This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration

Zhendong Bei, Zhibin Yu*, *Member, IEEE,* Huiling Zhang, Wen Xiong, Chengzhong Xu, *Senior Member, IEEE,* Lieven Eeckhout[†], *Member, IEEE,* and Shenzhong Feng

Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences
[†]Ghent University, Belgium

Abstract—Hadoop is a widely-used implementation framework of the MapReduce programming model for large-scale data processing. Hadoop performance however is significantly affected by the settings of the Hadoop configuration parameters. Unfortunately, manually tuning these parameters is very time-consuming, if at all practical. This paper proposes an approach, called RFHOC, to automatically tune the Hadoop configuration parameters for optimized performance for a given application running on a given cluster. RFHOC constructs two ensembles of performance models using a random-forest approach for the map and reduce stage respectively. Leveraging these models, RFHOC employs a genetic algorithm to automatically search the Hadoop configuration space. The evaluation of RFHOC using five typical Hadoop programs, each with five different input data sets, shows that it achieves a performance speedup by a factor of 2.11× on average and up to 7.4× over the recently proposed cost-based optimization (CBO) approach. In addition, RFHOC's performance benefit increases with input data set size.

Index Terms—performance tuning, MapReduce/Hadoop, system configuration, random forest, genetic algorithm

1 Introduction

MapReduce is a widely used programming model for processing and generating vast data sets on large-scale compute clusters. Hadoop is the most popular open-source MapReduce framework, using which a broad set of applications have been developed, including web indexing [1], machine learning [2], log file analysis [3], financial analysis [4] and bioinformatics processing [5]. A typical characteristic of these applications is that they run repeatedly with different input data sets.

The Hadoop framework has up to 190 configuration parameters, and overall performance is highly sensitive to the settings of these parameters. Because the Hadoop configuration for optimum performance is application-specific [6], applying the default or a single set of configuration settings optimized for a certain application to a wide range of applications leads to suboptimal performance. Unfortunately, manually tuning so many parameters without in-depth knowledge of the Hadoop internal system and the given application is extremely tedious and time-consuming, and may even cause serious performance degradation [6]–[8]. By consequence, automatically tuning the configuration parameters for a given long-running Hadoop application on a given cluster to achieve optimized performance is highly desirable.

A naive approach to find the optimum Hadoop configuration is to try every combination of configuration parameter values and choose the best one. Unfortunately, this is unrealistic because of the huge number of Hadoop configuration parameter combinations. To make things even worse, every run of a Hadoop application with a large input data set takes a considerable amount of time, leading to impractically long times if one were to explore the huge Hadoop configuration parameter optimization space exhaustively.

Performance models can predict the performance of an application with a given configuration much faster than an approach that requires executing the application. For example, Herodotou et al. [6]–[8] propose fine-grained analytical models to predict the execution time of each phase; putting together per-phase predictions then yields an estimate for the total execution time. Lama et al. [9] build a Support Vector Machines (SVM) based model to predict the performance of Hadoop jobs. However, these analytical performance models are based on oversimplified assumptions (see Section 2.3 for details), which limits overall performance.

In this paper, we propose a novel approach based on random-forest learning, which we call RFHOC, to predict the performance of an application on a given cluster with a given Hadoop configuration. Strictly speaking, random-forest is not a machine learning algorithm; instead, it is a robust ensemble model that combines the advantages of statistical reasoning and machine learning

• *Z. Bei, Z. Yu*, H. Zhang, W. Xiong, C. Xu and S. Feng are with the Center for Cloud Computing, Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, 518055, China.*
**corresponding author*
E-mail:{zd.bei,zb.yu,hl.zhang,wen.xiong,cz.xu,sz.feng}@siat.ac.cn.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

approaches [10]. We consider a number of observations from a real Hadoop system to train an ensemble model for each phase via random forest learning. The model takes Hadoop configurations as input and outputs a performance prediction. In a subsequent step, we then use the performance prediction models for each phase as part of a genetic algorithm to search for the optimum Hadoop configuration for the application of interest.

RFHOC has several advantages over existing analytical models and machine learning based approaches. First, RFHOC does not make any assumptions on the cost (execution time) of per-byte or per-record processing and the relationship between configuration parameters. Previously proposed analytical models typically assume the execution time of per-byte or per-record processing to be constant, even as Hadoop configurations change, as is the case for [8]; statistical models such as linear regression algorithms typically assume that the relationship between configuration parameters is linear. RFHOC on the other hand does not make these simplifying assumptions and recognizes that Hadoop configuration parameters interact with each other in complex non-linear ways. Second, random-forest learning makes predictions based on a set of regression or classification trees, rather than a single tree, which makes the performance prediction not only accurate, but also stable and robust when deployed on previously unseen data sets.

In particular, we make the following contributions in this paper:

- We employ random-forest learning to accurately predict the performance of MapReduce/Hadoop programs on a given cluster as a function of the Hadoop framework parameters.
- We leverage the random-forest based model and a genetic algorithm to automatically search for the optimal Hadoop framework configuration.
- We evaluate the proposed approach, called RFHOC, using five representative Hadoop workloads, each with five different input data sets. The results show that RFHOC achieves a performance speedup by a factor of 2.11× on average and up to 7.4× compared to the previously proposed cost-based optimization (CBO) approach [8].

The rest of the paper is organized as follows. Section 2 describes the motivation and background. Section 3 depicts the architecture and design of RFHOC. Section 4 presents the experimental setup, after which we discuss the results and provide detailed analysis in Section 5. Section 6 presents related work. Finally, we conclude in Section 7.

2 Motivation and Background

We first describe the MapReduce/Hadoop programming model, and we provide motivation and background for using random-forest learning to optimize performance.

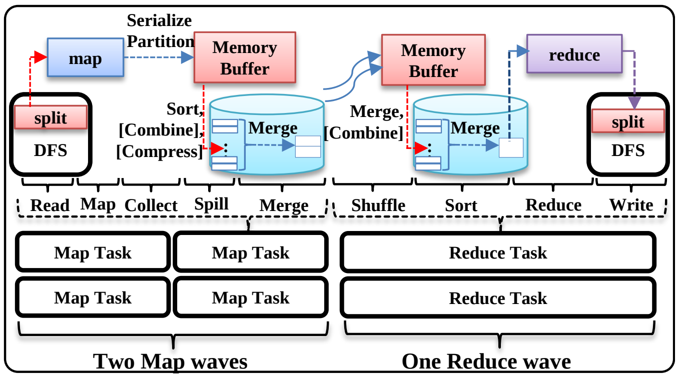

Fig. 1: An example MapReduce workflow.

2.1 MapReduce and Hadoop

MapReduce computation takes a set of input key/value pairs and produces a set of output key/value pairs. Hadoop is an open-source implementation framework for MapReduce. Users only need to write *map* and *reduce* functions and the rest is handled by the Hadoop framework. The execution of a Hadoop program can be divided into *map* and *reduce* stages. Large data sets are split into small blocks by the Hadoop Distributed File System (HDFS). In the *map* stage, each *map* task reads a small block and processes it. When *map* tasks complete, the outputs—also known as intermediate files—are copied to the *reduce* nodes. At the *reduce* stage, *reduce* tasks fetch the key/value pairs from the output files of the *map* stage, which they then sort, merge, and process. The *map* stage itself can be further divided into *read*, *map*, *collect*, *spill*, and *merge* phases. Similarly, the *reduce* stage can be divided into *shuffle*, *sort*, *reduce*, and *write* phases. Each phase consists of several what we call elementary 'operations' such as *mapping*, *merging*, etc., which is typically done on per-record or per-byte units.

Since the total number of tasks processed each time by a given cluster (TTC) is fixed, a Hadoop application needs several execution rounds to complete when the number of tasks is larger than the TTC. An execution round is typically named an execution *wave* by MapReduce/Hadoop programmers. For example, if there are four map tasks (determined by the amount of data) in a Hadoop job but a given cluster only has two map task slots, we need two *waves* to completely execute the four map tasks, as illustrated in Figure 1.

A Hadoop job can be represented by tuple $j = \langle p,d,r,c \rangle$, with p the Hadoop program, d the input data, r the cluster resource, and c the Hadoop configuration. The performance of job j depends on p, d, r and c. The goal in this paper is to optimize performance of repeatedly long-running Hadoop programs with different input data set sizes in a given compute cluster. We therefore assume the program p and the cluster resource r to be fixed. In other words, we optimize Hadoop's configuration c on cluster r for a given program p across various data sets d. We assume though that data sets differ only in size, i.e., other data properties such as compression ratio and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

```
Input: training set S, Inducer F, integer ntree (# of bootstrap samples)
1.  For i = 1 to ntree{
2.        $S'$ = bootstrap sample from S (i.i.d. sample with replacement).
3.        $C_i = F(S')$
4.  }
5.  $C^*(x) = \arg_{y \in Y} \sum_{i=1}^{ntree} C_i(x)/ntree$
Output: Aggregation $C^*$.
```

Fig. 2: The random forest algorithm.

record size are assumed to be constant, which we believe is a reasonable assumption in practice.

2.2 Random Forest

Random forest is an ensemble model that can be used for both classification and regression. It operates by constructing a multitude of decision trees at training time; prediction then combines the outputs by the individual trees to arrive at the final output (e.g., majority voting for classification, and average for regression). A key feature of random forests is that they correct for decision trees' tendency to overfit to their training data.

Figure 2 illustrates the algorithm for building a random forest model at training time. The training set S is a set of observations (there are n observations in total), consisting of a response y (the execution time of a Hadoop phase) for a given set of variables (the Hadoop configuration parameters). The m observations in S are assumed to be independently and identically distributed (i.i.d.). A bootstrap sample is generated by uniformly sampling m' instances from the training set S with replacement [11].

According to Figure 2, the random forest algorithm operates as follows:

1) Draw *ntree* bootstrap samples from the training data set *S* and store them in *S′*.
2) For each of the *ntree* bootstrap samples, grow an unpruned classification or regression tree. At each node, randomly sample *mtry* of the predictor variables and choose the best split among those variables [12]. This procedure is represented by the inducer *F* and the result is stored in C_i. There will be *ntree* trees at the end of this step.
3) Predict new data by aggregating the predictions of the *ntree* trees (i.e., majority votes for classification, and average for regression).

Random forest learning is an extension of a bagging algorithm [13], which chooses the best split among all predictor variables at each node when growing an unpruned tree for each bootstrap sample. Although choosing predictors randomly is somewhat counterintuitive, it turns out to perform very well compared to many other classifiers, including discriminant analysis, Support Vector Machines (SVM) and neural networks [12]. Moreover, it is robust against overfitting [12] and it does not make any assumptions about the predictor variables.

2.3 Prior Work Limitations

To find the Hadoop configuration that achieves optimized performance for a Hadoop program running on a given cluster, it is vitally important to create accurate performance models which take the Hadoop configurations as inputs. There are two prior studies along this line of thought. One is the analytical models built by Herodotou et al. [6]–[8]; the other is the Support Vector Machines (SVM) based models created by Lama et al. [9].

2.3.1 Analytical Models

Although the analytical performance models in [6]–[8] are fine–grained at the level of MapReduce phases, they assume the execution time per operation to depend on the cluster resources only. In addition, they further assume it to be constant across different Hadoop configurations. To verify whether the above assumption holds true, we take the same 10 Hadoop configuration parameters from [8] and we assign random values to them, forming six different Hadoop configurations, see Table 1. By running *teraSort* (one of the benchmarks in this work) on the six configurations, we observe how the execution time of an operation varies across configurations.

Figure 3 shows that the execution time of an operation varies dramatically when the Hadoop configuration changes. For example, the *CPU execution time for merging, per-record (CETMPR)* operation varies from 0.5 to 512 ms across the six experimented configurations. This variation may lead to substantial modeling errors when neglected during performance model building. Furthermore, the *CETMPR* operation with *io.sort.factor* set to 31 (config #1) is lower than when set to 75 (config #3) but higher than when set to 40 (config #6). This indicates that the relationship between *CETMPR* and *io.sort.factor* is non-monotonic and non-linear. Moreover, this also implies that *io.sort.factor* strongly interacts with other configuration parameters. We therefore believe that it is difficult to accurately predict the performance of Hadoop applications by using simple analytical models because these models need to necessarily simplify the complex relationships between configurations.

2.3.2 Support Vector Machines

Lama et al. [9] employ Support Vector Machines (SVM) to build a model called AROMA to predict the performance of Hadoop applications with different configurations. AROMA is based on the observation that jobs with similar resource consumption characteristics exhibit similar performance behavior across Hadoop configurations. This observation might hold true if the performance models were constructed for very fine–grained objects. However, Lama et al. use the amount of CPU, I/O, and network consumed at the

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

TABLE 1: Experimented configuration parameters.

Configuration Parameter	Config 1	Config 2	Config 3	Config 4	Config 5	Config 6
io.sort.factor	31	61	75	14	50	40
mapred.job.shuffle.merge.percent	0.29	0.24	0.61	0.28	0.3	0.89
mapred.output.compress	false	false	true	true	false	true
mapred.inmem.merge.threshold	825	720	717	268	969	469
mapred.reduce.tasks	1	97	79	1	66	2
io.sort.spill.percent	0.5	0.89	0.57	0.72	0.7	0.78
mapred.job.shuffle.input.buffer.percent	0.22	0.26	0.89	0.36	0.23	0.59
io.sort.record.percent	0.01	0.12	0.03	0.27	0.11	0.32
io.sort.mb	96	120	108	131	91	107
mapred.compress.map.output	false	false	true	true	false	true

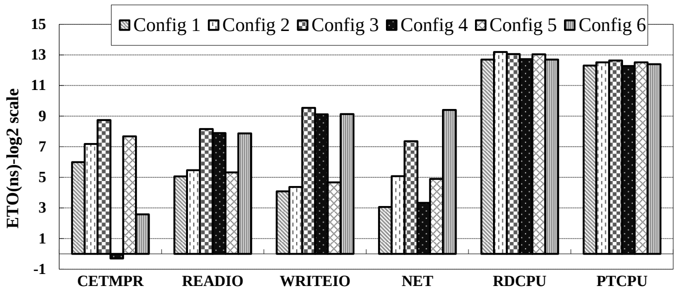

Fig. 3: The execution time of operations (ETO) (ns) of MapReduce phases under different Hadoop configurations. The configurations are in Table 1. CETMPR: CPU execution time for merging, per record; READIO: I/O execution time for reading from local disk, per byte; WRITEIO: I/O execution time for writing to local disk, per byte; NET: network transfer time per byte; RDCPU: CPU execution time of executing the reducer per record; PTCPU: CPU execution of time for partitioning, per record. The execution time of an operation is calculated by dividing the total execution time of a phase with the total number of operations, measured using *BTrace* [14].

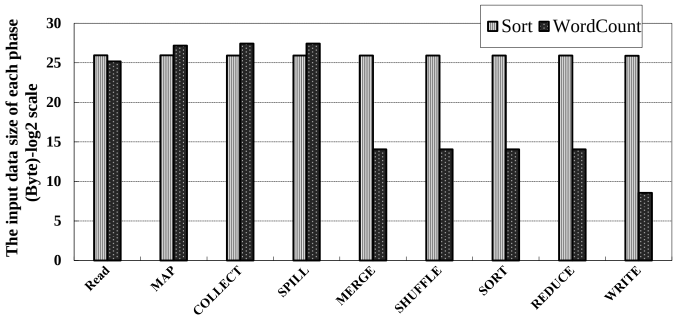

Fig. 4: The size of the data processed by the phases of *sort* and *wordcount* for the same Hadoop configuration.

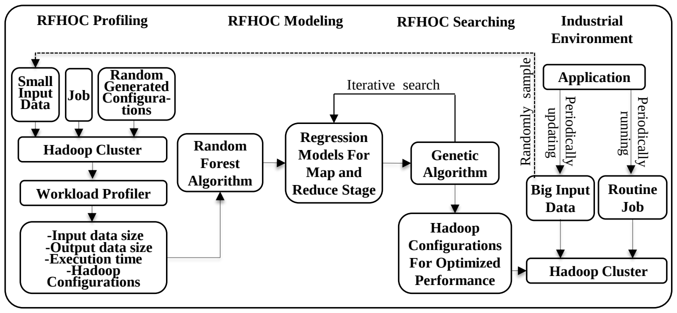

Fig. 5: System architecture of RFHOC.

Hadoop job level to identify the resource consumption characteristics, which is a coarse−grained approach. For example, AROMA groups the *sort* and *wordcount* benchmarks in the same group because they consume similar cluster resources. To verify this assumption, we conduct an experiment to observe the resource consumption for both benchmarks at the phase level, see Figure 4. The phases of *sort* process almost the same amount of data while the phases of *wordcount* are significantly different, which indicates that the resources consumed by each phase is similar for *sort* but is dramatically different for *wordcount*. Since AROMA does not capture these fine−grained resource consumption differences, the SVM-based models classify *sort* and *wordcount* into the same group, which is misleading.

In summary, previously proposed methods make various simplifying assumptions, which render these methods ineffective. These limitations and the advantages of random forest modeling motivate us to propose and explore random forest based models to predict the performance of Hadoop programs.

3 RFHOC Architecture

RFHOC is an automated performance tuning approach that adjusts the Hadoop configuration parameters for an application running on a given cluster to achieve optimized performance. It is designed for repeated long-running applications.

Figure 5 shows the block diagram of RFHOC. When end users first run a Hadoop application, the RFHOC workload profiler collects the Hadoop configurations being used and the execution times for the various MapReduce phases. Subsequently, the phase-level execution times and the corresponding Hadoop configurations are taken as input to the random forest algorithm to train per-phase performance prediction models.

We construct regression models to predict the performance for each phase for both the *map* and *reduce* stages. To build the models, we need to construct a training set S per phase. S is a matrix, with each row of S being the following vector:

$$v_j = \{pt_j, hcp_{1j}, ..., hcp_{ij}, ..., hcp_{nj}\}, j = 1, ..., m, \quad (1)$$

with v_j representing the j^{th} observation of a phase, pt_j the execution time of that phase in the j^{th} observation,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

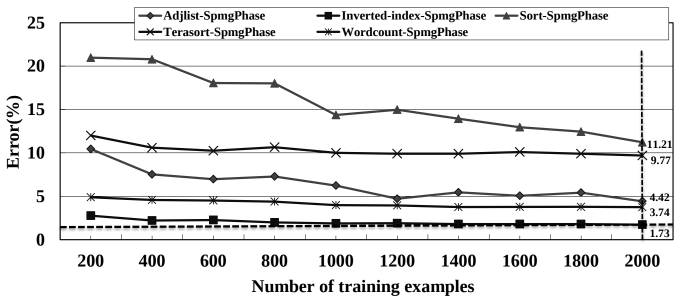

Fig. 6: Performance model error as a function of the number of training examples.

hcp_{ij} the i^{th} Hadoop configuration parameter (these parameters are shown in the first column of Table 1) of the j^{th} observation; n is the total number of Hadoop configuration parameters, and m is the total number of vectors in matrix S (observations or training examples). The training examples are collected in isolation, i.e., no jobs are running concurrently so as not to disturb the measurements.

To collect the vectors for a Hadoop programs, we need to repeatedly run the program, each time with different configurations to form the training set (the matrix S). The number of configurations needs to be determined upfront, during the training phase, and needs to be balanced: a large number of training configurations increases training time, whereas a small number of configurations may not enable accurate performance models. To understand this trade-off quantitatively, we have done the following experiment. We start to train the phase-level performance models using 200 Hadoop configurations, and we increase the training set by 200 each time. All Hadoop configurations are randomly generated with each comfiguration parameter within its respective value range (ranges are shown in Table 3). Figure 6 quantifies how accuracy is affected by the number of training examples, for the *merge* phase for our five benchmarks. We find the model to converge when given 2,000 training examples.

Once we have a performance model for each phase, we still do not know the optimum Hadoop configuration for a given program on a given cluster. To automatically search the optimized configuration, we employ a genetic algorithm (GA). The GA takes the performance predictions obtained by our random forest based models and the corresponding configurations as input to globally search the configuration optimization space, as illustrated in Figure 7. In step ①, we take a set of randomly chosen configurations as input to our performance models, which produce performance estimates at the phase level. The configurations and the corresponding performance estimates then serve as input to the GA. The sum of the phase-level performance estimates provides an estimate for total execution time, and is taken as the fitness value for the GA, as shown in step ②. In our GA, the 'crossover' operation randomly selects k configuration values from one configuration set and $n-k$ values from another configuration set,

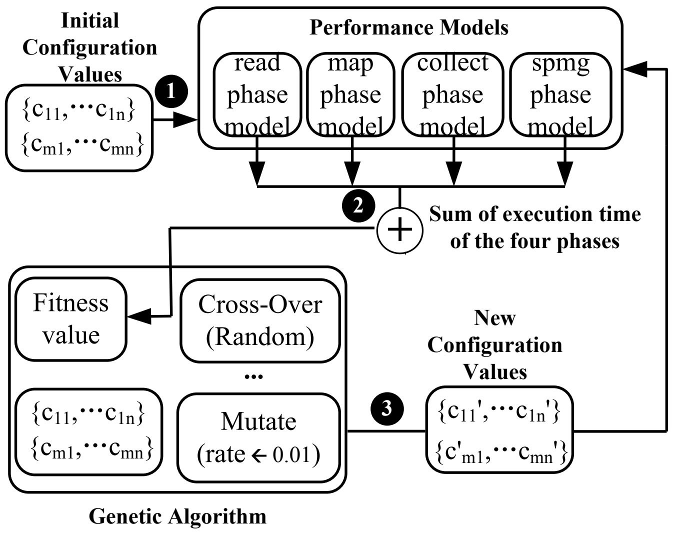

Fig. 7: The genetic algorithm uses the performance models to drive the search towards the optimum Hadoop configuration.

and combines them into a new configuration. In the new configuration, the probability for changing the value of a configuration parameter within its value range is controlled by the mutation rate and is set to 0.01. In step ③, the GA outputs a new set of configurations, which is passed on again to the performance models. Steps ② and ③ are iterated until a configuration is found that yields the highest performance (shortest execution time).

The overall Hadoop optimization framework is sufficiently fast to be useful in practice. Collecting the profiling data (running the Hadoop application of interest with a small input data set for the 2000 training configurations) takes less than two days. Training the performance prediction models using the profiling data takes on the order of minutes. Running the genetic algorithm to tune the Hadoop configuration parameters takes less than half an hour. Overall, we find RFHOC to take less than two days to optimize the Hadoop runtime for a given application — this time overhead is a one-time cost and is well justified as we target long- and repeatedly-running applications.

We now detail on the regression models for the map and reduce stages, and the genetic algorithm to search the Hadoop configuration space.

3.1 Regression Model for the Map Stage

The performance model of the *map* stage of a MapReduce job can be represented as:

$$ET_{map} = \sum_{i=1}^{4} ETph_i, \tag{2}$$

with ET_{map} the execution time of the *map* stage, and $ETph_i$ the execution time for phase i in the *map* stage. $ETph_i$ can be modeled as follows:

$$ETph_i = ETPerWavePh_i \times numTotalWaves, \tag{3}$$

with $ETPerWavePh_i$ the per-wave execution time for phase i, and *numTotalWaves* the total number of waves.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

As mentioned before, the *map* stage has five phases in total. In order to simplify the modeling, we build a single model for the *spill* and *merge* phases as they interact closely, yielding four models in total for the *map* stage. Note that the phases may overlap their execution to some extent; we therefore consider the average per-phase execution time, accounting for the respective proportional shares for each phase. The *numTotalWaves* parameter is computed as follows:

$$numTotalWaves = \lceil numTasks/totalSlots \rceil, \quad (4)$$

with *numTasks* the number of *map* tasks, and *totalSlots* the total number of execution slots in the cluster. If the cluster consists of homogeneous nodes, the *totalSlots* equals:

$$totalSlots = numNodes \times numSlotPerNode, \quad (5)$$

with *numSlotPerNode* the number of task slots per node, and *numNode* the total number of nodes in the cluster. If the nodes in a cluster are heterogeneous, i.e., different nodes can accommodate different task slot counts, the *totalSlots* is the sum of the task slots of all the nodes. The *numTasks* parameter in the *map* stage depends on the size of the input data of a Hadoop program, and is computed as follows:

$$numTasks = totalDataSize/splitDataSize, \quad (6)$$

with *totalDataSize* the size of the input data of a Hadoop program and *splitDataSize* the size of data each map task can handle. In this study, *splitDataSize* equals 64 MB, and *totalDataSize* is easily derived from the input data.

So far, the only parameter for which we do not know its value is $ETPerWavePh_i$. In this paper, we construct a random-forest model to estimate per-wave execution time; as mentioned before, we construct four models, one per phase, with the *spill* and *merge* phases sharing a single model (called *spmg*), see also Figure 7. For example, the model for the *read* phase (phase #1) can be expressed as:

$$ETPerWavePh_1 = f(c), \quad (7)$$

where *f(c)* is a random-forest based model. It takes Hadoop configuration *c* as input and outputs the execution time of the *read* phase in the *map* stage. Note that *f(c)* is a data model which means there is no formula for it. *f(c)* is obtained by using the random forest algorithm to train a model based on the collected observations:

$$RF(\{ETPh_1^k, c_k\}), k \in [1, 2000] \Longrightarrow f(c), \quad (8)$$

with RF the random forest algorithm as shown in Section 2, $ETPh_1^k$ the k^{th} observation on the execution time of the *read* phase, and c_k the Hadoop configuration of the k^{th} observation. $\{ETPh_1^k, c_k\}$ represents the set of 2000 training observations. Once we have constructed models for $ETPerWavePh_i$, we can easily calculate the execution time of the whole *map* stage of a MapReduce job by using formulas 1 through 5. Note that the determination of the two parameters of random-forest learning, *ntree* and *mtry*, will be described in Section 5.1.

Our random-forest based models do not make any assumptions and they are only based on the observed data. We also notice that $ETPh_1^p$ typically differs from $ETPh_1^q$ when c_p is different from c_q (different training examples). In contrast, the model for the *read* phase proposed by Herodotos et al. [8] equals:

$$\begin{aligned} ETPerWavePh_1 = {} & splitDataSize \\ & \times (csHdfsRT + csIncomT), \quad (9) \end{aligned}$$

with *csHdfsRT* the I/O execution time per byte for reading from HDFS, and *csIncomT* the CPU execution time per byte for uncompressing the input. They assume that *csHdfsRT* and *csIncomT* are constants across Hadoop configurations, which we find to be an invalid assumption as shown in Section 2, and which ultimately leads to reduced accuracy.

3.2 Regression Model for the Reduce Stage

The *reduce* stage of a MapReduce job consists of four phases: *shuffle*, *merge*, *reduce* and *write*. The *shuffle* and *sort* phases interact tightly, which is why we consider them as a single phase from a modeling perspective — similar to what we did for the *spill* and *merge* phases in the *map* stage. We model the execution time of the *reduce* stage as follows, similarly to what we did for the the *map* stage:

$$ET_{reduce} = \sum_{i=1}^{3} ETPhred_i, \quad (10)$$

with ET_{reduce} the execution time of the *reduce* stage, and $ETPhred_i$ the execution time of the i^{th} phase in the stage. $ETPhred_i$ can be calculated as:

$$ETPhred_i = numWaves \times perWaveETPhred_i, \quad (11)$$

with *numWaves* the number of waves for the *reduce* tasks, and $perWaveETPhred_i$ the per-wave execution time of the i^{th} phase in the *reduce* stage. *numWaves* is computed as such:

$$numWaves = \lceil numTasks/totalTaskSlots \rceil, \quad (12)$$

with *numTasks* the number of *reduce* tasks. Unlike the *map* stage, *numTasks* for the *reduce* stage is specified by the Hadoop configuration parameter *mapred.reduce.tasks*. Once *numWaves* is specified, the data shuffled by each *reduce* task wave is calculated as follows:

$$shuffleDataPerWave = \frac{totalShuffleData}{numWaves}, \quad (13)$$

with $totalShuffleData$ the total amount of data shuffled by the reduce stage (which is also equal to the amount of input data for the reduce stage). We define $perWaveET_{reduce}$ and *reduceTaskCost* as follows:

$$perWaveET_{reduce} = \sum_{i=1}^{3} perWaveETPhred_i, \quad (14)$$

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

$$reduceTaskCost = \frac{perWaveET_{reduce}}{shuffleDataPerWave}. \quad (15)$$

A small value for *reduceTaskCost* indicates that a large data set (*shuffleDataPerWave*) is processed in a short amount of time ($perWaveET_{reduce}$), yielding higher overall performance. From Equation 13, *shuffleDataPerWave* is affected by Hadoop parameter *mapred.reduce.tasks*. Variable $perWaveET_{reduce}$ in Equation 15 is computed via $perWaveETPhred_i$ which in turn are obtained via the models learned through the training observations.

We use an approach similar to the *map* stage for building a performance model for the *reduce* stage. Unlike the map stage though, we do not directly pass the sum of the execution time of all phases in the *reduce* stage to the GA as its fitness value. Instead, we take the $reduceTaskCost$ calculated by equation 15 as the fitness value of GA. The reason is that the performance of the reduce stage is affected not only by the execution time of the phases but also by the $shuffleDataPerTask$ which is controlled by the Hadoop configuration parameter *mapred.reduce.tasks*. Therefore, $reduceTaskCost$ is a better metric to reflect the performance of the *reduce* stage and using it as the fitness value of GA is more suitable.

3.3 Searching the Hadoop Configuration Space for Optimized Performance

Hadoop configuration parameters can be classified into two groups: parameters that primarily affect *map* task execution versus parameters that primarily affect the *reduce* tasks. For instance, the parameter *io.sort.mb* only affects the *spill* phase in the *map* stage, whereas parameter *mapred.job.shuffle.merge.percent* only affects the *shuffle* phase in the *reduce* stage [8]. We can thus separately search the optimized configuration parameter values for the *map* stage and the ones for the *reduce* stage. The configurations that optimize *map* performance and the ones that optimize *reduce* performance then most likely optimize overall performance when combined. Since we already construct random-forest based performance prediction models for the *map* and *reduce* stages, we can take the outputs of these models as the input for the search algorithm.

There exist many algorithms to search complex optimization spaces, such as recursive random search [15], pattern search [16], and genetic algorithms (GA) [2], [17]. Random recursive search is sensitive to getting stuck in local optima; pattern search typically suffers from slow local (asymptotic) convergence rates [16]. GA is a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover [2], and which is well-known for being robust against local optima [17]. Our goal is to find the configuration for optimized performance of a Hadoop program from the global space of configuration parameters, which is a complex space to explore with many local optima. We therefore employ GA in this study, and we take the output of the performance prediction models as the fitness value of the GA, as explained in the previous sections.

The genetic algorithm is implemented by calling the function 'rbga' in the R library 'genalg' [18]. The 'rbga' function has 11 parameters which are *stringMin, stringMax, suggestions, popSize, iters, mutationChance, elitism, monitorFunc, evalFunc, showSetting* and *verbose*. We set *popSize* (population size) to 200, *iters* (number of iterations) to 100, *mutationChance* to 0.01, *suggestions* to $null$, *elitism* to 40, *monitorFunc* to a monitoring function defined by ourselves showing the status of the GA iterations, and *showSetting* to false. The *evalFunc* is our random-forest based models and the input of these models is the Hadoop configuration which is in the range specified by parameter *stringMin* and *stringMax*. We will show the number of iterations needed by the genetic algorithm to achieve optimized performance in Section 5.

4 Experimental Methodology

The experimental platform consists of ten Sugon servers connected through gigabit Ethernet. Each server is equipped with an Intel Xeon CPU E5-2407 2.20 GHz quad-core processor and 32 GB PC3 memory. In order to fairly compare against the CBO-based approach which runs Hadoop programs in virtual machines (one virtual machine per physical server), we employ the same setup. Virtualization is enabled using Xen v3.0. We create ten virtual machines (VMs) with the same hardware configurations (8 virtual CPUs and 8 GB memory) and run them as Hadoop nodes. Each VM uses SUSE Linux Enterprise Server 11 and Hadoop 1.0.4.

To avoid interference effects among virtual machines running on the same physical node, we make each physical node own one virtual machine. We designate one server to host the master VM node and use the other servers to host the nine slave VM nodes. The master node runs *JobTracker* and *NameNode*. Each slave node runs both *TaskTracker* and *DataNode*. Each VM is initially configured with 4 map slots, 4 reduce slots, and 300 MB memory per task (MPT). The data block size is set to 64 MB. We use the dynamic instrumentation tool *BTrace* [14] to capture the timing characteristics of tasks on each slave node. The other components of RFHOC can run on a separate VM or a standalone machine as it processes the gathered features off-line. We run the five Hadoop programs listed in Table 2, each with five input data sets (50 GB, 100 GB, 200 GB, 500 GB and 1 TB). These programs are taken from the HiBench [19] and Puma benchmark suites [20]. They represent a sufficiently broad set of typical Hadoop workload behaviors. *WordCount* is CPU-intensive in the map stage; *TeraSort* is CPU and memory-intensive in the map stage and disk-intensive in the reduce stage; *Sort* is disk and memory-intensive;*adjlist* is CPU-intensive in the reduce stage;and *inverted-index* is disk and CPU-intensive.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

TABLE 2: Hadoop benchmarks considered in this study.

Hadoop Program	Benchmark Suite	Input Data Generation
Wordcount	HiBench	By Randomtextwriter
Terasort	HiBench	By Teragen
Sort	HiBench	By Randomtextwriter
Adjlist	Puma	By Createadjlistdata
Inverted-Index	Puma	From Wikipedia

TABLE 3: Hadoop configuration parameters explored in this study, with their default value and range.

Configuration Parameters	Default	Range
io.sort.factor	10	10-100
mapred.job.shuffle.merge.percent	0.66	0.2-0.9
mapred.output.compress	false	true or false
mapred.inmem.merge.threshold	1000	10-1000
mapred.reduce.tasks	1	1-1000
io.sort.spill.percent	0.8	0.5-0.9
mapred.job.shuffle.input.buffer.percent	0.7	0.0-0.8
io.sort.record.percent	0.05	0.01-0.5
io.sort.mb	100	(0.25-0.65)*MPT
mapred.compress.map.output	false	true or false

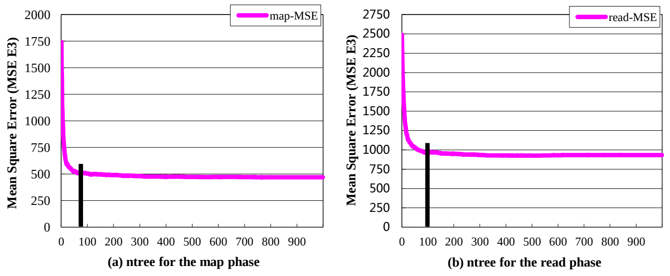

Fig. 8: Determining the *ntree* parameter for the *map* and *read* phases in the *map* stage.

To fairly compare against CBO, we select the 10 configuration parameters that were also used by CBO, see Table 3; the description of these parameters is shown in Table 4. For each benchmark, we evaluate five groups of input data sets (50 GB, 100 GB, 200 GB, 500 GB and 1 TB) generated by their respective input data generators. We employ cross-validation, i.e., we use different data sets for training versus evaluation. We use small input data sets varying between 2 GB to 5 GB at training time, and use the five aforementioned large input data sets (50 GB to 1 TB) for the evaluation.

5 Results and Analysis

In this section, we first present how to determine the *ntree* and *mtry* parameters for our random forest models. Subsequently, we illustrate the accuracy of our performance prediction models for the *map* and *reduce* stage. In addition, we investigate the error distribution and evaluate the scalability of our models with increased number of configuration parameters. Finally, we compare RFHOC with the CBO approach.

5.1 Determining Parameters for Random Forest

To leverage the random forest algorithm to construct models for automatically optimizing Hadoop programs, two parameters — *ntree* and *mtry* — need to be determined. Recall from Section 2 that *ntree* is the total number of trees used to construct an ensemble model, and *mtry* is the number of randomly selected predictor variables at each node of a tree. A high *ntree* value leads to higher accuracy but also results in longer model evaluation time. We conduct experiments to empirically determine a suitable value for *ntree*. Figure 8 illustrates the experiments for the *read* and *map* phases of the *map* stage. (We obtained similar results for the other phases; not shown here because of space constraints.) The mean square error (MSE) converges once *ntree* exceeds 100 for the *read* phase, which is what we set *ntree* to.

According to the suggestion from [21], using a larger *mtry* may give better results if one has a large number of predictor variables but expects only very few to be 'important'. Besides this qualitative suggestion, Efron et al. [11] and Liaw et al. [12] provide quantitative suggestions. Efron et al. recommend the integer part of $log_2 p + 1$ as the default value of *mtry*, while Liaw et al. propose $p/3$. p is the number of Hadoop configuration parameters which is 10 in this paper. Because we construct separate models for the *map* and *reduce* stages, we need to consider p for each stage. With 3 and 4 models for the *reduce* and *map* stages, respectively, the values produced by the above formulas ($log_2 p + 1$ and $p/3$) range between 1 and 3. We tried 1, 2, and 3 for the value of *mtry* in this paper and find 2 to be accurate enough for both stages.

5.2 Model Accuracy for the *Map* Stage

To evaluate the accuracy of our performance prediction models for the *map* stage, we predict and measure the execution times for each phase in the *map* stage 10 times (to remove the bias of different measurements for the same phase) and we subsequently compute the average error for each phase as follows:

$$err = \left(\sum_{i=1}^{t} \frac{|pre_i - mea_i|}{mea_i} \right) \Big/ t \qquad (16)$$

with pre_i the execution time predicted by our model for a given phase, mea_i the measured execution time of that phase, and t the total number of experiments.

Before evaluating the accuracy of our models in detail and comparing it against the state-of-the-art, we first briefly introduce the cost-based optimizer (CBO) [8] because the goals of our approach and CBO are almost the same but with different implementation approaches. The goal of CBO is to find the optimized Hadoop configuration c_{opt} for a given MapReduce program p with input data d running on a given cluster r to finally achieve the shortest execution time. To this end, CBO defines a cost model F which takes the configuration c

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

TABLE 4: Description of the ten Hadoop configuration parameters.

Configuration Parameters	Description
io.sort.factor	The number of streams to merge at once while sorting files.
mapred.job.shuffle.merge.percent	The usage threshold at which an in-memory merge will be initiated during the shuffle phase.
mapred.output.compress	Should the job outputs be compressed during the write phase?
mapred.inmem.merge.threshold	The threshold(the number of files)for the in-memory merge process during the shuffle phase.
mapred.reduce.tasks	The default number of reduce tasks per job.
io.sort.spill.percent	The soft limit in either the buffer or record collection buffers.
mapred.job.shuffle.input.buffer.percent	The percentage of memory to be allocated from the maximum heap size for storing map outputs.
io.sort.record.percent	The percentage of io.sort.mb dedicated to tracking record boundaries.
io.sort.mb	The total amount of buffer memory to use while sorting files during the collect phase, in MB.
mapred.compress.map.output	Whether the outputs of the maps should be compressed during the spill phase.

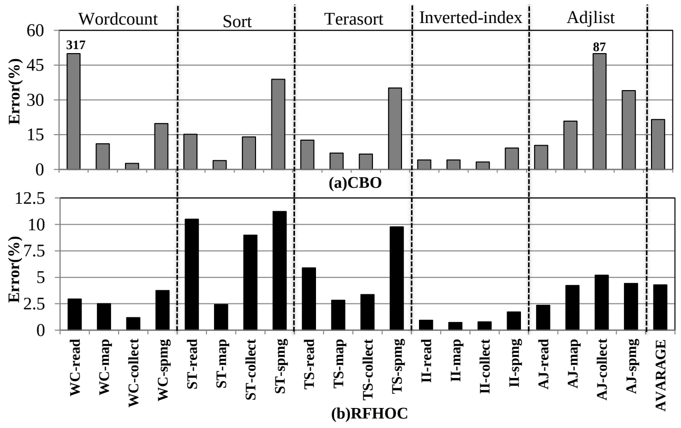

Fig. 9: The prediction error of each phase in the *map* stage of *terasort* (TS), *sort* (ST), *inverted-index* (II), *wordcount* (WC) and *adjlist* (AJ). spmg — the model for *spill* and *merge* phase in the map stage.

as input and outputs the execution time of a phase of a MapReduce program. CBO totally employs 10 configuration parameters based on well-known rules of thumb for optimizing Hadoop performance and F is built based on simulation and model analysis. Herodotou and Babu [8] first employ simulation to gather profiling information (execution time and configuration parameter values) of a MapReduce program. They then analyze the profiling data and create analytical models for each phase. Next, CBO uses recursive random search (RSS) to search the optimized configuration using the analytical models. The performance prediction models used in CBO are fine-grained models and they can output the execution time of a phase much faster compared to a simulation-based approach. However, the models used in CBO make some oversimplified assumptions, as illustrated in Section 2.3. These unrealistic assumptions introduce modeling errors; RFHOC on the other hand does not make any assumptions and we therefore expect it to be more accurate than CBO.

To fairly compare against the CBO approach, we re-implement CBO in our experimental environment and we perform the same experiment as what we do for our approach. Figure 9 illustrates the prediction errors of all the phases in the *map* stage for *terasort*, *sort*, *inverted−index*, *wordcount* and *adjlist*. The performance prediction accuracy for our RFHOC approach is significantly higher than for CBO. (Note the different scales for CBO versus RFHOC in Figure 9.) The maximum error for RFHOC (11.2%) occurs for the *merge* and *spill* phase of the *sort* benchmark. This is because the *spill* phase of *sort* contains a complex combination operation that significantly reduces the data size processed by the *merge* phase. The average error for *wordcount*, *adjlist*, *terasort*, and *inverted−index* is less than 5.5%; the maximum average error is observed for *sort* (8.3%). Overall, RFHOC has an average and maximum per-model error of 4.3% and 11.2%, respectively. In contrast, CBO's error is much higher, with an average and maximum per-model error of 32.9% and 317.7%, respectively.

5.3 Model Accuracy for the *Reduce* Stage

We now evaluate the accuracy of the classification models for the *reduce* stage, see Figure 10. The accuracy of RFHOC is significantly higher than CBO. The maximum error of RFHOC is 23.4% for the *sfst* model for the *adjlist* benchmark. The reason is that the *shuffle* and *sort* phases of *adjlist* strongly interact with each other in a complex way, complicating predicting its execution time. However, this maximum error is still extremely low compared to CBO with an error of 644%. On average, the error of RFHOC is 8.7% while that of CBO is 109.7% for the *reduce* stage.

Another interesting result is that the error is significantly higher for the *reduce* stage compared to the *map* stage, for both RFHOC and CBO. From Figure 9 and 10, we observe that the average error for RFHOC for the *map* stage is only 4.3% compared to 8.7% for the *reduce* stage. For CBO, the average error equals 32.9% for the *map* stage compared to 109.7% for the *reduce* stage. This indicates that it is more difficult to predict the performance of the phases for the *reduce* stage than those for the *map* stage, which reflects their complexity.

5.4 Error Distribution

The model accuracy evaluation presented so far used the average error for each phase. Although the average error is a good metric to evaluate a model's accuracy

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

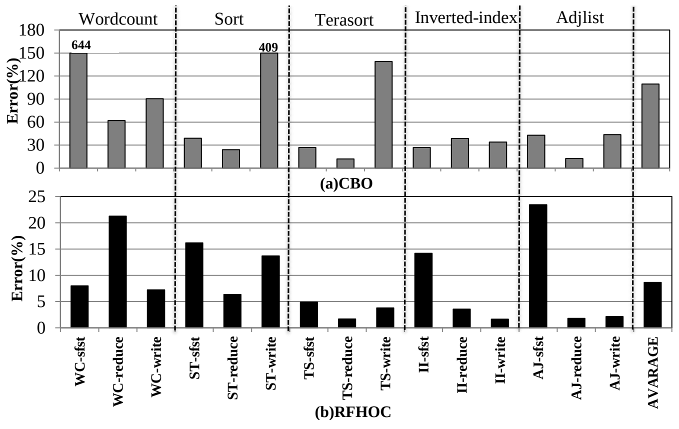

Fig. 10: The prediction error of each phase in the *reduce* stage of *terasort* (TS), *sort* (ST), *inverted-index* (II), *wordcount* (WC) and *adjlist* (AJ). sfst — the model for the *shuffle* and *sort* phase in the reduce stage.

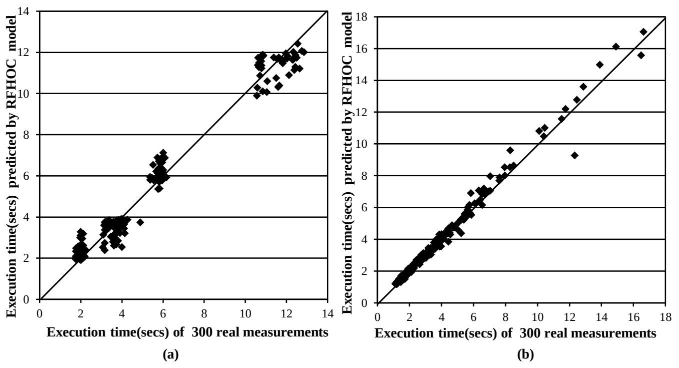

Fig. 11: Error distribution illustrating prediction versus real measurement for 300 randomly selected Hadoop configurations for *TeraSort*. (a) shows the error distribution for the *spmg* model in the map stage; (b) shows the error distribution for the *sfst* model in the reduce stage.

statistically, it might hide large errors for particular predictions due to outliers. To address this issue, we now present the error distribution of our prediction models using scatter plots. Figure 11 shows two scatter plots produced by 300 real measurements and 300 RFHOC predictions for benchmark *TeraSort* for 300 randomly selected Hadoop configurations. (a) is for the *spmg* phase of the map stage and (b) is for the *sfst* phase of the reduce stage. (We obtain similar results for the other phases and benchmarks.) The X axis represents the real measurements and the Y axis denotes the RFHOC predictions of the execution times (in seconds) of a phase with 300 different Hadoop configurations. This graph clearly shows that the model is fairly accurate across the entire Hadoop configuration space: all 300 data points are located around the bisector, which indicates that the predictions are close to the measurements. We observe a standard deviation of the predictions of 0.113 and 0.055 for the *spmg* and *sfst* models, respectively. We observe similar results for the other benchmarks and phases.

5.5 Model Scalability

So far, we assumed 10 Hadoop configuration parameters. However, the Hadoop framework has 190 configuration parameters and we want to know how scalable the models created by a random-forest learning based approach are. To this end, we build performance models for the map and reduce stage using 34 configuration parameters which include the previously used 10 parameters. In particular, we include 5 parameters related to data properties as listed in Table 5. Note that the data compression ratio is controlled by the *compression codec* in Table 5 and we use three different ratios generated by *gzip,LTO*, and *zlib*. Due to space limitation, we do not detail the other 19 configuration parameters in this paper.

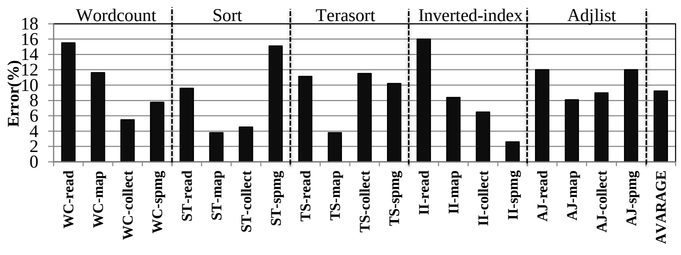

Fig. 12: The prediction error of each phase in the *map* stage when the models are built by 34 configuration parameters.

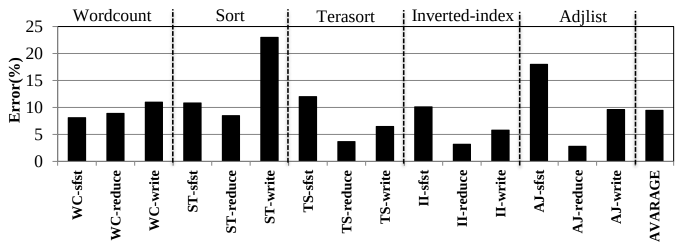

Fig. 13: The prediction error of each phase in the *reduce* stage when the models are built by 34 configuration parameters.

Figure 12 shows the relative errors for the phase models for the map stage. As can be seen, the high errors come from the *read* phase for benchmarks *WordCount* and *inverted-index*, and the *spmg* model for benchmark *Sort*, which are 15.8%, 16%, and 15.6%, respectively. This is because the compression ratio has very significant impact on the performance of the *read* phase while we only have three records including *gzip,LTO*, and *zlib* to train the models. However, these high errors are still extremely low compared to those of the CBO approach. The average error of the prediction models for the map stage is only 9.2%. While this is higher than the 4.3% of average error for the models built by the 10 configuration parameters for the map stage, it shows good scalability because the error increases by a factor of $2.1\times$ but the number of parameters of the models increases by a factor of $3.4\times$. In addition, the number of vectors needed in the training set of the models equals 3300, which is $1.65\times$ larger than the training set (2000 observations) for the models with 10 parameters, showing good scalability of our approach as well.

Figure 13 illustrates the errors of the prediction models for the *reduce* stage. We observe the highest error for the *write* phase of the benchmark *Sort*. However, the errors

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

TABLE 5: Description of the data properties.

Data Properties	Description
compression	Whether the input data compress or not compress.
compression codec	The codec is a program(gzip,LTO,zlib) capable of performing encoding and decoding on a input digital data stream.
compression type	If input data compress, then what type of compression should be used (RECORD or BLOCK)
split size	The chunk size of the input files.
input pair width	The width of input key-value pairs.

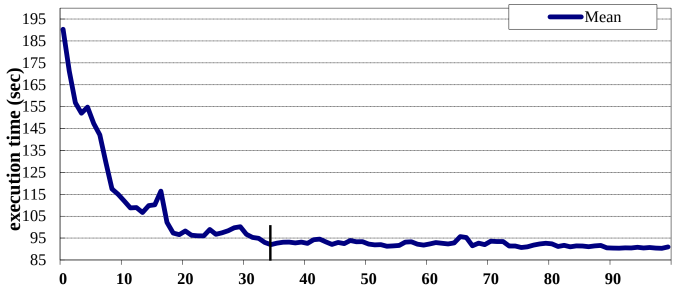

Fig. 14: The number of iterations of GA for the *map* stage for the *adjlist* benchmark.

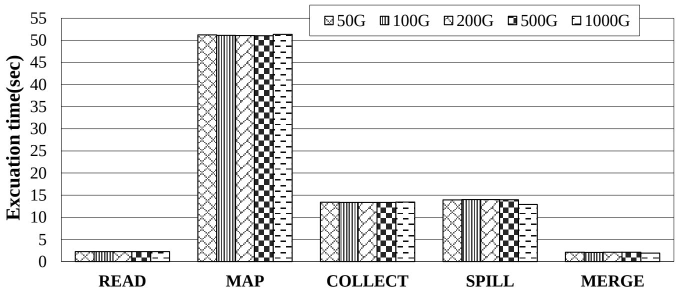

Fig. 16: The impact of input data size on the execution time of per-wave phases in the *map* stage of *inverted-index*.

for the phases of the other benchmarks are very low and the average error is only 9.8%. Compared to the map stage, our approach even shows better scalability because the error only increases by a factor of $1.1\times$ when the number of configuration parameters used to build the models for the reduce stage increases by a factor of $3.4\times$. Note that the number of vectors needed to train the prediction models for the reduce stage is similar to that for the map stage.

5.6 Speedup

Recall that the ultimate goal for our work is to improve Hadoop's performance. We optimize Hadoop's performance by using the GA to search the Hadoop configuration space using the random-forest models as input. Figure 14 shows the number of iterations for the GA for the *map* stage of the *adjlist* benchmark. The GA converges after 30 iterations; we find the GA to also quickly converge for the other benchmarks. The number of iterations to converge ranges from 30 to around 90, which indicates that the GA can efficiently identify the Hadoop configuration for optimized performance. Note that one iteration of the GA is done very quickly because it only requires evaluating the random forest model, hence, the overall exploration using the GA is done on the order of a couple tens of minutes (less than half an hour).

We now evaluate the speedup obtained by RFHOC. We re-implement the CBO approach [8] in our experimental environment for a fair comparison. Figure 15 shows the execution time and speedup for RFHOC over CBO for *inverted−index*, *terasort*, *sort*, *wordcount* and *adjlist*. Because the execution time of the benchmarks with the 1 TB input data set is much longer than for the smaller inputs, we report a logarithmic scale on the vertical axis. Clearly, programs tuned by RFHOC run much faster compared to the CBO approach. We define the speedup of programs tuned by RFHOC over CBO as follows:

$$speedup = ET_{CBO}/ET_{RFHOC} \tag{17}$$

with ET_{RFHOC} and ET_{CBO} the execution time of a program with Hadoop's runtime tuned by RFHOC and CBO, respectively. The minimum and maximum speedup of RFHOC over CBO are 1.05× and 7.4×, respectively. On average, the speedup equals 2.11×. In addition, we obtain two interesting observations. First, RFHOC achieves different speedups for different programs. The speedups for *wordcount* and *adjlist* are higher than 1.22× across all input data sets, which are more significant than for *inverted−index*, *sort* and *terasort* (the maximum speedup is lower than 1.96×). This indicates that fine-tuning the Hadoop parameters should indeed be done on a per-application basis.

Second, for a given program, RFHOC generally achieves higher speedup for increasingly larger input data sets, which is a nice property for big-data applications with expected ever-increasing data sets. This requires that the per-wave execution time ($ETPerWavePh_i$ in Equation 2) does not change significantly with increasing input data set sizes. We conduct experiments to verify this property. Figure 16 shows the variation of the execution time of the per-wave phases in the *map* stage for the *inverted-index* benchmark as we vary the input data set from 50 GB to 1 TB. As can be seen, the execution time does not change significantly. (We observe similar results for other programs.) This result indicates that RFHOC scales out well with increasingly larger input data sets.

5.7 Detailed Analysis: Wordcount

We now further analyze the results in more detail by considering one particular benchmark, namely *wordcount*, for which RFHOC achieves the highest speedup.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

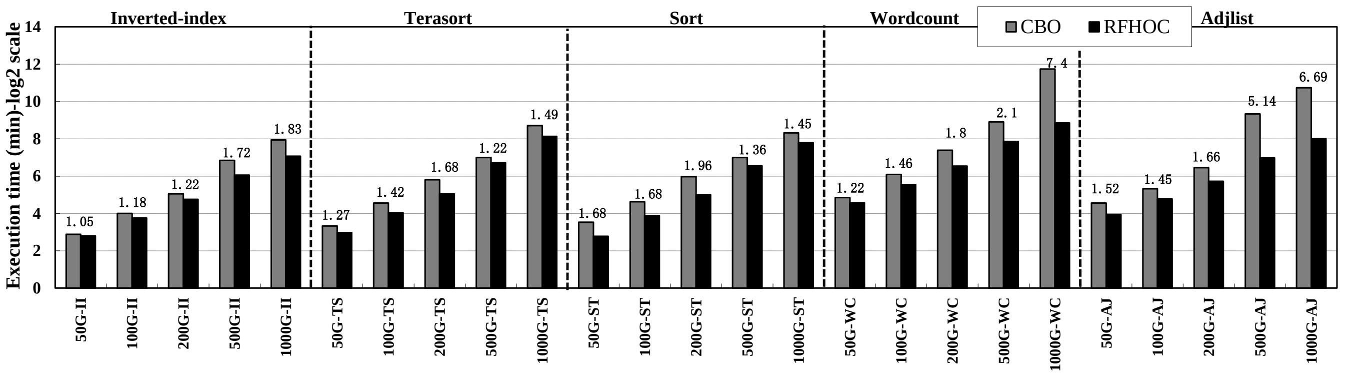

Fig. 15: Execution time for RFHOC and CBO (vertical axis) and speedup of RFHOC over CBO (numerical labels) for all benchmarks *terasort* (TS), *sort* (SO), *inverted-index* (II), *wordcount* (WC) and *adjlist* (AJ); five input data sizes (50 GB, 100 GB, 200 GB, 500 GB, and 1 TB) are considered per benchmark.

TABLE 6: Configuration parameters for *wordcount*.

Configuration Parameter	CBO	RFHOC
io.sort.mb	51	120
mapred.job.shuffle.merge.percent	0.32	0.41
mapred.output.compress	true	false
mapred.inmem.merge.threshold	710	650
mapred.reduce.tasks	620	36
io.sort.spill.percent	0.83	0.89
mapred.job.shuffle.input.buffer.percent	0.60	0.22
io.sort.record.percent	0.09	0.04
io.sort.factor	98	72
mapred.compress.map.output	true	false

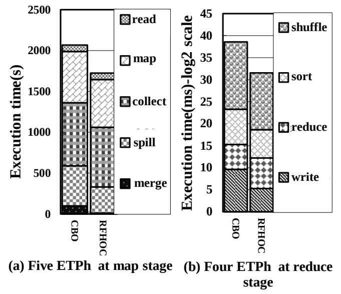

Fig. 17: Execution time comparison of the phases in the *map* and *reduce* stages for *wordcount* with CBO and RFHOC.

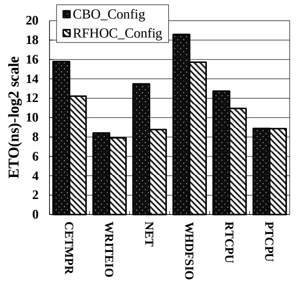

Fig. 18: Execution time comparison of the individual operations of *wordcount* with CBO and RFHOC.

Table 6 shows the configuration parameter values suggested by CBO and RFHOC. We ran and profiled the *wordcount* benchmark twice with the two groups of configurations settings. Figure 17 (a) and (b) shows the execution times of the various phases in the *map* and *reduce* stages, respectively, for the CBO and RFHOC configurations with the 50 GB compressed input data set. Figure 18 quantifies the execution times for the individual operations under the two configurations, which we find to be shorter across the board for the RFHOC configuration compared to the CBO configuration.

From Table 6, we observe that RFHOC selects a larger buffer memory size at the *collect* phase (through *io.sort.mb*) than CBO. The smaller size of buffer memory would directly lead to a larger number of spilled files at the *spill* phase. These files then need to be merged at the *merge* phase which could increase the value of CETMPR and WRITEIO as illustrated in Figure 18. Secondly, setting more *reduce* tasks (through *mapred.reduce.tasks*) may lead to a larger number of small files that need to be transferred via the network and then need to be processed by the *reduce* task. Thus it increases the value of NET and RDCPU (Figure 18). In addition, setting *mapred.compress.map.output* as true would increase the CPU cost for compressing which causes imbalance between I/O and CPU resource utilization. Actually, it increases both CETMPR and WRITEIO, see Figure 18. Similarly, setting *mapred.output.compress* as true may increase the I/O execution time of writing to HDFS per byte (WHDFSIO) as shown in Figure 18.

The *wordcount* benchmark is a CPU-bound application that counts the number of occurrences of each word in a given input file. Particulary, *wordcount* specifies a combiner at the *spill* phase. Hence, the output of each *map* is passed through the local combiner to make local aggregation on sorted keys. Actually, the combining operation may compete with other mappers for CPU resources. Setting improper configurations may aggravate this contention, resulting in poor performance. The higher execution times for the individual operations further aggravates resource contention. It significantly increases the ETPh of CBO at the *spill, merge, shuffle* and *write* phases, as illustrated in Figure 17 (a) and (b). Overall, RFHOC saves about 22% total execution time compared to CBO in these phases.

We further observe the details of execution time of phases and the *reduceTaskCost* of *wordcount* running with the CBO and RFHOC configurations when the sizes of input data set are 50 GB, 100 GB, 200 GB, 500 GB and 1 TB. From Figure 19 (a) and (b), we observe that RFHOC is able to speed up the phase executions over CBO at the *map* stage as input data size increases. Secondly, the *reduceTaskCost* for RFHOC increases more slowly compared to CBO at the *reduce* stage. This indicates that the CBO approach cannot minimize all the ETPhs in the MapReduce work flow, which affects total execution time. On the contrary, RFHOC is able to capture the

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

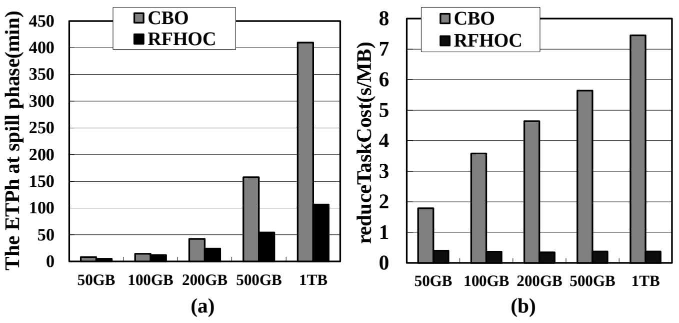

Fig. 19: The execution time for the *spill* phase (a), and the $reduceTaskCost$ (b) for *wordcount* as a function of input data set size.

key characteristics of each workload to achieve a performance speedup of up to 7.4× as the input data size increases to 1 TB.

6 Related Work

The studies closest to our work are [6]–[8], [22] as they also focus on tuning the Hadoop configuration parameters to optimize Hadoop program performance. Herodotou et al. [6]–[8] develop analytical models to predict the performance of the various phases of Hadoop/MapReduce jobs, and they propose a cost-based optimization (CBO) approach leveraging the analytical models to optimize Hadoop's runtime configuration. However, the models are based on various simplifying assumptions, as described in Section 2.3, which renders the obtained configurations to be suboptimal. RFHOC on the other hand does not make any assumptions and is based on an advanced learning approach, namely random forest, along with a genetic algorithm to search the optimization space, yielding overall better Hadoop performance.

Liao et al. [22] propose Gunther, which also searches the best Hadoop configuration for a given application directly using a genetic algorithm. However, this approach uses five configuration parameters only, and the performance improvement is significantly lower than through RFHOC. Moreover, Gunther needs to run the target Hadoop program once for each iteration of the genetic algorithm, which is time-consuming (and impractical) when optimizing real Hadoop applications with large input data sets. In contrast, RFHOC employs performance models to predict performance during each iteration of the genetic algorithm, which is much faster.

Several related studies optimize Hadoop systems by optimizing resource allocation [9], [23], which is an orthogonal problem. These studies consider a scenario in which Hadoop programs run in the cloud, and share resources with other applications, and in which the hardware or virtualized hardware resources may change dynamically. Lama et al. [9] propose a Support Vector Machines (SVM) based approach to optimize the resource allocation and Hadoop configurations for applications running in the cloud. Kambatla et al. [23] study strategies for provisioning a MapReduce job. The goal of these studies is different than ours — dynamically allocating hardware resources versus setting software parameters in our work. Moreover, the resource utilization patterns used in these works are based on coarse-grained models which may cause significant errors and may lead to local optima, as discussed in Section 2.3.

Scheduling is yet another popular way to optimize Hadoop systems. Verma et al. [24] design a job scheduler to determine the job ordering. Zaharia et al. [25] propose a job scheduler for the Hadoop framework to improve the performance of MapReduce programs running in heterogeneous virtualized environments. Bu et al. [26] study the impact of interference of virtual machines and different levels of data locality on performance. They then propose interference and locality-aware task scheduling policies to optimize MapReduce application in virtual clusters. Our approach targets repeatedly running applications on a given cluster and optimizes Hadoop performance through the orthogonal approach of tuning Hadoop configuration parameters.

Another kind of related work is about using genetic algorithm (GA) to optimize computer system performance. Corcran et al. leverage a GA to optimize the file placement in a distributed system [27]. Vera et al. use a GA to optimize program locality [28]. Joshi et al. employed GA to generate benchmarks that stress microprocessor power consumption [29]. We employ GA for exploring the Hadoop configuration space driving by performance models built using random forest learning.

7 Conclusion

Performance tuning is a challenging problem for Hadoop/MapReduce workloads because of the large number of Hadoop configuration parameters. Yet, there is a significant opportunity for optimizing performance beyond Hadoop's default runtime configuration parameters. Previously proposed techniques to automatically tune the Hadoop configuration parameters build analytical models based on oversimplified assumptions, affecting the overall model's accuracy and ultimately the achievable performance improvements.

In this paper, we propose RFHOC, a novel methodology to optimize Hadoop performance by leveraging the notion of a random forest to build accurate and robust performance prediction models for the phases of the *map* and *reduce* stage of a Hadoop program of interest. Taking the output of these models as the input to a genetic algorithm to automatically search the Hadoop configuration space yields a Hadoop configuration setting that leads to optimized application performance. We evaluate RFHOC using 5 Hadoop benchmarks, each with 5 input data sets ranging from 50 GB to 1 TB. The results show that RFHOC speeds up Hadoop programs significantly over the previously proposed Cost-Based Optimization (CBO) approach. The speedups achieved are significant: by 2.11× on average and up to 7.4×. Furthermore, we find RFHOC's performance benefits to increase with increasing input data set sizes.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2449299, IEEE Transactions on Parallel and Distributed Systems

References

[1] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tummarello, "Sindice.com: A documentoriented lookup index for open linked data," *International Journal on Metadata Semantics Ontologies*, vol. 3, pp. 37–52, Nov. 2008.

[2] L. Lie, "Heuristic artificial intelligent algorithm for genetic algorithm," *Key Engineering Materials*, vol. 439, pp. 516–521, 2010.

[3] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita, "Jaql: A scripting language for large scale semistructured data analysis," in *Proceedings of the VLDB Conference*, Sep. 2011, pp. 1272–1283.

[4] M.-P. Wen, H.-Y. Lin, A.-P. Chen, and C. Yang, "An integrated home financial investment learning environment applying cloud computing in social network analysis," in *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, Jul. 2011, pp. 751–754.

[5] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg, "Searching for SNPs with cloud computing," *Genome Biol*, vol. 10, no. 11, p. R134, 2009.

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

[7] 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.

[8] 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.

[9] P. Lama and X. Zhou, "Aroma: Automated resource allocation and configuration of MapReduce environment in the cloud," in *Proceedings of the 9th ACM International Conference on Autonomic Computing (ICAC)*, Sep. 2012, pp. 63–72.

[10] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[11] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. CRC press, 1994, vol. 57.

[12] A. Liaw and M. Wiener, "Classification and regression by RandomForest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[13] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[14] "A dynamic instrumentation tool for java." [Online]. Available: kenai.com/projects/btrace

[15] 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.

[16] V. Torczon and M. W. Trosset, "From evolutionary operation to parallel direct search: Pattern search algorithms for numerical optimization," *Computing Science and Statistics*, pp. 396–401, 1998.

[17] M. Kumar, M. Husian, N. Upreti, and D. Gupta, "Genetic algorithm: Review and application," *International Journal of Information Technology and Knowledge Management*, vol. 2, no. 2, pp. 451–454, 2010.

[18] C. B. Lucasius and G. Kateman, "Understanding and using genetic algorithms part 1. concepts, properties and context," *Chemometrics and Intelligent Laboratory Systems*, vol. 19, no. 1, pp. 1–33, 1993.

[19] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the Mapreduce-based data analysis," in *Proceedings of the 26th IEEE International Conference on Data Engineering Workshops (ICDEW)*, Mar. 2010, pp. 41–51.

[20] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue MapReduce benchmarks suite," Purdue University, Tech. Rep., 2012.

[21] C. Strobl, J. Malley, and G. Tutz, "An introduction to recursive partitioning: Rationale, application, and characteristics of classification and regression trees, bagging, and random forests." *Psychological methods*, vol. 14, no. 4, pp. 323–323, 2009.

[22] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based autotuning of MapReduce," in *Proceedings of Euro-Par 2013 Parallel Processing*, Aug. 2013, pp. 406–419.

[23] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (HotCloud)*, Jun. 2009.

[24] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC)*, Jun. 2011, pp. 235–244.

[25] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments." in *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2008, pp. 29–42.

[26] X. Bu, J. Rao, and C.-z. Xu, "Interference and locality-aware task scheduling for MapReduce applications in virtual clusters," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, Jun. 2013, pp. 227–238.

[27] A. L. Corcran and D. A. Schoenefeld, "A genetic algorithm for file and task placement in a distributed system," in *Proceedings of the 1st IEEE World Congress on Computational Intellignece*, Mar. 1994, pp. 340–344.

[28] X. Vera, J. Abella, A. Gonzalez, and J. Llosa, "Optimizing program locality through gmes and gas," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques(PACT)*, Mar. 2003, pp. 68–78.

[29] A. M.Joshi, L. Eeckhout, L. K.John, and C. Isen, "Automated microprocessor stressmark generation," in *Proceedings of the 15th International Symposium on High Perforance Computer Architecture(HPCA)*, 2008, pp. 229–239.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.