A Self-Tuning System based on Application Profiling and Performance Analysis for Optimizing Hadoop MapReduce Cluster Configuration

Dili Wu* and Aniruddha Gokhale*
* ISIS, Dept of EECS, Vanderbilt University,
1025 16th Ave S, Nashville, TN 37212, USA
Email:{dili.wu,a.gokhale}@vanderbilt.edu

***Abstract*—One of the most widely used frameworks for programming MapReduce-based applications is Apache Hadoop. Despite its popularity, however, application developers face numerous challenges in using the Hadoop framework, which stem from them having to effectively manage the resources of a MapReduce cluster, and configuring the framework in a way that will optimize the performance and reliability of MapReduce applications running on it. This paper addresses these problems by presenting the Profiling and Performance Analysis-based System (PPABS) framework, which automates the tuning of Hadoop configuration settings based on deduced application performance requirements. The PPABS framework comprises two distinct phases called the Analyzer, which trains PPABS to form a set of equivalence classes of MapReduce applications for which the most appropriate Hadoop configuration parameters that maximally improve performance for that class are determined, and the Recognizer, which classifies an incoming unknown job to one of these equivalence classes so that its Hadoop configuration parameters can be self-tuned. The key research contributions in the Analyzer phase includes modifications to the well-known $k-means++$ clustering and Simulated Annealing algorithms, which were required to adapt them to the MapReduce paradigm. The key contributions in the Recognizer phase includes an approach to classify an unknown, incoming job to one of the equivalence classes and a control strategy to self-tune the Hadoop cluster configuration parameters for that job. Experimental results comparing the performance improvements for three different classes of applications running on Hadoop clusters deployed on Amazon EC2 show promising results.**

Keywords*—*MapReduce; Hadoop; self-tuning; optimization.

I. Introduction

As the amount of data to be analyzed keeps growing and the need to process it fast becomes important, classical parallel data warehousing techniques, which traditionally have been the best technologies to store, manage and analyze data, tend now to be inefficient, inflexible and thus incredibly expensive for both commercial and technical consideration [1]. Scientists and engineers are therefore adopting new parallel programming models which can not only meet the requirements of scalability, but also automatically handle resource management, fault tolerance, and other issues on distributed system [2]. One such paradigm called MapReduce [3], is becoming quite popular for high performance computing and Big Data analytics.

The basic concept behind MapReduce is breaking a computation task into *map* and *reduce* phases [3], [4]. At the beginning of the map phase, the input job is divided and then assigned to several worker nodes by a master node. Then the map function running on the worker nodes converts the original data to a series of key-value pairs for future use. In the reduce phase, the master node collects the output of the map function from the worker nodes, and then assigns reduce tasks that do aggregating, combining, filtering or transforming functions on these key-value pairs using a user-supplied function to form the final output [4], [5].

A widely used, open-source platform for building MapReduce-based applications is Apache Hadoop [6]. Despite the popularity of Hadoop MapReduce, application developers face a number of challenges using Hadoop to obtain the most effective performance for their applications. For example, applications developers often must decide how best to make use of the cluster resources, and thus optimize its performance for both general jobs, such as sorting and searching, and other specific applications [7].

Recently, researchers have shown that Hadoop cluster configurations play a significant role on the performance delivered to applications, *i.e.*, even a tiny change to one configuration parameter's value makes a huge difference to performance when running the same MapReduce job with the same size of data input [8]. Moreover, because of its blackbox-like feature, it is also incredibly difficult to find a straightforward mathematical model relating the cluster configuration to a specific job. In summary, it is unreasonable to use the same configuration for all kinds of MapReduce jobs; however, it is also hard for developers to find an optimal configuration for their job.

To address this problem, we have developed the *Profiling and Performance Analysis Based Self-tuning (PPABS)* framework. The PPABS framework comprises two distinct phases called the Analyzer, which trains PPABS to form a set of equivalence classes of MapReduce applications for which the optimal[1] Hadoop configuration parameters are determined, and the Recognizer, which classifies an incoming unknown job to one of these equivalence classes so that its Hadoop configuration parameters can be self-tuned.

This paper makes the following research contributions in the context of the two phases of the PPABS framework:

[1] In this paper "optimal" refers to the desired solution that will maximize performance improvements.

- The Analyzer combines profiling of MapReduce job performance with data mining techniques to dynamically classify MapReduce applications into a set of well-defined partitions or equivalence classes. Specifically, we provide a modified $k{-}means{+}{+}$ clustering algorithm. Additionally, it includes modifications to the well-known Simulated Annealing algorithm to find the desired solution and fine-tune the Hadoop cluster configuration for the job classes identified in the first step.
- The Recognizer classifies an unknown incoming job to one of the equivalence classes by first executing the job on a small portion of its inputs using default Hadoop configurations and applying pattern recognition techniques to classify it. After classifying the job to one of the equivalence classes, the configuration settings are automatically applied to provide significantly improved performance for the new incoming job.

To evaluate PPABS, we deployed a Hadoop cluster in Amazon EC2 and trained our PPABS system to create three equivalence classes using applications drawn from the Hadoop Examples set, Hadoop Benchmarks set, and HiBench. The experimental results comparing the performance improvements made possible by PPABS for three representative applications one per equivalence class: Word Count, Tera Sort, and Grep over default configurations supported by Hadoop show promising results.

The rest of this paper is organized as follows. Section II discusses related work and compares them with our system; Section III presents in detail the design and implementation of PPABS; In Section IV we evaluate PPABS and analyze the performance improvements seen in our three representation application examples; and finally in Section V, we provide concluding remarks and discuss future work in this field.

II. Related Work

Research on performance analysis of MapReduce started more than six years ago. Most of the earliest works studied the scheduling and fault tolerance mechanisms of MapReduce, such as the one presented in [9] that discussed the mathematical model of MapReduce. In [10], the authors point out that Hadoop's scheduler can cause severe performance degradation in heterogeneous environments and present a new scheduler called Longest Approximate Time to End. Another work focused on analyzing the variant effect of resource consumption of different settings for the Map and Reduce slots, which was described in [11]. Based on the experiments, the authors showed that the difference in computation utilization pattern depends on different kinds of MapReduce jobs.

In [12], the authors classified MapReduce applications into three categories based on their CPU and I/O utilization. Even though it was the first such effort illustrating an approach to improving performance by categorizing jobs, the downside of this approach is that it considered the average utilization of CPU and I/O as the only criteria to classify MapReduce jobs and thus overlooked the more important part, which is the overall pattern of the performance characteristics of the application.

There are not many existing approaches on solving the performance problem that focus on tuning configuration parameters. A related work that characterized resources and job utilization patterns from analyzing ten months of MapReduce logs of Yahoo appears in [13]. This article pointed out that two MapReduce applications can be considered to be of the same kind if their performance pattern is similar; moreover, it concluded that similar jobs would have similar optimal solutions for the cluster configuration. In our work, we use these guidelines.

In one of the recent studies that combines MapReduce with Machine Learning algorithms, researchers in the University of Sydney modeled the relation between applications' configuration parameters and their CPU usage history [14]. However, instead of studying performance improvement with this model, they used it only to predict the entire CPU usage in time clock cycles for the unknown job.

AROMA [15] and Starfish [16] are two recent research efforts that are related to our approach. The former system can automatically allocate the cluster resources by adapting the cluster to new jobs if their resource utilization signature matches the previously executed jobs. Such an approach does not guarantee the performance of jobs whose utilization pattern is different with any previous ones. More importantly, this paper did not present a clear way to find the optimal configuration solution even for the executed jobs. The latter effort is an attempt to find the optimal cluster settings by collecting the profile information of previous jobs, simulating executing time of new jobs, and then searching for the parameter space for solutions. However, since Hadoop MapReduce is such a complex framework with a number of parts for whom well-defined mathematical models are not yet established, we believe that the "What-If Engine Theory" introduced in this related work has the potential to fail in its searching strategy for optimal parameters.

Despite several attempts to improve the performance of MapReduce cluster exist, we believe these related works do not provide a comprehensive and reliable solution to optimize the configuration settings of MapReduce cluster, which have been shown to impact performance. Even though there are some slight similarities, our research is different from all the works above; we not only use data mining techniques to analyze a job's profile, but also optimize the performance of MapReduce applications in a robust manner, no matter whether they are already known to have executed or are totally new.

III. PPABS Design and Implementation

This section details our PPABS (Profiling and Performance Analysis-Based Self-tuning) approach. Since we target the automatic configuration management for applications deployed in the the Hadoop cluster, we first provide an overview of Hadoop and its configurability. Subsequently we describe the PPABS details.

A. Overview of Hadoop

Apache Hadoop is an open-source software framework implementing the MapReduce computing model that supports reliable and scalable Big Data computing. This framework is written in the Java language and it consists of four major modules: Hadoop Common module, Hadoop Distributed File System module, Hadoop YARN module and Hadoop MapReduce module. These modules can be tuned through configuration parameters, which together dictate the performance delivered to the applications.

The Hadoop Common module contains the common utilities that support the other Hadoop modules. In this module, scripts and Java Archive files are provided to support the basic operations of Hadoop such as starting and stopping Hadoop, formatting the NameNode and so on. In addition, it also provides source code, documentation and other necessary files of Hadoop for developers.

The Hadoop Distributed File System (HDFS) module is a distributed file system that provides high throughput access to data. Compared to other existing distributed file systems, it is designed to work on low-cost, commodity hardware and to provide high quality fault-tolerance at the same time. The HDFS module has a master/slave architecture shown in Figure 1 in which a NameNode is the master that manages the file system name space, and several DataNodes serve as slaves that manage only the node they are running on.

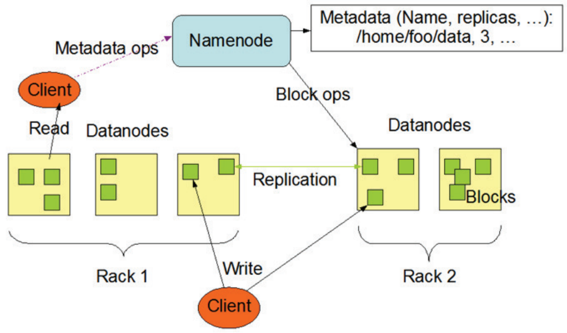


Fig. 1. Hadoop Architecture with HDFS Artifacts

The Hadoop Yarn module is a framework for job scheduling and resource management. The fundamental idea behind this module is to split the JobTracker, which is responsible for supervising the running process of MapReduce applications, into a Resource Manager and an Application Master. The former arbitrates the resources of the entire Hadoop system, while the latter is responsible to track and monitor the status of Resource Containers from the Scheduler, which is an important component of the Resource Manager.

The Hadoop MapReduce module is a Hadoop YARN-based system for parallel data processing. Similar to the HDFS module, it has a master/slave architecture. For each MapReduce job, a JobTracker serves as a master which can be regarded as an interaction point between the client and the framework. However, when Hadoop is running a job, a number of TaskTrackers, considered as servers, not only execute tasks based on the instruction from the JobTracker, but also handle data movement between the two phases of MapReduce.

B. Hadoop Configurability

There are more than 100 parameters available for users to manipulate in the Hadoop MapReduce framework. Depending on the way in which they make an impact on the performance of MapReduce applications, these parameters are generally divided into three groups: core parameters, MapReduce-relevant parameters and DFS-relevant parameters (DFS stands for Distributed File System). The Hadoop framework uses configuration files for setting the values of these parameters in each group. Section III-D3a shows the parameters we manipulated in this research.

- **Core parameters:** These are used for defining the most important features of a MapReduce cluster. The parameters in this group are associated only to the cluster itself such as where the temporary data is stored, how large the buffer size is, and what the threshold of shuffle group is, among others.

- **MapReduce-relevant parameters:** The parameters in this group are relevant to the MapReduce procedure: some of them have a direct effect only on the Map phase or Reduce phase, while others may have an effect on both phases.

- **DFS-relevant parameters:** The parameters, such as the one specifying how many replicas should be stored, belong to DFS group.

C. PPABS Architecture and Solution Overview

Our overall approach to the automated configuration management of a Hadoop cluster is based on a machine learning phase, which then is used for self-tuning. The machine learning phase requires training. Based on the learned knowledge, the system makes effective configuration decisions for a new, incoming job and applies these configuration decisions to automate the entire process thereby relieving the application developers from these challenges. The overall design and workflow of the PPABS systems shown in Figure 2 can be viewed as comprising two major parts: the Analyzer and the Recognizer. The Analyzer is based on history and is a completely offline step; the Recognizer however runs each time when a client submits a new job and hence is semi-online.

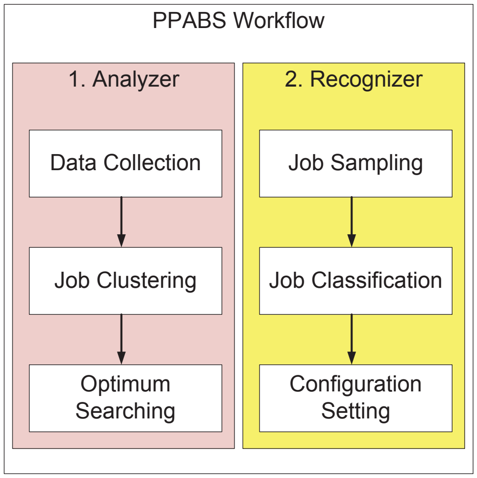


Fig. 2. PPABS Architecture

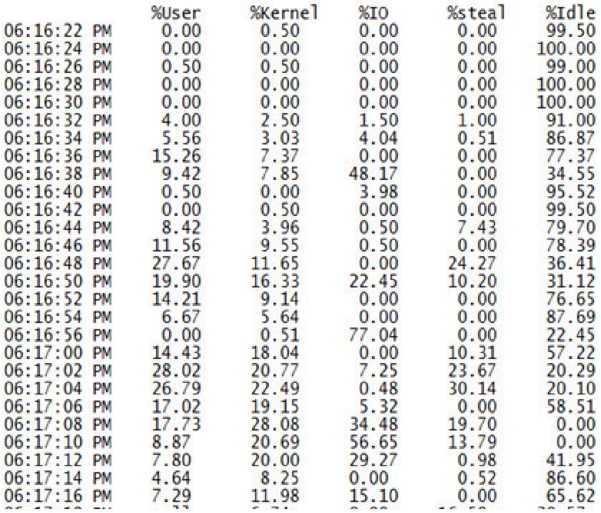
D. PPABS Analyzer Phase

There are three steps in the Analyzer phase: Data Collection (Section III-D1), Job Clustering (Section III-D2) and

Optimum Searching (Section III-D3). In the first step, we collect data from previous jobs and model a job's performance pattern along several attributes and their values. Using these attributes, in the second step we group previous jobs using modifications we made to the well-known *k-means* ++ clustering algorithm [17]. Subsequently we search the configuration parameters space to find the desired solution for each cluster with our modified Simulated Annealing algorithm.

1) Collecting History Data of Job Performance: Based on insights gained from related scientific works [11], [13], we arrived at an important conclusion: MapReduce applications can be classified into a finite set of categories or equivalence classes, where each equivalence class illustrates similar CPU and I/O utilization patterns among MapReduce applications belonging to that class. Moreover, through our investigations we have also found that different sizes of input data will affect the performance pattern differently to some degree. Therefore, it is necessary to develop a more accurate method, beyond simply CPU and I/O utilization, to define the performance characteristics. In our case, therefore, we have redefined the performance model and developed a new solution, which can not only eliminate this effect caused by different data sizes, but also describe the jobs' performance based on their features, for example, whether they are CPU-bound or IO-bound.

In this new model, we considered the entire computer system over a period of time, which is also our sampling time interval and is attributed to three resources: CPU, memory and disk. Each of these resources is responsible for a time interval in this entire sampling period. For instance, if a sampling period is 2 seconds, then it is of interest to us to know the amount of time that the computer spends on CPU, memory and disk in this period. Considering that the CPU subsystem is either running in kernel mode or user mode all the time, we also divide it into two parts: kernel and user.



```
                %User    %Kernel    %IO     %steal    %Idle
06:16:22 PM     0.00      0.50      0.00     0.00     99.50
06:16:24 PM     0.00      0.00      0.00     0.00    100.00
06:16:26 PM     0.50      0.50      0.00     0.00     99.00
06:16:28 PM     0.00      0.00      0.00     0.00    100.00
06:16:30 PM     0.00      0.00      0.00     0.00    100.00
06:16:32 PM     4.00      2.50      1.50     1.00     91.00
06:16:34 PM     5.56      3.03      4.04     0.51     86.87
06:16:36 PM    15.26      7.37      0.00     0.00     77.37
06:16:38 PM     9.42      7.85     48.17     0.00     34.55
06:16:40 PM     0.50      0.00      3.98     0.00     95.52
06:16:42 PM     0.00      0.50      0.00     0.00     99.50
06:16:44 PM     8.42      3.96      0.50     7.43     79.70
06:16:46 PM    11.56      9.55      0.50     0.00     78.39
06:16:48 PM    27.67     11.65      0.00    24.27     36.41
06:16:50 PM    19.90     16.33     22.45    10.20     31.12
06:16:52 PM    14.21      9.14      0.00     0.00     76.65
06:16:54 PM     6.67      5.64      0.00     0.00     87.69
06:16:56 PM     0.00      0.51     77.04     0.00     22.45
06:17:00 PM    14.43     18.04      0.00    10.31     57.22
06:17:02 PM    28.02     20.77      7.25    23.67     20.29
06:17:04 PM    26.79     22.49      0.48    30.14     20.10
06:17:06 PM    17.02     19.15      5.32     0.00     58.51
06:17:08 PM    17.73     28.08     34.48    19.70      0.00
06:17:10 PM    8.87      20.69     56.65    13.79      0.00
06:17:12 PM    7.80      20.00     29.27     0.98     41.95
06:17:14 PM    4.64       8.25     0.00      0.52     86.60
06:17:16 PM    7.29      11.98     15.10     0.00     65.62
```

Fig. 3. Snippet of Performance Analysis Data for Word Count Example

Figure 3 shows an example of the kinds of data we collect during the training phase of PPABS. As shown, we have gathered a five-dimensional statistical data from jobs that have been executed for training purposes. However, since our sampling technique is time-series based, it is obviously impossible to get the same running time from different jobs; for instance, it may take 5 minutes to complete job A while it could take 10 minutes or even longer to complete another. In fact, the length of data gathered always varies from job to job. In our case, a reasonable solution to solve this issue of data heterogeneity and ensuring that we get the same length of time series data is to use sampling as suggested in [18]. Thus, in this stage, the collected data of each job is converted into normalized sets so that all the sets are ultimately of the same length.

Using this information, the performance model for a specified time interval can be described by Equation 1.

$$P[x] = Kernel[x] + User[x] + Idle[x] + IO[x] + Steal[x] \quad (1)$$

where, P is the performance model and x is used to indicate the index of the sampling interval, *e.g.*, $P[3]$ means the performance in the third interval. More details of each variable can be found in Table I. Since Equation 1 represents only one time interval, the entire running performance of a Hadoop MapReduce job of which the running time can be divided into N intervals, is captured in Equations 2 and 3.

TABLE I. Description of Variables in the Performance Model

Name	Description
Kernel	The amount of time CPU spends in the kernel mode
User	The amount of time CPU spends in the user mode
Steal	The amount of time CPU spends in involuntary wait
IO	The amount of time the system spends on IO
Idle	The amount of time when the system is idle

$$P[k] = P_M[k] \bigcup \overline{P}_W[k] \quad (2)$$

$$P[Total] = \sum P[k] \quad (3)$$

Note that the master node is responsible for collecting data and assigning tasks to the slave nodes, however, slave nodes only execute their own tasks based on the instructions from the master. Therefore, on one hand, the performance of master node and the performance of slave nodes are significantly different when running a MapReduce job, while on the other hand, the difference among the slave nodes is very slight. However, because the size of a Hadoop MapReduce cluster is often large, it is not feasible to use all the utilization data gathered from each slave node. To solve this problem by processing the gathered data, we came up with Equation 2 for the k^{th} interval in which P_M is the performance of the master node in a Hadoop MapReduce cluster, and $\overline{P}_W$ is the average performance of all worker nodes in this cluster. The total performance over all the intervals is shown in Equation 3.

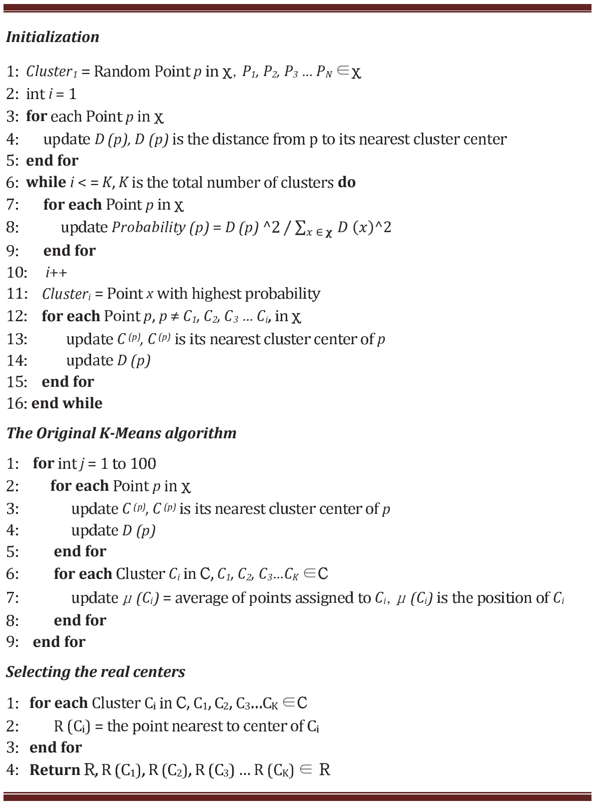
2) Job Clustering: Since our goal is to identify the optimal configuration settings of a MapReduce cluster for each MapReduce application, no matter whether the job is a totally new one or the one used in the training phase, it is very important to group all previously executed jobs used in the training phase based on their performance pattern so that our system can recognize and classify a new unknown job with the group rules and make sure an optimal solution of this group is loaded correctly. Therefore a clustering[2] algorithm is needed in this grouping situation. Several clustering algorithms are frequently used in the field of Data Mining [19]. After conducting a comparison on their cluster models and use cases, we determined that *k-means++* [17], which is a popular Centroid-based Clustering algorithm, is an appropriate choice for our research.

k-means++ belongs to the *k-means* class of algorithms for which the goal is to improve the clustering performance of the original *k-means*. In the original *k-means,* a D-dimensional vector is considered as a point and the goal is to automatically partition N input points into K clusters in which each point belongs to a cluster with the shortest distance from the point itself to the center of this cluster. In our case, since the collected performance data of a MapReduce job is a multi-dimensional vector (see Figure 3), it is also appropriate to be regarded as a point. However, one major disadvantage is that the clusters found by the *k-means* algorithm could be arbitrarily bad compared to the desired solution.

Compared to the original algorithm, the *k-means++* algorithm specifies the cluster initialization stage before proceeding with the standard *k-means* clustering algorithm. This improvement not only addresses the obstacle we mentioned above but also guarantees finding a solution that is $O(log\ k)$ competitive to the optimal one.

However, one drawback of the *k-means++* algorithm for our context is that the cluster centers obtained from it are very likely to be points which cannot be mapped to realistic and feasible MapReduce application configurations. Thus, applying the *k-means ++* algorithm as is makes no sense for this step of our PPABS system. We have therefore modified it with a *selection stage* to ensure that the centers we finally find are real, feasible points. The details of the modified algorithm are described in Figure 4.

In the modified *k-means++* algorithm, first we initialize the clusters by randomly selecting their centers (see Initialization phase). Then we iteratively update these clusters by reassigning points to them. Finally we select the real cluster centers by approximating them to the nearest point of the virtual center for each cluster (see Line 2 in Selecting Real Clusters). Each real cluster center we find from this Job Clustering step is eligible to represent the cluster itself, which also represents an equivalence class of MapReduce applications that have similar performance patterns that get assigned to the same cluster.



```
Initialization
1: $Cluster_1$ = Random Point $p$ in $\chi$, $P_1, P_2, P_3 \ldots P_N \in \chi$
2: int $i$ = 1
3: for each Point $p$ in $\chi$
4:     update $D(p)$, $D(p)$ is the distance from p to its nearest cluster center
5: end for
6: while $i < = K$, $K$ is the total number of clusters do
7:     for each Point $p$ in $\chi$
8:         update $Probability(p) = D(p)$ ^2 / $\sum_{x \in \chi} D(x)$^2
9:     end for
10:    $i$++
11:    $Cluster_i$ = Point $x$ with highest probability
12:    for each Point $p$, $p \neq C_1, C_2, C_3 \ldots C_i$, in $\chi$
13:        update $C^{(p)}$, $C^{(p)}$ is its nearest cluster center of $p$
14:        update $D(p)$
15:    end for
16: end while

The Original K-Means algorithm
1: for int $j$ = 1 to 100
2:     for each Point $p$ in $\chi$
3:         update $C^{(p)}$, $C^{(p)}$ is its nearest cluster center of $p$
4:         update $D(p)$
5:     end for
6:     for each Cluster $C_i$ in C, $C_1, C_2, C_3 \ldots C_K \in$ C
7:         update $\mu(C_i)$ = average of points assigned to $C_i$, $\mu(C_i)$ is the position of $C_i$
8:     end for
9: end for

Selecting the real centers
1: for each Cluster $C_i$ in C, $C_1, C_2, C_3 \ldots C_K \in$ C
2:     R ($C_i$) = the point nearest to center of $C_i$
3: end for
4: Return R, R ($C_1$), R ($C_2$), R ($C_3$) ... R ($C_K$) $\in$ R
```

Fig. 4. Modified K-Means++ Clustering Algorithm

3) Searching for the Optimum Configuration: In the previous step we find several "center" MapReduce applications, which are essentially the centers of the clusters found using the clustering technique shown in Section III-D2. Each such center is representative of all the applications that belong to that equivalence class and hence the optimum configuration for that center application applies to all applications belonging to that class. Our job now is to find an optimum Hadoop configuration per equivalence class. The details of the searching algorithm are presented, which comprises two parts: selecting the parameters and conducting a search.

a) Parameter Selection: Recall that each configuration parameter may have some impact on the global performance of a MapReduce application executing in the cluster. Even though some parameters have Boolean values, the types of most parameters are still Integer or Double. Therefore, assuming there are 100 parameters and the average number of their possible values is 1,000, the size of the search space becomes 100,000. In this situation, the searching time in total is 20,000,000 seconds if the average running time for a MapReduce application is 200 seconds (in most cases, the running time could be longer than 1,000 seconds).

It is unrealistic and expensive for us to use all parameters in this research. Thus, our strategy to decrease the size of the parameter space consists of three steps: (1) For parameters that have binary values, we just simply either set their values to the default or the values recommended by Apache Hadoop group [20], (2) For others, following the suggestions from related research on Hadoop parameters [11], [21], [22], we compared the parameters with each other based on their impact on the performance of a MapReduce cluster and only select the most important ones, and (3) For the parameters we selected from Step (2), instead of selecting all possible values from the original range, we use the optimal range of their values described in [23]. Subsequently, the list of parameters being

[2] Not to be confused with the term Hadoop cluster. Here the term clustering is taken from the data mining literature.

selected to build the search space is as shown in Table II.

TABLE II. DESCRIPTION OF SELECTED PARAMETERS

Name	Default Value	Optimal Range	Description
io.sort.mb	100	100 ∼ 300	The size of buffer when sorting files
io.sort.factor	10	50 ∼ 100	The number of streams while sorting
io.sort.spill.percent	0.8	0.5 ∼ 0.8	The soft threshold to decide whether spill contents to disk
mapred.tasktracker.map.tasks.maximum	2	1 ∼ 4	The maximum number of Map tasks running on a node
mapred.tasktracker.reduce.tasks.maximum	2	1 ∼ 4	The maximum number of Reduce tasks running on a node
mapred.child.java.opts	200m	200 ∼ 1,000m	Java opts for the children processes
mapred.reduce.parallel.copies	5	6 ∼ 12	The number of parallel transfers running in Reduce phase
dfs.block.size	64m	128 ∼ 640m	The size of block in file system

b) Searching Algorithm: The analysis above shows that the size of the entire search space has shrunk significantly. Now our parameter space can be modeled mathematically as a two-dimensional vector S in which every parameter is a one-dimensional vector $\overrightarrow{S[i]}, i = 1, 2, 3 \cdots N$. In addition, the length of $\overrightarrow{S[i]}$ depends on how many possible values could be set for the i^{th} parameter. For example, the length of $\overrightarrow{S[6]}$, for the parameter `mapred.reduce.parallel.copies` is 6 while the length of $\overrightarrow{S[7]}$ for the parameter `dfs.block.size` is 9 if we decide to change it in 64 unit increments.

Thus, our goal of finding the optimal configuration settings has been converted to searching for the optimal combination of these vectors so that the output, which is the execution time for a MapReduce job, of this combination is as small as possible. This problem now becomes a Combinatorial Optimization problem [24], a subject that is aimed at searching an optimal solution from a finite set of objects. To address the challenges stemming from having to conduct an exhaustive search, an alternative metaheuristic method is Local Search.

There are several well-defined methods within Local Search, such as Hill Climbing, Tabu Search and Simulated Annealing. The drawback with Hill Climbing is called premature convergence, which means that this greedy algorithm is very likely to find the nearest local optimum with low quality [25]. On the other hand, even though Tabu Search can prevent this disadvantage by maintaining a Tabu list to record the previous tries, it is not a reasonable method because it is too time consuming. In Tabu Search, every neighboring candidate should be tried so that this algorithm can choose the best one to navigate. Whereas in our case, using Tabu Search means to run a MapReduce job several times with every possible configuration settings in each step; it is definitely unrealistic and inefficient even with the reduced set of parameters we have chosen.

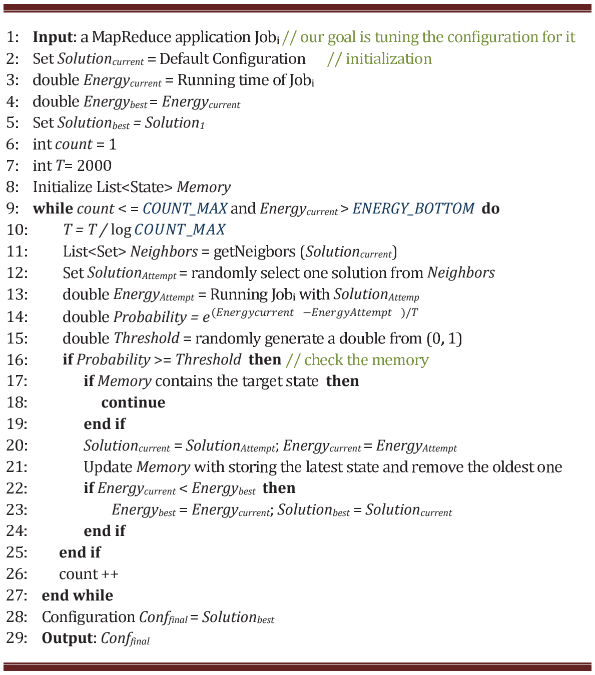
Compared to these two algorithms described above, Simulated Annealing is more appealing for our case to search the optimal solution for MapReduce cluster configuration. The main reason is that Simulated Annealing prevents low quality local optimum by occasionally accepting a solution even if it may be worse than the current one with a probability-based mechanism [26], [27]. However the shortcoming of the standard Simulated Annealing method is that it is memoryless, *i.e.*, it could simply repeat the previous track without "remembering" that it has done the same step before. For the purpose of preventing our system from entering this situation, we have modified the original algorithm by combining it with a concept from Tabu Search. That is to say, we add a memory structure to the standard Simulated Annealing to record the latest status in our approach.

Before introducing the details of the modified Simulated Annealing algorithm, it is necessary to describe its terminology and how we combine it with our research. The following additional concepts are part of our modified algorithm:

- **Energy:** It is the criterion to decide whether a candidate solution is good, which is based on the running time of a MapReduce application with a configuration setting.

- **Neighbors:** They represent the states that this algorithm could "jump" to from the current state. In our system, these are the nearest configuration settings we could change from the current one. For instance, if the current solution vector is $[S[0][x], S[1][y], S[2][z] \cdots S[M][\alpha]]$, then its neighbors can be $[S[0][x \pm 1], S[1][y], S[2][z] \cdots S[M][\alpha]], [S[0][x], S[1][y \pm 1], S[2][z] \cdots S[M][\alpha]] \cdots [S[0][x], S[1][y], S[2][z] \cdots S[M][\alpha \pm 1]]$.

- **Probability and Temperature:** This criterion decides whether an attempt will actually be selected.



```
Input: a MapReduce application Job_i // our goal is tuning the configuration for it
Set $Solution_{current}$ = Default Configuration     // initialization
double $Energy_{current}$ = Running time of Job_i
double $Energy_{best}$ = $Energy_{current}$
Set $Solution_{best}$ = $Solution_1$
int count = 1
int T= 2000
Initialize List<State> Memory
while count < = COUNT_MAX and $Energy_{current}$ > ENERGY_BOTTOM do
    T = T / log COUNT_MAX
    List<Set> Neighbors = getNeigbors ($Solution_{current}$)
    Set $Solution_{Attempt}$ = randomly select one solution from Neighbors
    double $Energy_{Attempt}$ = Running Job_i with $Solution_{Attemp}$
    double Probability = $e^{(Energycurrent\ -EnergyAttempt\ )/T}$
    double Threshold = randomly generate a double from (0, 1)
    if Probability >= Threshold then // check the memory
        if Memory contains the target state then
            continue
        end if
        $Solution_{current}$ = $Solution_{Attempt}$; $Energy_{current}$ = $Energy_{Attempt}$
        Update Memory with storing the latest state and remove the oldest one
        if $Energy_{current}$ < $Energy_{best}$ then
            $Energy_{best}$ = $Energy_{current}$; $Solution_{best}$ = $Solution_{current}$
        end if
    end if
    count ++
end while
Configuration $Conf_{final}$ = $Solution_{best}$
Output: $Conf_{final}$
```

Fig. 5. Modified Simulated Annealing Algorithm

The PPABS system implemented the modified Simulated Annealing algorithm, shown in Figure 5, to find the optimal parameter settings for each "center" MapReduce application. This algorithm initializes variables such as Temperature, Current Energy and Memory first, and then it iteratively searches the parameter space to find candidate solutions. In this case, whether a candidate solution is good or bad depends on its performance, which is the execution time of running a MapReduce application with the candidate configuration settings. When the iteration ends, this algorithm returns the best solutions for each "center" MapReduce application. The PPABS system then generates the configuration files based on the best solutions and saves these configuration files into a configuration library for future use by the Recognizer phase.

E. PPABS Recognizer Phase

The Recognizer phase also consists of three steps: Job Sampling (Section III-E1), Job Classification (Section III-E2) and Configuration Setting (Section III-E3). When a new job is submitted by the client, the PPABS system samples this job by running it with only part of its input data at first. Based on the job profile gathered from the first step, in the Job Classification step the system determines the equivalence class to which this unknown job belongs to. After completing these steps above, the Recognizer then loads the configuration files corresponding to the identified equivalence class and runs the submitted job with its entire input data set. Section III-E4 discusses the cost model for the Recognizer phase.

1) Job Sampling: One major advantage of any distributed file system is that it uses objects to associate logical paths with physical addresses so that it becomes possible for us to break data with very large size into smaller parts, and then store them in distributed manner while maintaining a unified logical path for this data. Similarly, HDFS, the Hadoop Distributed File System, is an implementation of the Scalable DFS for Hadoop MapReduce framework. Since the entire file system is broken into blocks, any very large data set submitted is also stored in distributed blocks. This feature of HDFS can be exploited to sample a job by using data only from a few blocks instead of the entire data set.

Moreover, typically a MapReduce job consists of two parts: the job itself and data. After comparing these two parts we have found that the size of the job (*i.e.*, size of the executable) is much smaller than the size of data for most MapReduce jobs. Take WordCount as an example; the size of this job's codes is only a few kilobytes, whereas the size of the input data set can become as large as hundreds of terabytes. Moreover, since the performance pattern of a MapReduce job is closely related to the job itself instead of the size of data set [8], we believe that sampling the job, which means to run a newly submitted job with only a small part of input data and using the default configurations, is reasonable for us to understand the performance pattern of this job and classify it into one of the previously identified equivalence classes, and thereby tune its parameters. This is precisely the approach we use in PPABS when a new job is submitted to our system.

2) Job Classification: While we are running a new job with part of its input data, we also collect the performance data from the Job Sampling step as above. Similar to the training process described in Section III-D1, the data gathered from the sampling step is a multidimensional time series. Since we used "center" to describe the performance patterns of a group of MapReduce jobs in Section III-D2, the procedure of Pattern Recognition is converted to a problem of finding the nearest center of a point if we model this new job that must be classified as a point which is newly added to the clustering space. Therefore, the algorithm we use is very simple as shown in Figure 6.

```
**Input**: Point $P_{unknown}$
double $D_{current}$ = the distance between $P_{\mathrm{unknown}}$ and $C_1$
Cluster $C_{short} = C_1$; double $D_{min} = D_{current}$
**for each** Cluster $C_i$ in $\mathbf{C}$, $C_1, C_2, C_3 \ldots C_K \in \mathbf{C}$
  $D_{current}$ = compute the distance between $P_{\mathrm{unknown}}$ and $C_i$
  **if** $D_{current} < D_{min}$ **then**
    update $D_{min}$ and $C_{short}$
  **end if**
**end for**
**Output**: $C_{short}$
```

Fig. 6. Classifying an Incoming Unknown Job

3) Configuration Setting: After sampling and classifying an unknown incoming job, our system automatically loads the tuned configuration files from the configuration library and runs this job again with its entire data set. One issue with this solution is that if the number of executed jobs increases with the increasing number of submitted jobs, then it is necessary to re-cluster the jobs so that we can keep the centers updated. Nevertheless, if the clusters are recomputed, retuning our existing MapReduce configuration settings is also needed. This process, unfortunately, takes considerable time. Therefore, it becomes significantly crucial for us to maintain a balance between updating clusters and making the system stay at its current status. In this case, our approach is to set a counter which is added by one each time when a job is completed. With this mechanism, our PPABS system restarts if the counter reaches the threshold we initially set, otherwise this system just stays and uses the results it previously found.

4) Cost Model of the Recognizer and Impact on Application Performance: Understanding the cost of this semi-online Recognizer step is important. The three major steps in the Recognizer are Job Sampling, Job Classification and Configuration Setting. Different from the Analyzer, the Recognizer is a semi-online part of our PPABS system. Thus, the time spent on these three steps must be included in the total running time for a new job as shown in Equation 4.

$$T_{Total} = T_{Sampling} + T_{Classification} + T_{Setting} + T_{OptTotal} \tag{4}$$

If we assume the cost model of a MapReduce job as linear related to the size of data set, and the time spent on a MapReduce job running with its entire data set using the default configuration settings is $T_{Default}$, we can get Equation 5. In Equation 5, M is the total number of blocks used to store the total data set, while S is the number of blocks used to store the sampling part.

$$T_{Sampling} = T_{Default} * (S/M) \tag{5}$$

From our analysis we found that the time spent on the Job Classification step and Configuration Setting is so small compared to the time spent on other steps, that our cost model can be approximated as shown in Equation 6, where $T_{OptTotal}$ is the total running time of the new job with the optimum configuration settings using its entire data set.

$$T_{Total} = T_{Default} * (S/M) + T_{OptTotal} \tag{6}$$

The performance improvement stemming from our PPABS system can be determined from how much T_{Total} is an improvement over $T_{Default}$. Theoretically, the percentage improvement can be modeled as shown in Equation 7, which can be derived as shown in the Proof below.

$$\triangle I = \frac{\frac{M-S}{M} * T_{Default} - T_{OptTotal}}{T_{Default}} * 100\% \tag{7}$$

Proof:

$$\begin{aligned}\triangle T &= T_{default} - T_{Total} \\ &= T_{default} - (T_{default} * (S/M) + T_{OptTotal}) \\ &= T_{default} * (1 - (S/M)) - T_{OptTotal} \\ &= \frac{M-S}{M} * T_{default} - T_{OptTotal}\end{aligned}$$

$$Since\ perf\ improvement\ is\ given\ by,$$

$$\triangle I = \frac{\triangle T}{T_{default}} * 100\%$$

$$Substituting\ for\ \triangle T\ gives\ Equation\ 7.$$

■

IV. Evaluating PPABS-generated Hadoop Configurations

This section describes results of experiments comparing the performance of MapReduce applications using Hadoop cluster configurations defined by PPABS versus the default ones.

A. Experimental Settings

We implemented and evaluated the PPABS system on a Hadoop MapReduce cluster that was deployed on the Amazon EC2 Web Service. This Hadoop cluster consists of five DataNodes, considered in the slave roles only, and one NameNode which serves is both the slave role and master role in our system. The details of these nodes are listed in Table III. The version of Hadoop we used is 1.0.4 and we have set the number of replicas to be 6 since there are 6 nodes in total in this cluster.

TABLE III. Description of Cluster Configuration

Node	Instance Type	CPU	Memory	Storage	Num
Name Node	M1 Medium	2 EC2 Compute Units	3.75 GB	300GB	1
Data Node	M1 Small	1 EC2 Compute Unit	1.7 GB	200GB	5

Since the Data Mining technique we used to profile and analyze the performance of MapReduce applications is $k-means++$, this clustering analysis algorithm needs to be trained first. Thus, it is necessary for us to decide which applications should be included in the training set of our PPABS system. In this experiment, the training set of MapReduce applications consists of three well-known sets that have predominantly being used in research: Hadoop Examples set, Hadoop Benchmarks set and HiBench, which is another benchmark set implemented by Intel. In total there are 48 applications in these three sets, however, we decided to select three most popular applications — WordCount, TeraSort and Grep — as the “unknown” and “incoming” applications to test the performance of our system. Therefore, the size of training set is 45.

Note that the training set may not be as large as some experiments in the field of Data Mining, however, in the field of MapReduce, we have plenty of reasons to believe that this set in our research is solid and large enough, especially compared to the related works, of which the number of applications used is usually less than 5. Moreover, in the step of Data Collection and Performance Analysis, or in the tuning step, we set the size of input data to be 1GB for each application.

B. Experimental Results

First, we present in Table IV the intermediate results generated by the Analyzer, which is the offline part of our PPABS system. We have tested the clustering algorithm in situations when we set $K = 3, 4, 5$ and then the size of each cluster and the average distance (normalized by percentage) between each point and the cluster center this point belongs to are also listed in the table. It can be seen from this table that the accuracy of the modified $k-means++$ algorithm we used to cluster MapReduce jobs increases as K increases, which describes the number of clusters. The reason why we did not set $K > 5$ in this experiment is because a large K will make the tuning step more complicated. Based on the observation, we decided to set $K = 4$ for the next step in this evaluation.

TABLE IV. Results of Job Clustering

Num of Clusters	Average Distance	Size of Cluster
K=3	21.3/100	[19, 12, 14]
K=4	15.7/100	[11, 15, 11, 8]
K=5	13.5/100	[12, 7, 9, 8, 9]

Next, to evaluate the performance of the Recognizer, we provide a comparison between the tuned configuration settings and the default cluster configuration settings for WordCount, TeraSort and Grep in Table V after they are submitted to our system. We can find that there is significant difference among the tuned configurations and the default one, for instance, the value of parameter `io.sort.mb` in each of the tuned configuration is twice as large as the default value. Moreover, from the output of the Recognizer, we also notice that the configuration files loaded for these three MapReduce applications we have submitted are also different in some way. Take for example the parameter `io.sort.factor` whose optimal value found by PPABS for TeraSort is larger than the one for WordCount, while the value of this parameter for Grep is only 30, which is smaller than the others.

TABLE V. COMPARISON OF MAPREDUCE CONFIGURATION SETTINGS FOR THE 3 APPLICATIONS

Name	Default	Word Count	Tera Sort	Grep
io.sort.mb	100	240	220	280
io.sort.factor	10	50	80	30
io.sort.spill.percent	0.8	0.67	0.6	0.8
mapred.tasktracker.map.-tasks.maximum	2	4	4	3
mapred.tasktracker.reduce.-tasks.maximum	2	2	3	2
mapred.child.java.opts	200m	500m	800m	800m
mapred.reduce.parallel.copies	5	8	10	8
dfs.block.size	64m	256m	256m	374m
mapred.map.output.compress	FALSE	TRUE	TRUE	TRUE

Finally, the performance of PPABS is evaluated as follows. We compared the execution time of WordCount with tuned configuration to the execution time with the default settings in Figure 7. Next, we compared the execution time of the other two applications with the tuned configuration to execution time with the default one in Figures 8 and 9. Moreover, in order to evaluate whether our system indeed improves the performance of MapReduce jobs when the input data is very large, the size of input data is set to 1GB, 5GB, and 10GB. We repeated this evaluation several times to discard outliers and consider the average behavior.

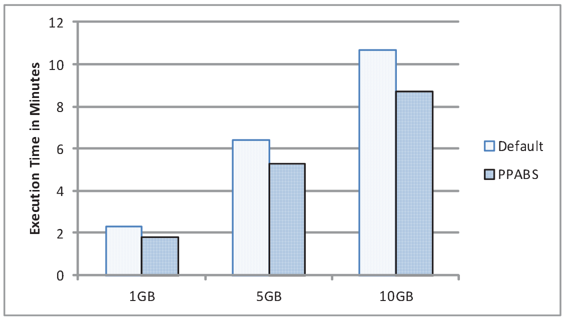


Fig. 7. Performance Comparison for Word Count

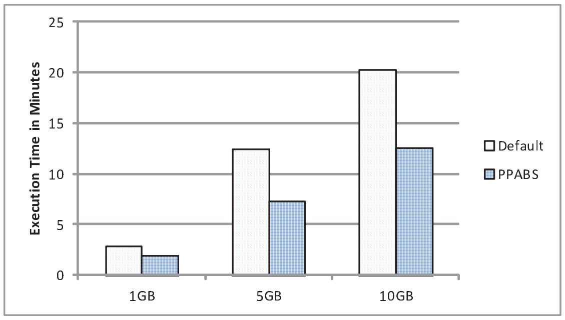


Fig. 8. Performance Comparison for Tera Sort

These figures demonstrate that the PPABS system improves the performance of MapReduce cluster. Besides, when the size of input data set is relatively small such as only 1GB, this improvement is not very obvious. While when the size of input data becomes larger, for example, more than 5GB, the performance of all three Hadoop applications is significantly improved by our system. How much improvement in the performance also varies by the characteristics of different jobs: when the size of input data set is 10 GB, the average execution time for TeraSort decreases by 38.4%, however, in the same situation, the average execution time for WordCount decreases by only 18.7%. The reason for this difference is because our PPABS recognizes the incoming unknown jobs by classifying and assigning each of them to a cluster, but the distance between each job, which can be modeled as a point, and its cluster center varies by the job itself. Therefore, if a job is far from its cluster center, it is possible that its performance is not optimized as well as the performance of another job which is very close to its cluster center.

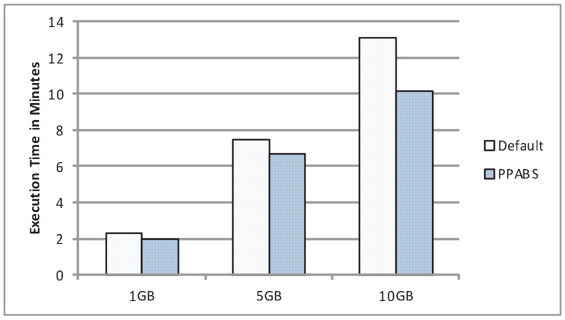


Fig. 9. Performance Comparison for Grep

V. CONCLUSIONS

This paper presented PPABS, which is a profiling and performance analysis based self-tuning system that is aimed at optimizing the configuration of the Hadoop MapReduce cluster. This system consists of two major parts: the Analyzer and the Recognizer. The former is called by PPABS before a new job is submitted. It analyzes and processes data gathered from the executed jobs and then uses a modified $k - means + +$ clustering algorithm to group these jobs based on their performance pattern into one of a predefined set of equivalence classes. A modified Simulated Annealing algorithm is presented to search for the optimal solutions for each "center" found from the Job Clustering step. The latter is called when a new job enters the system. It samples the new job by running it only with a small part of its entire data set at first. Then the Recognizer compares the new job's profile with profiles of the "centers" and classifies this new-incoming job into one group we previously found. The last step for the Recognizer is selecting the tuned configuration files to load and run the new job with updated configuration settings.

We implemented and evaluated the benefits of PPABS by running real MapReduce applications on Amazon EC2. The experimental results showed the effectiveness of our approach in improving the performance of MapReduce applications using our self-tuned configurations compared to the execution time of the same jobs running with the default configuration. We observed that the improvement by our PPABS system is significant when the size of input data set is large.

The following limitations exist in our work requiring additional research, however, we believe the presented work is a step in the right direction. First, we have used the term "optimal" in a somewhat loose sense in that we have not determined the structure of the objective function nor do we have a closed form formula for the optimization function. Moreover, we have not investigated if the objective function contains any discontinuities in which case local searches may be useless. Second, we have not conducted any sensitivity analysis to determine the impact of individual parameters so that the less interesting parameters can be eliminated from the optimization problem thereby reducing the number of dimensions. We will leverage our prior experience [28] in understanding these effects. Third, so far we focused only on CPU and I/O activities; other systemic effects also need to be considered. Fourth, the size of our experiments is small; in reality MapReduce applications require a large number of compute resources.

Our future work in this area centers around two enhancements. We plan to add more MapReduce applications into our training set so that we can let our system "remember" more executed jobs as history data. We plan to optimize the Job Sampling step with appropriate tradeoffs: on one hand, the time of sampling should be as short as possible; on the other hand, we have to make sure the sampled performance pattern can describe the features of each job as well. The Amazon EC2 testbed we used comprised DataNodes that were homogeneous but the NameNode was different in configuration. Our future work will experiment with additional heterogeneity and scale in the system.

ACKNOWLEDGMENTS

This work was supported in part by NSF CAREER Award CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We thank the anonymous reviewers for their excellent feedback that helped strengthen the paper.

REFERENCES

[1] C. Ordonez, I.-Y. Song, and C. Garcia-Alvarado, "Relational versus Non-relational Database Systems for Data Warehousing," in *Proceedings of the ACM 13th international workshop on Data warehousing and OLAP*, ser. DOLAP '10. New York, NY, USA: ACM, 2010, pp. 67–68. [Online]. Available: http://doi.acm.org/10.1145/1871940.1871955

[2] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, 2007, pp. 13–24.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[4] ——, "MapReduce: A Flexible Data Processing Tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629198

[5] J. Dean, "Experiences with MapReduce, An Abstraction for Large-scale Computation," in *PACT: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, vol. 16, no. 20, 2006, pp. 1–1.

[6] A. Bialecki, M. Cafarella, D. Cutting, and O. OŠMalley, "Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware," *Wiki at http://lucene. apache. org/hadoop*, 2005.

[7] K. Wang, X. Lin, and W. Tang, "Predator — An Experience Guided Configuration Optimizer for Hadoop MapReduce," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, 2012, pp. 419–426.

[8] S. Babu, "Towards Automatic Optimization of MapReduce Programs," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 137–142. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807150

[9] G.-z. SUN, F. XIAO, and X. XIONG, "Study on Scheduling and Fault Tolerance Strategy of MapReduce [J]," *Microelectronics & Computer*, vol. 9, p. 053, 2007.

[10] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 29–42.

[11] K. Kambatla, A. Pathak, and H. Pucha, "Towards Optimizing Hadoop Provisioning in the Cloud," in *Proc. of the First Workshop on Hot Topics in Cloud Computing*, 2009, p. 118.

[12] C. Tian, H. Zhou, Y. He, and L. Zha, "A Dynamic MapReduce Scheduler for Heterogeneous Workloads," in *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*, 2009, pp. 218–224.

[13] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010, pp. 94–103.

[14] N. B. Rizvandi, J. Taheri, R. Moraveji, and A. Y. Zomaya, "On Modelling and Prediction of Total CPU usage for Applications in MapReduce Environments," in *Proceedings of the 12th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 414–427. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33078-0_30

[15] P. Lama and X. Zhou, "AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud," in *Proceedings of the 9th international conference on Autonomic computing*, ser. ICAC '12. New York, NY, USA: ACM, 2012, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/2371536.2371547

[16] 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 *Fifth Biennial Conference on Innovative Data Systems Research (CIDR)*, Alisomar, CA, USA, Jan. 2011, pp. 261–272.

[17] D. Arthur and S. Vassilvitskii, "k-means++: tThe Advantages of Careful Seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. [Online]. Available: http://dl.acm.org/citation.cfm?id=1283383.1283494

[18] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast Subsequence Matching in Time-series Databases," in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '94. New York, NY, USA: ACM, 1994, pp. 419–429. [Online]. Available: http://doi.acm.org/10.1145/191839.191925

[19] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data Clustering: A Review," *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/331499.331504

[20] "Hadoop Common Core," hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml.

[21] H. Herodotou, "Hadoop Performance Models," *arXiv preprint arXiv:1106.0940*, 2011.

[22] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[23] S. Joshi, "Hadoop Tuning Guide," Advanced Micro Devices, Tech. Rep., Oct. 2012.

[24] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1982.

[25] J. M. Hinson and J. E. R. Staddon, "Matching, maximizing, and hill-climbing," *Journal of the Experimental Analysis of Behavior (JEAB)*, vol. 40, no. 3, pp. 321–331, Nov. 1983.

[26] P. J. Van Laarhoven and E. H. Aarts, *Simulated Annealing*. Springer, 1987.

[27] E. Aarts, J. Korst, and W. Michiels, "Simulated Annealing," in *Search Methodologies*, E. Burke and G. Kendall, Eds. Springer US, 2005, pp. 187–210. [Online]. Available: http://dx.doi.org/10.1007/0-387-28356-0_7

[28] C. Yilmaz, A. Porter, A. S. Krishna, A. Memon, D. C. Schmidt, and A. Gokhale, "Reliable Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 124–141, Feb. 2007.