2008. Autotuning Configurations in Distributed Systems for Performance Improvements using Evolutionary Strategies. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS) (ICDCS'08)*. IEEE Computer Society, Beijing, China, 769–776.

[38] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Ozcan. 2015. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. In *Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB Endowment), Vol.8, No.13 (VLDB'15)*, Vol. 8. Hawai'i, USA, 2110–2121.

[39] Xueyuan Su, Garret Swart, Brian Goetz, Brian Oliver, and Paul Sandoz. 2014. Changing engines in midstream: A java stream computational model for big data processing. *Proceedings of the VLDB Endowment* 7, 13 (Sept. 2014), 1343–1354.

[40] Apache HBase Team. 2016. Apache HBase. (June 2016). Retrieved Januray 20, 2018 from http://hadoop.apache.org/hbase/

[41] Aparch Spark Team. 2016. Aparch Spark. (March 2016). Retrieved Januray 20, 2018 from http://spark.apache.org/

[42] Aparch Spark Team. 2016. Spark Configuration. (May 2016). Retrieved Januray 20, 2018 from http://spark.apache.org/docs/latest/configuration.html

[43] Aparch Spark Team. 2016. Tuning Spark. (June 2016). Retrieved Januray 20, 2018 from http://spark.apache.org/docs/latest/tuning.html

[44] Spark Streaming Team. 2016. Spark Streaming. (March 2016). Retrieved Januray 20, 2018 from http://spark.apache.org/streaming/

[45] White Tom. 2012. *Hadoop: The definitive guide*. O'Reilly Media, Inc.

[46] Virginia Torczon and Michael W Trosset. [n. d.]. From Evolutionary Operation to Parallel Direct Search: Pattern Search Algorithms for Numerical Optimization. *Computing Science and Statistics* 29 ([n. d.]).

[47] Inc. TypeSafe. 2015. Apache Spark Survey from Typesafe. (Jan. 2015). Retrieved Januray 20, 2018 from https://dzone.com/articles/apache-spark-survey-typesafe-0

[48] Md. Wasi ur Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. Panda. 2016. MR-Advisor: A Comprehensive Tuning Tool for Advising HPC Users to Accelerate MapReduce Applications on Supercomputers. In *Proceedings of 2016 IEEE 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'16)*. IEEE Computer Society, Los Angeles, CA, 198–205.

[49] Guolu Wang, Jungang Xu, and Ben He. 2016. A Novel Method for Tuning Configuration parameters of Spark Based on Machine Learning. In *Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications (HPCC) (HPCC'16)*. IEEE Computer Society, Sydney, Australia, 586–593.

[50] Jingjing Wang and Magdalena Balazinska. 2016. Toward elastic memory management for cloud data analytics. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM Press, San Francisco, CA, 1–7.

[51] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: A Resilient Distributed Graph System on Spark. In *Proceedings of the First International Workshop on Graph Data Management Experimence and System*. 1–5.

[52] Wen Xiong, Zhibin Yu, Lieven Eeckhout, Zhengdong Bei, Fan Zhang, and Chengzhong Xu. 2015. SZTS: A Novel Big Data Transportation System Benchmark Suite. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP) (ICPP'15)*. IEEE, Beijin, China, 819–828.

[53] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have Given Me Too Many Knobs. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM Press, Bergamo, Italy, 307–319.

[54] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (SOSP'13)*. USENIX Association, Farmington, Pennsylvania, 244–259.

[55] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *Comput. Surveys* 47, 4 (July 2015), 1–41.

[56] Tao Ye and Shivkumar Kalyanaraman. [n. d.]. A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 ([n. d.]).

[57] Nezih Yigitbasi, Theodore L. Willke, Guangdeng Liao, and Dick H. J. Epema. 2013. Towards Machine Learning-Based Auto-tuning of MapReduce. In *Proceedings of the 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS) (MASCOTS'13)*. IEEE Computer Society, San Francisco, CA, 11–20.

[58] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (SOSP'11)*. USENIX Association, Cascais, Portugal, 159–172.

[59] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud) (HotCloud'10)*. USENIX Association, Boston, MA, 1–8.

[60] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS'14)*. ACM Press, Salt Lake City, UT, 687–700.

[61] Yao Zhao, Fei Hu, and Haopeng Chen. 2016. An Adaptive Tuning Strategy on Spark Based on In-memory Computation Characteristics. In *Proceedings of the 18th International Conference on Advanced Communication Technology (ICACT) (ICACT'16)*. PyeongChang, Korea (South), 484–488.