CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **26**:2005–2026
Published online 6 March 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3229

SPECIAL ISSUE PAPER

Parameterizable benchmarking framework for designing a MapReduce performance model[§]

Zhuoyao Zhang[1], Ludmila Cherkasova[2,*,†] and Boon Thau Loo[1]

[1]*Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA*
[2]*Hewlett-Packard Labs, Palo Alto, CA, USA*

SUMMARY

In MapReduce environments, many applications have to achieve different performance goals for producing time relevant results. One of typical user questions is how to estimate the completion time of a MapReduce program as a function of varying input dataset sizes and given cluster resources. In this work, we offer a novel performance evaluation framework for answering this question. We analyze the MapReduce processing pipeline and utilize the fact that the execution of map (reduce) tasks consists of specific, well-defined data processing phases. Only map and reduce functions are custom, and their executions are user-defined for different MapReduce jobs. The executions of the remaining phases are *generic* (i.e., defined by the MapReduce framework code) and depend on the amount of data processed by the phase and the performance of the underlying Hadoop cluster. First, we design *a set of parameterizable microbenchmarks* to profile the execution of generic phases and to derive *a platform performance model* of a given Hadoop cluster. Then, using the job past executions, we summarize job's properties and performance of its custom map/reduce functions in a compact job profile. Finally, by combining the knowledge of the job profile and the derived platform performance model, we introduce *a MapReduce performance model* that estimates the program completion time for processing a new dataset. The proposed benchmarking approach derives an accurate performance model of Hadoop's generic execution phases (*once*), and then, this model is *reused* for predicting the performance of different applications. The evaluation study justifies our approach and the proposed framework: We use a diverse suite of 12 MapReduce applications to validate the proposed model. The predicted completion times for most experiments are within 10% of the measured ones (with a worst case resulting in 17% of error) on our 66-node Hadoop cluster. Copyright © 2014 John Wiley & Sons, Ltd.

Received 16 December 2013; Revised 19 February 2014; Accepted 22 January 2014

KEY WORDS: MapReduce processing pipeline; Hadoop cluster; benchmarking; job profiling; performance modeling

1. INTRODUCTION

MapReduce and Hadoop represent an economically compelling platform for efficient large-scale data processing and cost-effective analytics over 'Big Data' in the enterprise. Hadoop is increasingly being deployed in enterprise private clouds and also offered as a service by public cloud providers (e.g., Amazon's Elastic Map-Reduce). In both usage scenarios, there are open performance analysis and prediction issues to estimate the appropriate resources needed for timely processing of the customer applications. An emerging new trend is a shift toward using Hadoop for support of latency-sensitive applications, for example, personalized recommendations [2], advertisement placement [3], and real-time web indexing [4]. These MapReduce applications are typically a part of an

*Correspondence to: Ludmila Cherkasova, Hewlett-Packard Labs, Palo Alto, CA, USA.
[†]E-mail: lucy.cherkasova@hp.com
[§]This is an extended version of the short ICPE'2013 paper [1].

elaborate business pipeline, and they have to produce results by a certain time deadline. There is a slew of interesting applications associated with live business intelligence that require (soft) completion time guarantees. When a Hadoop cluster is shared by multiple users, a key question is how to automatically tailor and control resource allocations to different applications for achieving their performance goals and (soft) completion time deadlines.

While there were quite a few research efforts to design different models and approaches for predicting performance of MapReduce applications [5–9], this question still remains a challenging research problem. Some of the past modeling efforts aim to predict the job completion time by analyzing the execution times' distribution of map and reduce tasks [8], and propose a simple performance model that is based on estimating the lower and upper bounds on the job completion time. However, to predict the application completion time for processing a larger dataset, an additional effort is required to derive the scaling factors by performing multiple runs of this application with different size datasets and creating *per application* scaling model [7, 9]. Related efforts [5–7] aim to perform a more detailed (and more expensive) job profiling at a level of phases that comprise the execution of map and reduce tasks. Note that such phase profiling is carried out per each job, and the derived model cannot be reused for different MapReduce jobs. These approaches require significant job profiling and modeling efforts.

In this work, we offer a new approach for designing a MapReduce performance model that can efficiently predict the completion time of a MapReduce application for processing a new dataset as a function of allocated resources. The proposed solution utilizes a useful rationale of the detailed *phase* profiling method [5] in order to accurately estimate durations of map and reduce tasks, and then applies a simple, bounds-based analytical models designed in [8]. However, our new framework proposes a very *different approach* to estimate the execution times of these job phases. We observe that the executions of map and reduce tasks consist of specific, well-defined data processing phases and distinguish two types among them. Note that only map and reduce functions are custom, and their computations are user-defined for different MapReduce jobs. The executions of the remaining phases are *generic*, that is, strictly regulated and defined by the Hadoop processing framework. The execution time of each generic step depends on the amount of data processed by the phase and the performance of underlying Hadoop cluster. In the earlier papers [5–7], profiling is performed for all the phases (including the generic ones) for each application separately. Then, the profiled measurements are used for predicting a job completion time. In our work, we design *a set of parameterizable microbenchmarks* to measure generic phases and to derive *a platform performance model* of a given Hadoop cluster. The new framework consists of the following key components:

- *A set of parameterizable microbenchmarks* to measure and profile different phases of the MapReduce processing pipeline of a given Hadoop cluster. These microbenchmarks might be executed in a small cluster deployment, for example, having only five nodes. This test cluster employs the same nodes hardware and the same Hadoop configuration as a production cluster, and its advantage is that the benchmarking process does not interfere with the production jobs. The input parameters of microbenchmarks impact the amount of data processed by different phases of map and reduce tasks. We concentrate on profiling of general (noncustomized) phases of the MapReduce processing pipeline. Intuitively, the executions of these phases on a given Hadoop cluster are similar for different MapReduce jobs, and the phase execution time mostly depends on the amount of processed data. By running a set of diverse benchmarks on a given Hadoop cluster, we collect a useful training set that characterizes the execution times of different phases while processing different amounts of data.
- *A platform profile and a platform performance model* that characterize the execution time of each *generic phase* during MapReduce processing. Using the created training set (i.e., the collected platform profile) and a robust linear regression, we derive a platform performance model that estimates each phase duration as a function of processed data.
- *A compact job profile* for each MapReduce application of interest that is extracted from the past job executions. It summarizes the job's properties and performance of its custom map and reduce functions. This job profile captures the inherent application properties such as the job's map (reduce) selectivity, that is, the ratio of the map (reduce) output to the map (reduce) input.

DOI: 10.1002/cpe

This parameter describes the amount of data produced as the output of the user-defined map (reduce) function, and therefore, it helps in predicting the amount of data processed by the remaining generic phases.

- *A MapReduce performance model* that combines the knowledge of the extracted job profile and the derived platform performance model to estimate the completion time of the programs for processing a new dataset.

The proposed evaluation framework aims to *divide* (i) the performance characterization of the underlying Hadoop cluster and (ii) the extraction of specific performance properties of different MapReduce applications. It aims to derive *once* an accurate performance model of Hadoop's generic execution phases as a function of processed data, and then *reuse* this model for characterizing performance of generic phases across different applications (with different job profiles).

We validate our approach using a set of 12 realistic applications executed on a 66-node Hadoop cluster. We derive the platform profile and platform performance model on a small five-node test cluster and then use this model together with extracted job profiles to predict the job completion time on the 66-node Hadoop cluster. Our evaluation shows that the designed platform model accurately characterizes different execution phases of MapReduce processing pipeline of a given cluster. The predicted completion times of most considered applications (10 out of 12) are within 10% of the measured ones.

The rest of this paper is organized as follows. Section 2 provides the MapReduce background and presents our phase profiling approach. Section 3 describes microbenchmarks and the objectives for their selection. Sections 4 and 5 introduce main performance models that form a core of the proposed framework. Section 6 evaluates the accuracy and effectiveness of our approach. Section 7 discusses challenges of modeling the applications with a data skew in the shuffle/reduce stages. Section 8 outlines the related work. Section 9 presents a summary and future work directions.

2. MAPREDUCE PROCESSING PIPELINE

In the MapReduce model [10], the main computation is expressed as two user-defined functions: *map* and *reduce*. The map function takes an input pair and produces a list of intermediate key/value pairs. The intermediate values associated with the same key k_2 are grouped together and then passed to the reduce function. The reduce function takes intermediate key k_2 with a list of values and processes them to form a new list of values.

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$reduce(k_2, list(v_2)) \rightarrow list(v_3)$$

MapReduce jobs are executed across multiple machines: The map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*.

For detailed examples of MapReduce processing (WordCount, etc.), we refer users to broadly available tutorials and books on Hadoop (e.g., [11, 12]).

The execution of each map (reduce) task is composed of a specific, well-defined sequence of processing phases [13] as shown in Figure 1. Note that only map and reduce phases with customized map and reduce functions execute the user-defined pieces of code. The execution of the remaining phases is generic (i.e., defined by Hadoop code) and mostly depends on the amount of data flowing through a phase. Our goal is to derive a platform performance model that predicts a duration of each generic phase on a given Hadoop cluster platform as a function of processed data. In order to accomplish this, we run a set of microbenchmarks that create different amounts of data for processing per map (reduce) tasks and for processing by their phases. We profile the duration of each generic phase during the task execution and derive a function that defines a phase performance as a function of the processed data from the collected measurements. During the benchmarking process, we collect measurements of four generic phases of the map task execution and two generic phases of the reduce task execution.

DOI: 10.1002/cpe

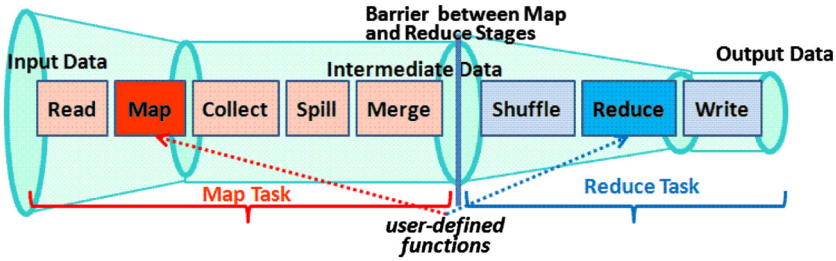

Figure 1. MapReduce processing pipeline.

Map task consists of the following generic phases:

1. *Read* phase—A map task typically reads a block (e.g., 64 MB) from the Hadoop distributed file system. However, written data files might be of arbitrary size, for example, 70 MB. In this case, there will be two blocks: one of 64 MB and the second of 6 MB, and therefore, map tasks might read files of varying sizes. There are many applications and workflows when map tasks read entire files or compressed files of varying sizes. We measure the duration of the read phase and the amount of data read by the map task.
2. *Collect* phase—This generic phase follows the execution of the map phase with a user-defined map function. We measure the time it takes to buffer map phase outputs into memory and the amount of generated intermediate data.
3. *Spill* phase—We measure the time taken to locally sort the intermediate data and partition them for the different reduce tasks, applying the combiner if available [‡] and then writing the intermediate data to local disk as spilled files. The size and the number of spilled files depend on the size of in-memory sort buffer defined by the Hadoop configuration.
4. *Merge* phase—We measure the time for merging different spill files into a single spill file for each destined reduce task.

Reduce task consists of the following generic phases:

1. *Shuffle* phase—We measure the time taken to transfer intermediate data from map tasks to the reduce tasks and merge-sort them together. We combine the shuffle and sort phases because in the Hadoop implementation, these two subphases are interleaved. The processing time of this phase depends on the amount of intermediate data destined for each reduce task and the Hadoop configuration parameters. In our testbed, each JVM[§] (i.e., a map/reduce slot) is allocated 700 MB of RAM. Hadoop sets a limit (~46% of the allocated memory) for in-memory sort buffer. The portions of shuffled data are merge-sorted in memory, and a spill file ($700 \times 0.46 = 322$ MB) is written to disk. After all the data are shuffled, Hadoop merge-sorts first 10 spilled files and writes them in the new sorted file. Then, it merge-sorts the next 10 files and writes them in the next new sorted file. At the end, it merge-sorts these new sorted files. Thus, we can expect that the duration of the shuffle phase might be approximated with a different linear function when the intermediate dataset per reduce task is larger than 3.22 GB (3.22 GB = 322 MB×10) in our Hadoop cluster. For a differently configured Hadoop cluster, this threshold can be similarly determined from the cluster configuration parameters.
2. *Write* phase—This phase follows the execution of the reduce phase that executes a custom reduce function. We measure the amount of time taken to write the reduce output to the Hadoop distributed file system.

Note that in platform profiling, we do not include phases with user-defined map and reduce functions. However, we do need to profile these custom map and reduce phases for modeling the execution of given MapReduce applications:

[‡]A combine operation applies the job reduce function at the map task side to optimize the amount of intermediate data during the shuffle phase. From the viewpoint of the reduce task, this combined data contain the same information as the original map task output, but it leads to far fewer pairs output to disk, read from disk, and transferred over network.
[§]JVM stands for Java Virtual Machine.

DOI: 10.1002/cpe

- *Map* (*Reduce*) phase—We measure a duration of the entire map (reduce) function and the number of processed records. We normalize this execution time to estimate a processing time per record.

Apart from the phases described earlier, each executed task has a constant overhead for setting and cleaning up. We account for these overheads separately for each task.

2.1. *Profiling techniques*

For accurate performance modeling it is desirable to minimize the overheads introduced by the additional monitoring and profiling technique. There are *two* different approaches for implementing phase profiling.

1. Currently, Hadoop already includes several *counters* such as the number of bytes read and written. These counters are sent by the worker nodes to the master periodically with each heartbeat. We *modified the Hadoop code* by adding counters that measure durations of the six generic phases to the existing counter reporting mechanism. We can activate the subset of desirable counters in the Hadoop configuration for collecting the set of required measurements.
2. We also implemented the alternative profiling tool inspired by Starfish [5] approach based on *BTrace*—a dynamic instrumentation tool for Java [14]. BTrace intercepts class byte codes at run-time based on event-condition-action rules and injects byte codes for the associated actions. This approach does have a special appeal for production Hadoop clusters because it has a zero overhead when monitoring is turned off. However, in general, the dynamic instrumentation overhead is much higher compared with adding new Hadoop counters directly in the source code. We instrumented selected Java classes and functions internal to Hadoop using BTrace in order to measure the time taken for executing different phases.

Figure 2 explains the main components and steps of our framework.

We execute a set of microbenchmarks (described in Section 3) and measure the durations of six generic execution phases for processing different amount of data: *read, collect, spill,* and *merge* phases for the map task execution, and *shuffle* and *write* phases in the reduce task processing. These measurements form the *platform profile* to represent the MapReduce processing pipeline of the underlying Hadoop cluster. This profiling is performed on a small test cluster (five nodes in our experiments) with the same hardware and configuration as the production cluster. While for these experiments both profiling approaches can be used, the Hadoop counter-based approach is preferable because of its simplicity and low overhead, and that the modified Hadoop version can be easily deployed in this test environment.

Then, using the created platform profile, we derive with linear regression technique a platform performance model.

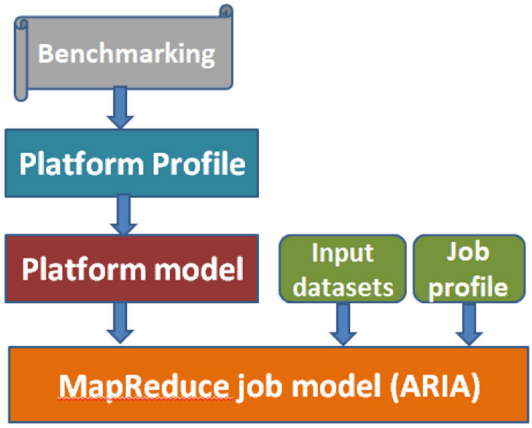

Figure 2. Overall framework.

DOI: 10.1002/cpe

The *MapReduce performance model* needs additional measurements that characterize the execution of user-defined map and reduce functions of a given job as well as the information on the input data size. For profiling the map and reduce phases of the given MapReduce jobs in the production cluster, we apply our alternative profiling tool that is based on the BTrace approach. Remember that this approach does not require Hadoop or application changes and can be switched on for profiling a targeted MapReduce job of interest. Because we only profile map and reduce phase executions, the extra overhead is relatively small.

For the benchmarking experiments, we use a Hadoop cluster with a default First In First Out (FIFO) scheduler. Similarly, in the evaluation experiments (presented in Section 6), we use the Hadoop cluster with FIFO scheduler as well. The goal of this paper is to design an accurate performance model that can be used by a new SLO-driven[¶] Hadoop scheduler (like the ARIA project [8]). Applying the designed performance model, such a new job scheduler controls the amount of allocated resources (i.e., map and reduce slots) for achieving the predefined job completion time.

3. MICROBENCHMARKS AND PLATFORM PROFILE

Our goal is to create a set of benchmarks that generate different amounts of data for processing by different phases. Note that there are a few distinct places in the MapReduce processing pipeline where the amount of processed data can change as a result of different input datasets or application semantics (as it is shown in Figure 1). The input data size per map task might be different depending on the Hadoop configuration and the data format used. The amount of processed data may change after the user-defined map function in the map phase. The next possible change is the shuffle phase: The amount of data per reduce tasks depends on the number of reduce tasks used in the job configuration. Finally, the user-defined reduce function may change the amount of the output data during the reduce phase.

Therefore, we generate a set of parameterizable microbenchmarks to characterize execution times of generic phases for processing different data amounts on a given Hadoop cluster by varying the following parameters:

1. *Input size per map task* $\left(M^{inp}\right)$: This parameter controls the size of the input read by each map task. Therefore, it helps to profile the read phase durations for processing different amount of data.
2. *Map selectivity* $\left(M^{sel}\right)$: This parameter defines the ratio of the map output to the map input. It controls the amount of data produced as the output of the map function and therefore directly affects the collect, spill, and merge phase durations in the map task. Map output determines the overall amount of data produced for processing by the reduce tasks, and therefore impacting the amount of data processed by shuffle and reduce phases and their durations.
3. *A number of map tasks* N^{map}: Increasing this parameter helps to expedite generating the large amount of intermediate data per reduce task.
4. *A number of reduce tasks* N^{red}: Decreasing this parameter helps to expedite the training set generation with the large amount of intermediate data per reduce task.

Thus, each microbenchmark MB_i is parameterized as

$$MB_i = \left(M_i^{inp}, M_i^{sel}, N_i^{map}, N_i^{red}\right)$$

Each created benchmark uses input data consisting of 100 byte key/value pairs generated with *TeraGen* [15], a Hadoop utility for generating synthetic data. The map function simply emits the input records according to the specified map selectivity for this benchmark. The reduce function is defined as the identity function. Most of our benchmarks consist of a specified (fixed) number of map and reduce tasks. For example, we generate benchmarks with 40 map and 40 reduce tasks each for execution in our small cluster deployments with five worker nodes

[¶]SLO stands for Service Level Objective.

DOI: 10.1002/cpe

Row *number* j	Data *MB* $Data_1$	*Read msec* Dur_1
1	16	2010
2	16	2020
...	...	...

Row *number* j	Data *MB* $Data_2$	*Collect msec* Dur_2
1	8	1210
2	8	1350
...	...	...

Figure 3. A fragment of a platform profile for *read* and *collect* phases.

(see setup details in Section 6). We run benchmarks with the following parameters: $M^{inp} = \{2, 4, 8, 16, 32, 64 \text{ MB}\}$; $M^{sel} = \{0.2, 0.6, 1.0, 1.4, 1.8\}$. For each value of M^{inp} and M^{sel}, a new benchmark is executed. We also use benchmarks that generate special ranges of intermediate data per reduce task for accurate characterization of the shuffle phase. These benchmarks are defined by $N^{map} = \{20, 30, ..., 150, 160\}$; $M^{inp} = 64$ MB, $M^{sel} = 5.0$ and $N^{red} = 5$, which result in different intermediate data size per reduce tasks ranging from 1 to 12 GB.

We generate the platform profile by running a set of our microbenchmarks on the small five-node test cluster that is composed of the worker nodes with the same hardware as a given production Hadoop cluster. Moreover, we use the same Hadoop configuration in the test and production clusters.

While executing each microbenchmark, we gather durations of generic phases and the amount of processed data for all executed map and reduce tasks. A set of these measurements defines the platform profile that is later used as the training data for a platform performance model: We collect durations of *six* execution phases that reflect essential steps in processing of map and reduce tasks on the given platform:

- *Map task processing*: In the collected platform profiles, we denote the measurements for phase durations and the amount of processed data for the read, collect, spill, and merge phases as $(Dur_1, Data_1)$, $(Dur_2, Data_2)$, $(Dur_3, Data_3)$, and $(Dur_4, Data_4)$, respectively.
- *Reduce task processing*: In the collected platform profiles, we denote phase durations and the amount of processed data for shuffle and write as $(Dur_5, Data_5)$ and $(Dur_6, Data_6)$, respectively.

Figure 3 shows a small fragment of a collected platform profile as a result of executing the microbenchmarking set. There are six tables in the platform profile, one for each phase. Figure 3 shows fragments for the read and collect phases. There are multiple map and reduce tasks that process the same amount of data in each microbenchmark. This is why there are multiple measurements in the profile for processing the same data amount.

4. PLATFORM PERFORMANCE MODEL

Now, we describe how to create a platform performance model M_{Phases} that characterizes the phase execution as a function of processed data. To accomplish this goal, we need to find the relationships between the amount of processed data and durations of different execution phases using the set of collected measurements. Therefore, we build six submodels $M_1, M_2, \ldots, M_5$, and M_6 that define the relationships for read, collect, spill, merge, shuffle, and write, respectively, of a given Hadoop cluster. To derive these submodels, we use the collected platform profile (collected as a result of executing a set of microbenchmarks on a given Hadoop cluster or its small deployment; see Figure 3).

In what follows, we explain how to build a submodel M_i, where $1 \leqslant i \leqslant 6$. By using measurements from the collected platform profiles, we form a set of equations that express a phase duration as a linear function of processed data (we will evaluate in Section 6 whether a linear function may serve as a good approximation). Let $Data_i^j$ be the amount of processed data in the row j of platform profile with K rows. Let Dur_i^j be the duration of the corresponding phase in the same row j. Then, using linear regression, we solve the following sets of equations (for each $i = 1, 2, \cdots, 6$):

$$A_i + B_i \cdot Data_i^j = Dur_i^j, \quad \text{where } j = 1, 2, \cdots, K \tag{1}$$

To solve for (A_i, B_i), one can choose a regression method from a variety of known methods in the literature (a popular method for solving such a set of equations is a nonnegative least squares

DOI: 10.1002/cpe

regression). With ordinary least squares regression, a few bad outliers can significantly impact the model accuracy, because it is based on minimizing the overall absolute error across multiple equations in the set. To decrease the impact of occasional bad measurements and to improve the overall model accuracy, we employ robust linear regression [16] (which is typically used to avoid a negative impact of a small number of outliers).

Let $\left(\hat{A}_i, \hat{B}_i\right)$ denote a solution for the equation set (1). Then, $M_i = \left(\hat{A}_i, \hat{B}_i\right)$ is the submodel that defines the duration of execution phase i as a function of processed data. The platform performance model is $M_{Phases} = (M_1, M_2, \ldots, M_5, M_6)$.

In addition, we have implemented a test to verify whether two linear functions may provide a better fit for approximating different segments of training data (ordered by the increasing data amount) instead of a single linear function derived on all data. As we will see in Section 6, the shuffle phase is better approximated by a combination of two linear functions over two data ranges: less than 3.2 GB and larger than 3.2 GB (confirming the conjecture that was discussed in Section 2).

5. MAPREDUCE PERFORMANCE MODEL

In this section, we describe a MapReduce performance model that is used for predicting a completion time of a given MapReduce job. We do it by applying the *analytical model* designed and validated in ARIA [8]. The proposed performance model utilizes the knowledge about average and maximum map (reduce) task durations for computing the lower and upper bounds on the job completion time as a function of allocated resources (map and reduce slots). Typically, the estimated completion time based on the average of the lower and upper bounds serves as a good prediction: It is within 10% of the measured completion time as shown in [8].

To apply the analytical model, we need to estimate the map and reduce tasks distributions to approximate the average and maximum completion time of the map and reduce tasks. The overall flow of computation is shown in Figure 4.

To achieve this, for a given MapReduce job, a special compact job profile is extracted automatically from the previous run of this job. It includes the following metrics:

- map/reduce selectivity that reflects the ratio of the map/reduce output size to the map/reduce input size; and
- processing time per record of map/reduce function.

We also collect characteristics of the input dataset such as (1) the number of data blocks and (2) the average/maximum data block size (both in bytes and in the number of records). Such information defines the number of map tasks and the average/maximum input data per map task.

For map tasks, the task completion time is estimated as a sum of durations of all the map stage phases. The generic phase durations for read, collect, spill, and merge are estimated according to the platform model $M_{Phases} = (M_1, M_2, \ldots, M_5, M_6)$ by applying a corresponding function to the data amount processed by the phase. Note that the data size for the collect, spill, and merge phases is estimated by applying the map selectivity to the map task input data size (this information is available in the extracted job profile). The map phase duration depends on the user-defined map

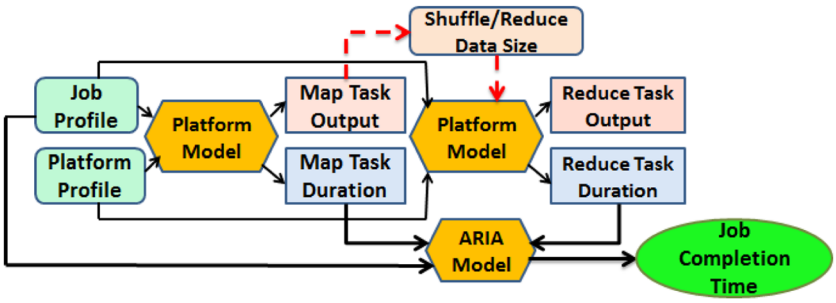

Figure 4. MapReduce performance model.

DOI: 10.1002/cpe

functions and is estimated according to the number of input records and the map function processing time per record (again available from the extracted job profile). Depending on the average and maximum input data size, we estimate the average and maximum map task durations, respectively.

For reduce tasks, the task completion time is estimated as a sum of durations of the shuffle, reduce, and write phases. Similarly to the map task computation described earlier, the shuffle and write phase durations are estimated according to the platform model and the phase input data size. The reduce function duration is estimated according to the number of reduce records and the reduce function processing time per record available from the job profile.

The input size for the shuffle phase depends on the overall data amount of the map outputs and the number of reduce tasks. Thus, the input size of the reduce task can be estimated as follows:

$$Data_{shuffle} = (M^{inp} \times M^{sel} \times N^{map})/N^{red} \tag{2}$$

where we assume that each map output is uniformly distributed across the reduce tasks.

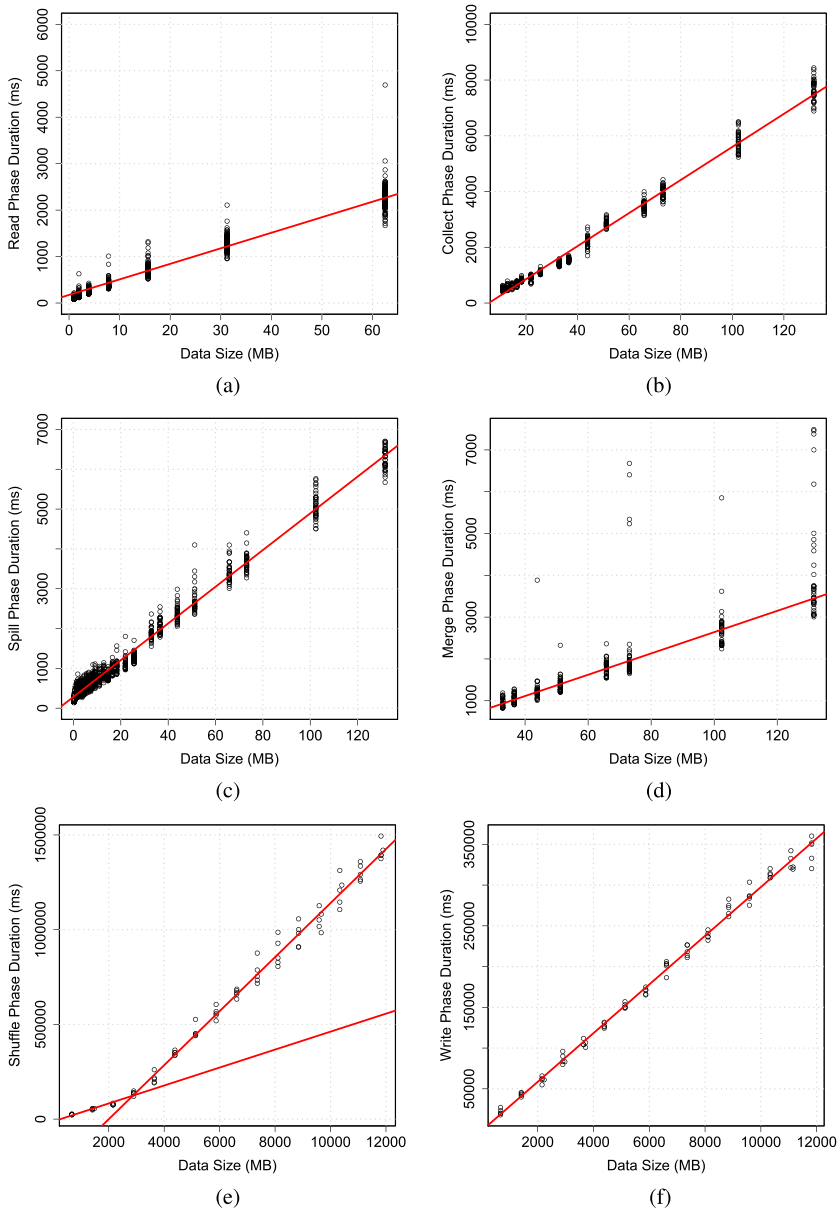

Figure 5. Benchmark results: (a) read; (b) collect; (c) spill; (d) merge; (e) shuffle; and (f) write.

DOI: 10.1002/cpe

6. EVALUATION

6.1. Experimental testbed

All experiments are performed on 66 HP DL145 G3 machines. Each machine has four AMD 2.39 GHz cores, 8 GB RAM, and two 160 GB 7.2K rpm SATA hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. We used Hadoop 0.20.2 with two machines dedicated as the JobTracker and the NameNode, and remaining 64 machines as workers. Each worker is configured with two map and two reduce slots. The file system block size is set to 64 MB. The replication level is set to 3. We disabled speculative execution because it did not lead to significant improvements in our experiments.

6.2. Accuracy of the platform performance model

To profile the generic phases in the MapReduce processing pipeline of a given production cluster, we execute the designed set of microbenchmarks on the small five-node test cluster that uses the same hardware and configuration as the large 66-node production cluster. Figure 5 shows the relationships between the amount of processed data and the execution durations of different phases for a given Hadoop cluster. Figure 5(a)–(f) reflects the platform profile for six generic execution phases: read, collect, spill, and merge phases of the map task execution, and shuffle and write phases in the reduce task.

Each graph has a collection of dots that represent phase duration measurements (Y-axes) of the profiled map (reduce) tasks as a function of processed data (X-axes). The line on the graph shows the linear regression solution that serves as a model for the phase. As we can see (visually), the linear regression provides a good solution for five out of six phases. As we expected, the shuffle phase is better approximated by a linear piecewise function composed of two linear functions (see a discussion about the shuffle phase in Section 2): one is for processing up to 3.2 GB of intermediate data per reduce task, and the second segment is for processing the datasets larger than 3.2 GB.

To validate whether our explanation on the shuffle phase behavior (provided in Section 2) is correct, we perform a set of additional experiments. We configured each JVM (i.e., a map/reduce slot) with 2 GB RAM (compared with the JVM with 700 MB of RAM used in previous experiments). As we explained earlier, Hadoop sets a limit ($\sim$ 46% of the allocated memory) for in-memory sort buffer. The portions of shuffled data are merge-sorted in memory, and a spill file (in the new case, $\sim$ 900 MB) is written to disk. After all the data are shuffled, Hadoop merge-sorts first 10 spilled

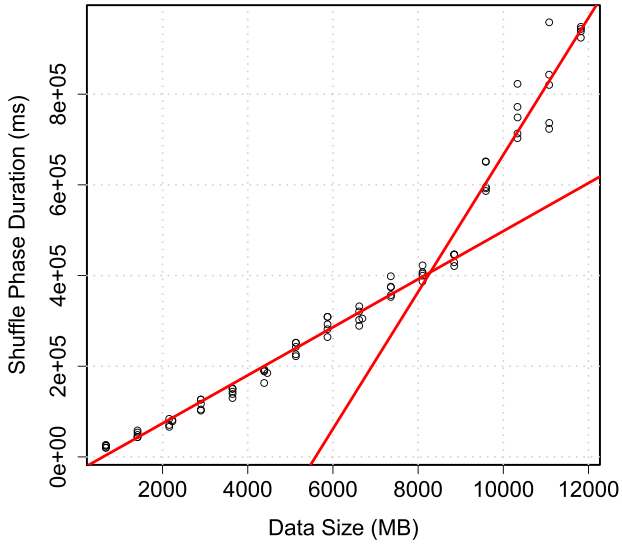

Figure 6. Shuffle phase model for Hadoop where each JVM (slot) is configured with 2 GB of memory.

DOI: 10.1002/cpe

files and writes them in the new sorted file. Then, it merge-sorts the next 10 files and writes them in the next new sorted file. Finally, at the end, it merge-sorts these new sorted files. Thus, we can expect that in the new configuration, the shuffle performance is significantly different for processing intermediate data large than 9 GB. Figure 6 indeed confirms our conjecture: shuffle performance changes for processing intermediate data large than 9 GB. Indeed, the shuffle phase performance is affected by the JVM memory size settings, and its performance can be more accurately approximated by a linear piecewise function.

In order to formally evaluate the accuracy and fit of the generated model M_{Phases}, we compute for each data point in our training dataset a *prediction error*. That is, for each row j in the platform profile, we compute the duration dur_i^{pred} of the corresponding phase i by using the derived model M_i as a function of data $Data^j$. Then, we compare the predicted value dur_i^{pred} against the measured duration dur_i^{measrd}. The relative error is defined as follows:

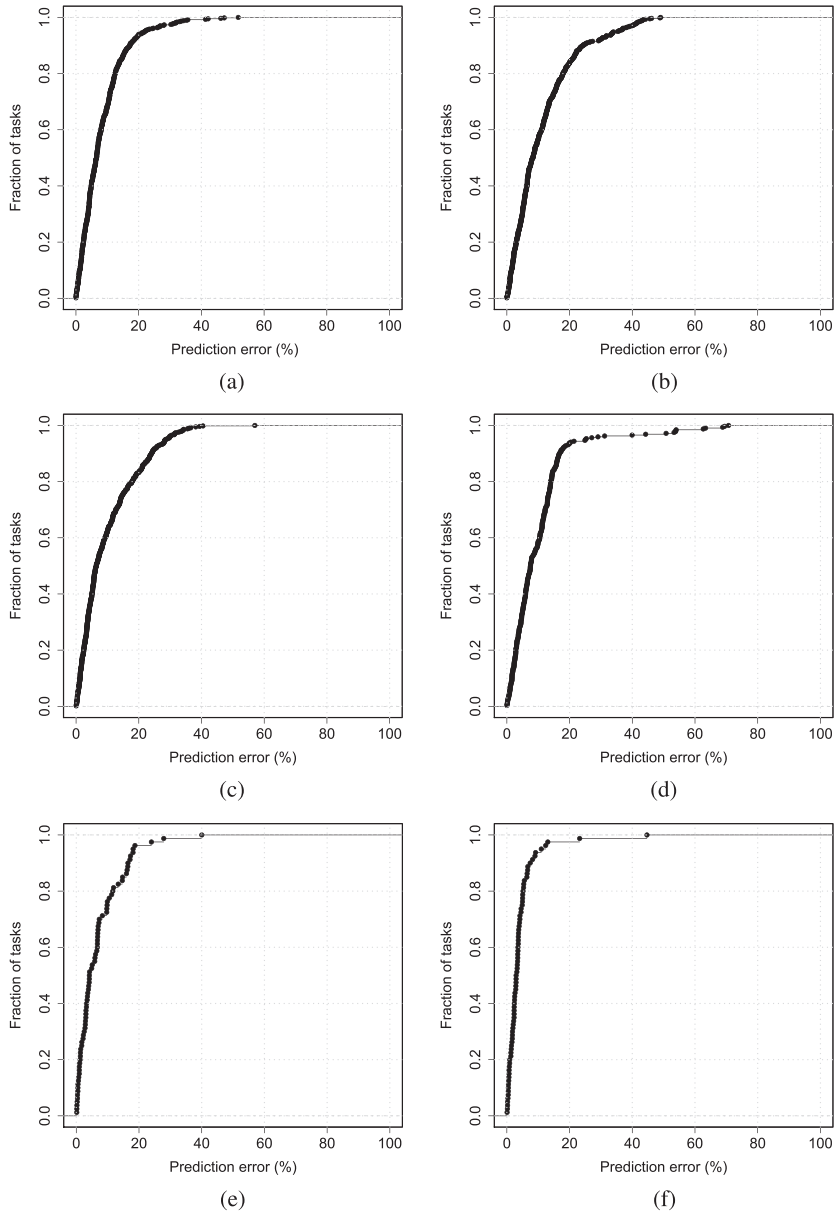

Figure 7. CDF of prediction error: (a) read; (b) collect; (c) spill; (d) merge; (e) shuffle; and (f) write.

DOI: 10.1002/cpe

Table I. Relative error distribution.

Phase	Error $\leqslant 10\%$	Error $\leqslant 15\%$	Error $\leqslant 20\%$
Read (%)	66	83	92
Collect (%)	56	73	84
Spill (%)	61	76	83
Merge (%)	58	84	94
Shuffle (%)	76	85	96
Write (%)	93	97	98

$$error_i = \frac{\left| dur_i^{measrd} - dur_i^{pred} \right|}{dur_i^{measrd}}$$

We calculate the relative error for all the data points in the platform profile. Figure 7 shows the CDF of relative errors for all six phases.

The CDF of relative errors proves that our performance model fits well to the experiment data.

Table I summarizes the relative errors for derived models of six generic processing phases. For example, for the read phase, 66% of map tasks have the relative error less than 10%, and 92% of map tasks have the relative error less than 20%. For the shuffle phase, 76% of reduce tasks have the relative error less than 10%, and 96% of reduce tasks have the relative error less than 20%. In summary, almost 80% of all the predicted values are within 15% of the corresponding measurements. It shows that the derived platform performance model (constructed with the linear regression technique and the set of equations based on the collected benchmark data as shown in Equation (1)) fits well these experimental data. The next step is to evaluate the prediction power and the accuracy of the designed model using the real MapReduce jobs.

6.3. Accuracy of the MapReduce performance model

To validate the accuracy of the proposed MapReduce performance model, we use it for predicting the completion times of the 12 applications made available by the Tarazu project [17].

Table II gives a high-level description of these 12 applications with the job settings (e.g., a number of map and reduce tasks). Applications 1, 8, and 9 process synthetically generated data, applications 2 to 7 use the Wikipedia articles dataset as input, while applications 10 to 13 use the Netflix movie ratings dataset. We present the results of running these applications with (i) *small input datasets* defined by parameters shown in columns 3–4 and (ii) *large input datasets* defined by parameters shown in columns 5–6, respectively.

These applications exhibit and represent different processing patterns. Such applications as *TeraSort, RankInvIndex, SeqCount, AdjList*, and *KMeans* are shuffle- and write-heavy: These applications process a similar amount of data along their entire MapReduce processing pipeline. While the other applications, such as *WordCount, Grep, HistMovies*, and *HistRatings*, quickly reduce the amount of data for processing during the map stage.

Figure 8 shows the comparison of the measured and predicted job completion times[‖] for the 12 applications (with a small input dataset) executed using a five-node test cluster. The graphs reflect that the designed MapReduce performance model closely predicts the job completion times. The measured and predicted durations are less than 10% for most cases (with 17% error being a worst case for WordCount and HistRatings). Note the split at Y-axes in order to accommodate a much larger scale for a completion time of the KMeans application.

The next question to answer is whether the platform performance model constructed using a small five-node test cluster can be effectively applied for modeling the application performance in the larger production clusters. To answer this question, we execute the same 12 applications (with a large

[‖]All the experiments are performed five times, and the measurement results are averaged. We also show the additional variance bar, that is, the minimal and maximal measurement values across the five runs. This comment applies to the results in Figures 8–14.

DOI: 10.1002/cpe

Table II. Application characteristics.

Application	Input data (type)	Input (GB) small	No. of Map, Reduce tasks	Input (GB) large	No. of Map, Reduce tasks
1. TeraSort	Synthetic	2.8	44, 20	31	495, 240
2. WordCount	Wikipedia	2.8	44, 20	50	788, 240
3. Grep	Wikipedia	2.8	44, 1	50	788, 1
4. InvIndex	Wikipedia	2.8	44, 20	50	788, 240
5. RankInvIndex	Wikipedia	2.5	40, 20	46	745, 240
6. TermVector	Wikipedia	2.8	44, 20	50	788, 240
7. SeqCount	Wikipedia	2.8	44, 20	50	788, 240
8. SelfJoin	Synthetic	2.1	32, 20	28	448, 240
9. AdjList	Synthetic	2.4	44, 20	28	508, 240
10. HistMovies	Netflix	3.5	56, 1	27	428, 1
11. HistRatings	Netflix	3.5	56, 1	27	428, 1
12. KMeans	Netflix	3.5	56, 16	27	428, 50

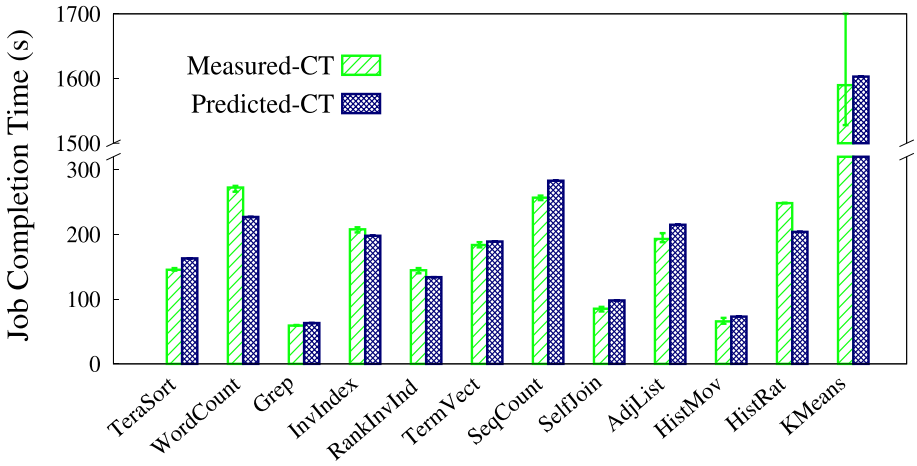

Figure 8. Predicted versus measured completion times of 12 applications on the small *five-node* test cluster.

input dataset) on the 66-node production cluster. Figure 9 shows the comparison of the measured and predicted job completion times for the 12 applications executed on the 66-node Hadoop cluster.

The graphs reflect that the designed MapReduce performance model (derived from the measurements collected at the small five-node test cluster) closely predicts the job completion times processed on the large Hadoop cluster. The predicted completion times closely approximate the measured ones: For 11 applications, they are less than 10% of the measured ones (a worst case is WordCount that exhibits 17% of error). Note the split at Y-axes for accommodating the KMeans completion time in the same figure. These results justify our approach for building the platform performance model by using a small test cluster. Running benchmarks on the small cluster significantly simplifies the approach applicability, because these measurements do not interfere with production workloads. The collected platform profile leads to a good quality platform performance model that can be efficiently used for modeling production jobs in the larger enterprise cluster.

To provide more details on the accuracy of the constructed platform performance model for predicting different generic phase durations, we further analyze the performance of two applications (randomly chosen from the set of studied 12 applications): TeraSort and InvertedIndex.

The TeraSort application splits the input dataset into 64 MB blocks and sorts them into a total order. The map and reduce functions of TeraSort simply emit the input records. The InvertedIndex application parses the input data and creates a mapping between each word and the corresponding documents. The map function reads each document in the input dataset and emits for each word

DOI: 10.1002/cpe

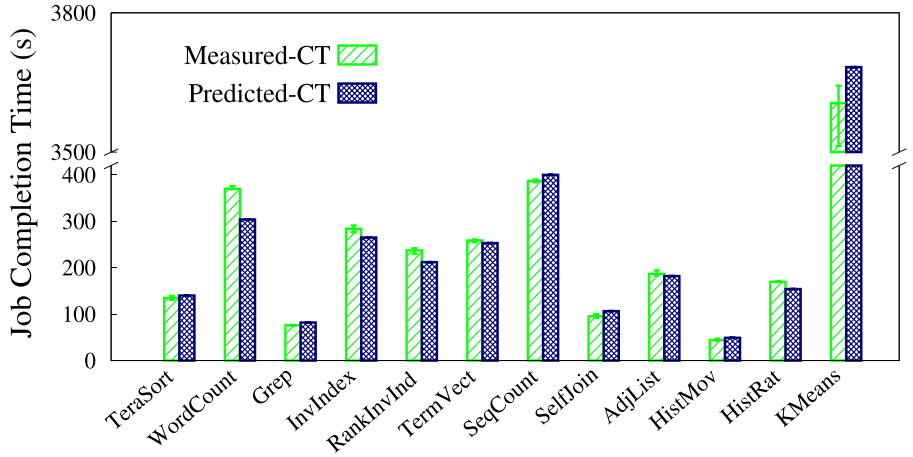

Figure 9. Predicted versus measured completion times of 12 applications (with a large input dataset) on the large *66-node* production cluster.

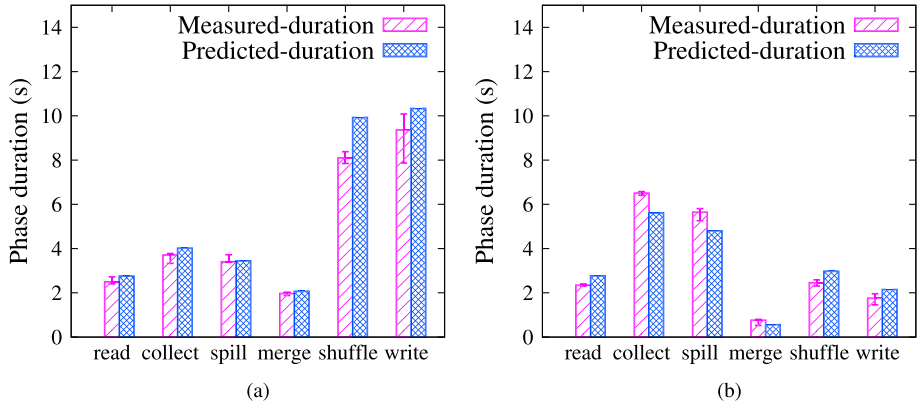

Figure 10. Predicted versus measured durations of the generic phases of the MapReduce processing pipeline on the small *five-node* test cluster: (a) TeraSort and (b) InvertedIndex.

a record that consists of a pair *(word, documentID)*. The reduce function aggregates the list of *documentID*s for each word.

We first execute and analyze these two applications on the five-node *test* cluster. The input dataset for TeraSort is synthetically generated with a size of 2.8 GB. The InvertedIndex application is executed on a 2.8 GB subset of Wikipedia data. The number of reduce tasks is fixed to 20 for both applications. Figure 10 compares the measured and predicted durations of the six generic phases.

The graphs reflect that the constructed performance model could accurately predict the durations of each phase as a function of the processed data. The differences between the measured and predicted durations are within 15% in most cases. Only for the merge phase of the InvertedIndex application, the difference is around 26%. Note that TeraSort and InvertedIndex have different map and reduce selectivities, and it is reflected in different data amounts processed by the five generic phases as observed in Figure 10. However, the read phase in both applications processes the same amount of data (64 MB), and the durations of read phases are similar for both applications.

In the next set of experiments, we aim to analyze in more detail whether the platform performance model constructed using a small five-node test cluster effectively predicts the phase performance of applications executed in the larger-size production clusters. To answer this question, we execute the same two applications TeraSort and InvertedIndex on the 66-node production cluster. The input size is scaled up to 31 GB for the TeraSort application and 50 GB for the InvertedIndex application. The

DOI: 10.1002/cpe

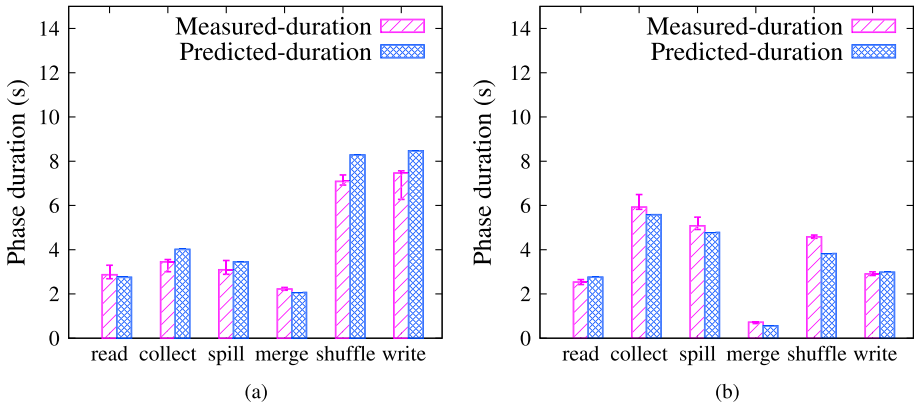

Figure 11. Predicted versus measured durations of the generic phases of the MapReduce processing pipeline on the large *66-node* production cluster: (a) TeraSort and (b) InvertedIndex.

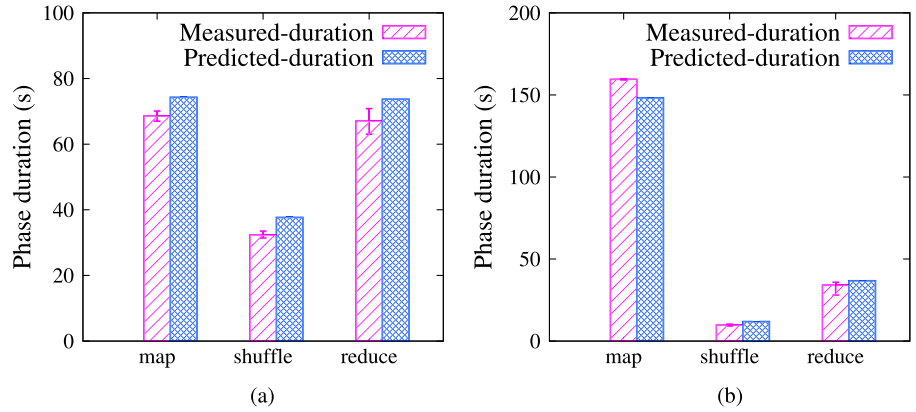

Figure 12. Predicted versus measured durations of different stages for (a) TeraSort and (b) InvertedIndex on the small *five-node* test cluster.

number of reduce tasks is configured to 240. Figure 11 shows the measured and predicted durations of six processing phases. The predicted phase durations closely approximate the measured ones. These results justify that the platform performance model derived from the small test cluster can be effectively applied to a larger production cluster with the same configurations.

We further demonstrate the accuracy of the proposed MapReduce performance model by comparing the measured and predicted durations of the map, shuffle, and reduce stages (as defined in ARIA work [8]) for the TeraSort and InvertedIndex applications. Note that the map stage defines the time for processing all the map tasks, the shuffle stage reflects the time required for shuffling the intermediate data to the reduce tasks, and the reduce stage reflects the time needed for processing all the reduce tasks. Therefore, showing the durations of the map, shuffle, and reduce stages for the studied applications provide the useful insights about where the time is spent during the job processing. Figures 12 and 13 show the comparison results for two sets of experiments when both applications are executed on the five-node test cluster and the 66-node production cluster, respectively.

The results show that the difference between the measured and predicted durations is within 10% for the map and reduce stages. The shuffle stage shows slightly larger differences with the maximum error value around 17%. However, because the shuffle stage durations constitute relatively small fractions in the overall completion time of both applications, the shuffle stage error has a limited impact on the accuracy of the predicted job completion times.

DOI: 10.1002/cpe

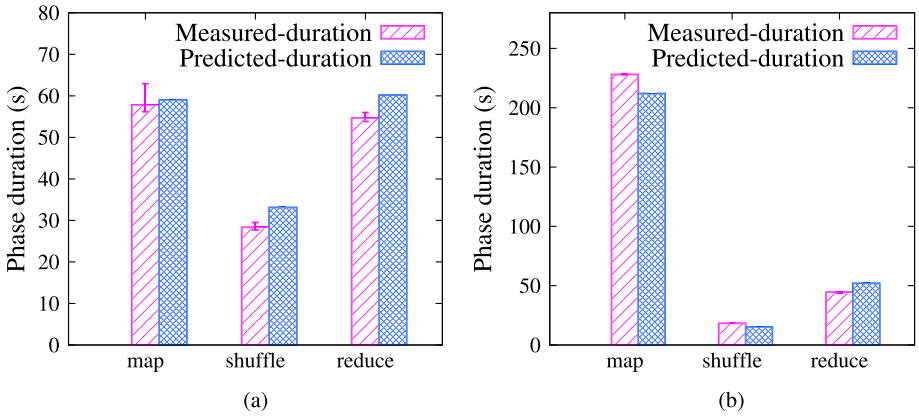

Figure 13. Predicted versus measured durations of different stages for (a) TeraSort and (b) InvertedIndex on the large *66-node* production cluster.

7. DISCUSSION AND FUTURE WORK

The proposed MapReduce performance model relies on the assumption that the intermediate data generated by the map stage are uniformly distributed across the reduce tasks. However, this assumption may not hold for some applications with skewed input/output data.

For example, the computation and the outcome of the map function in the KMeans application significantly depend on the specified number of reduce tasks. It defines the number of clusters in the KMeans clustering algorithm (i.e., the K value). In our experiments with 12 applications, we extract the job profiles by executing these applications on a small five-node test cluster and small input datasets (defined by parameters shown in columns 3–4 of Table II). We build a platform profile by benchmarking the same five-node test cluster. Then, we use these job profiles and the derived platform profile for predicting the application performance on the larger datasets (defined by parameters shown in columns 5–6 of Table II) and a large 66-node production cluster. Figure 9 shows the comparison of the measured and predicted job completion times for 12 applications executed on the 66-node Hadoop cluster. The predicted completion times closely approximate the measured ones: For 11 applications, they are less than 10% of the measured ones. In these experiments, we considered the KMeans application with $K = 50$, that is, the number of reduce tasks is set to 50. The performance prediction for the KMeans application with $K = 50$ is quite accurate.

However, the situation changes when we perform the same experiments for KMeans with $K = 16$. Table III shows the measured and predicted completion times for the *KMeans_16* and *KMeans_50* when these jobs are executed on the large 66-node cluster.

The measured completion time for *KMmeans_16* is 1910 s, while the predicted completion time is 1275 s (i.e., a prediction error of 33%). For a more detailed analysis of the KMeans execution time under different parameters, we break down its overall completion time into three main stages, that is, map, shuffle, and reduce stages, and compare their durations when it is configured with 16 and 50 reduce tasks. The results are illustrated in Figure 14.

We can see that the proposed model predicts accurately the duration of the map stage in both cases: the difference between the measured and predicted results is 3% and 4%, respectively. However, for KMeans with 16 reduce tasks, the model underestimates the shuffle and reduce stage durations. Because the shuffle and reduce stages represent a significant fraction in the overall completion time of *KMeans_16*, this leads to a significant inaccuracy in predicting the job completion time.

The prediction error is caused by the skew in the intermediate data and the unbalanced data distribution to the reduce tasks. For KMeans with 50 reduce tasks, the map stage has a higher execution time because the increased number of groups (centroids) in the clustering algorithm increases the number of comparisons for each record in the map phase and it leads to an increased compute time of the map function. Moreover, the increased number of reduce tasks results in a smaller portion of

DOI: 10.1002/cpe

Table III. Measured and predicted completion times for KMmeans with a different number of reduce tasks (i.e., $K = 16$ and $K = 50$).

	Measured completion time (s)	Predicted completion time (s)
KMeans_16	1910	1275
KMeans_50	3605	3683

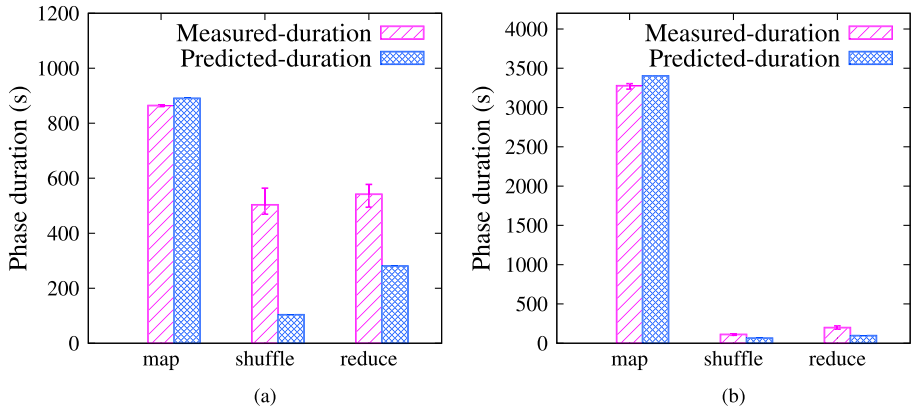

Figure 14. Predicted vs measured stage durations for KMeans application with different number of reduce tasks: (a) $K = 16$ and (b) $K = 50$.

data processed by each reduce task. This significantly decreases the durations of shuffle and reduce stages and masks a possible impact of a data skew in these stages on the overall completion time. Therefore, the prediction of execution time for *KMmeans_50* is more accurate.

It is an interesting future direction to add some data sampling (or augment the job profiling with a special technique) for estimating the data skew in the original (intermediate) data and evaluating the data skew impact on the overall job completion time.

Another interesting direction for the future work is designing an analysis framework to automate the parameter setting required for a good coverage benchmarking set. The quality of the derived platform model critically depends on the generated training dataset, that is, the platform profile. In Section 6, we showed how the shuffle phase model depends on the Hadoop cluster configuration parameters and how these configuration settings impact the benchmark parameters. Automating the derivation of these parameters is a challenging problem that needs to be solved.

A related research thread is to perform a sensitivity study of the model accuracy with respect to the benchmarking set size. Intuitively, using a larger set of benchmarks generated by diverse parameter values should improve the model quality. An interesting question is what is the minimal benchmarking set to use for achieving a predefined model quality.

8. RELATED WORK

In the past few years, performance modeling, workload management, and job scheduling in MapReduce environments have received much attention.

Different approaches [5–9] were offered for predicting the performance of MapReduce applications.

Starfish [5] applies dynamic Java instrumentation to collect a run-time monitoring information about job execution at a fine granularity and by extracting a diverse variety of metrics. Such a detailed job profiling enables the authors to predict job execution under different Hadoop configuration parameters, automatically derive an optimized cluster configuration, and solve cluster sizing

DOI: 10.1002/cpe

problem [6]. However, collecting a large set of metrics comes at a cost, and to avoid significant overhead, profiling should be applied to a small fraction of tasks. Another main challenge outlined by the authors is a design of an efficient searching strategy through the high-dimensional space of parameter values. Our phase profiling approach is inspired by Starfish [5]. We build a lightweight profiling tool that only collects selected phase durations, and therefore, it can profile each task at a minimal cost. Moreover, we apply counter-based profiling in a small test cluster to avoid impacting the production jobs.

Tian and Chen [7] propose an approach to predict MapReduce program performance from a set of test runs on small input datasets and small number of nodes. By executing 25–60 diverse test runs, the authors create a training set for building a regression-based model of a given application. The derived model is able to predict the application performance on a larger input and a different size Hadoop cluster. It is an interesting approach, but it requires executing 25–60 test runs per each application for deriving the corresponding model. Our approach offers a single platform model that can be used across multiple applications.

ARIA [8] proposes a framework that automatically extracts compact job profiles from the past application run(s). These job profiles form the basis of a MapReduce analytic performance model that computes the lower and upper bounds on the job completion time. ARIA provides a fast and efficient capacity planning model for a MapReduce job with timing requirements. The later work [9] enhances and extends this approach by running the application on the smaller data sample and deriving the scaling factors for these execution times to predict the job completion time when the original MapReduce application is applied for processing a larger dataset. By contrast, our approach derives the performance model for Hadoop's generic execution phases and reuses this model for different applications.

Polo *et al*. [18] introduce an online job completion time estimator that can be used for adjusting the resource allocations of different jobs. However, their estimator tracks the progress of the map stage alone and has no information or control over the reduce stage.

In [19], the authors design a model based on closed queuing networks for predicting the execution time of the map phase of a MapReduce job. The proposed model captures contention at compute nodes and parallelism gains due to the increased number of slots available to map tasks.

Castiglione *et al*. [20] pursue an interesting attempt to apply the mean field analysis as a modeling approach for performance evaluation of big data architectures.

In [21], the authors try to adopt the online scheduling strategy from real-time system like earliest deadline first that assigns higher priority to a job with a tighter (earlier) deadline. However, it does not provide any guarantees for achieving the job performance goals: The scheduler assigns all available resources to the job with the highest priority (i.e., the earliest deadline) and then kills the job if its deadline cannot be satisfied.

Morton *et al*. [22] propose *ParaTimer* for estimating the progress of parallel queries expressed as Pig scripts that can translate into directed acyclic graphs of MapReduce jobs. The authors rely on earlier debug runs of the query for estimating throughput of map and reduce stages on the user input data samples. However, the approach is based on a simplified assumption that map (reduce) tasks of the same job have the same duration and works only with FIFO scheduler.

Ganapathi *et al*. [23] use kernel canonical correlation analysis to predict the performance of MapReduce workloads. However, they concentrate on Hive queries and do not attempt to model the actual execution of the MapReduce job, but discover the feature vectors through statistical correlation.

Besides the papers that concentrate on performance modeling, there are plenty of works that focus on designing Hadoop schedulers which aim to optimize different performance objectives.

With a primary goal of minimizing the completion times of large batch jobs, the simple *FIFO* scheduler (initially used in Hadoop) was quite efficient. As the number of users sharing the same MapReduce cluster increased, a new *capacity* scheduler [24] was introduced to support more efficient and flexible cluster sharing. Capacity scheduler partitions the cluster resources into different resource pools and provides separate job queues and priorities for each pool. However, within the pools, there are no additional capabilities for performance management of the jobs.

DOI: 10.1002/cpe

As a new trend, in current MapReduce deployments, there is an increasing fraction of ad hoc queries that expect to get quick results back. When these queries are submitted along with long production jobs, neither the FIFO or capacity scheduler works well in these situations. This situation has motivated the design of the *Hadoop fair scheduler* (HFS) [25]. It allocates equal shares to each of the users running the MapReduce jobs, and also tries to maximize data locality by delaying the scheduling of the task, if no local data are available. Similar fairness and data locality goals are pursued in *Quincy* scheduler [26] proposed for the Dryad environment [27]. The authors design a novel technique that maps the fair-scheduling problem to the classic problem of min-cost flow in a directed graph to generate a schedule. The edge weights and capacities in the graph encode the job competing demands of data locality and resource allocation fairness. While both HFS and Quincy allow fair sharing of the cluster among multiple users and their applications, these schedulers do not provide any special support for achieving the application performance goals and the service-level objectives.

A step in this direction is proposed in FLEX [28], which extends HFS by proposing a special slot allocation schema to optimize explicitly some given scheduling metric. FLEX relies on the speedup function of the job (for the map and reduce stages) that produces the job execution time as a function of the allocated slots. This function aims to represent the application model, but it is not clear how to derive this function for different applications and for different sizes of input datasets. FLEX does not provide a technique for job profiling and detailed MapReduce performance model, but instead uses a set of simplifying assumptions about the job execution, tasks durations, and job progress over time. The authors do not offer a case study to evaluate the accuracy of the proposed approach and models in achieving the targeted job deadlines.

Another interesting extension of the existing Hadoop FIFO and fair-share schedulers using the dynamic proportional sharing mechanism is proposed by the Dynamic Priority scheduler [29]. It allows users to purchase and bid for capacity (map and reduce slots) dynamically by adjusting their spending over time. The authors envision the scheduler to be used by deadline or cost optimizing agents on users' behalf. While this approach allows dynamically controlled resource allocation, it is driven by economic mechanisms rather than a performance model and/or application profiling for achieving job completion deadlines.

Originally, Hadoop was designed for homogeneous environment. There has been recent interest [30] in heterogeneous MapReduce environments. There is a body of work focusing on performance optimization of MapReduce executions in heterogeneous environments. Zaharia *et al*. [31] focus on eliminating the negative effect of stragglers on job completion time by improving the scheduling strategy with speculative tasks. The Tarazu project [17] provides a communication-aware scheduling of map computation that aims at decreasing the communication overload when faster nodes process map tasks with input data stored on slow nodes. It also proposes a load-balancing approach for reduce computation by assigning different amounts of reduce work according to the node capacity. Xie *et al*. [32] try improving the MapReduce performance through a heterogeneity-aware data placement strategy: Faster nodes store larger amount of input data. In this way, more tasks can be executed by faster nodes without a data transfer for the map execution. Polo *et al*. [33] show that some MapReduce applications can be accelerated by using special hardware. The authors design an adaptive Hadoop scheduler that assigns such jobs to the nodes with corresponding hardware.

Much of the recent work also focuses on anomaly detection, stragglers, and outliers control in MapReduce environments [34–36] as well as on optimization and tuning cluster parameters and testbed configuration [37, 38]. While this work is orthogonal to our research, these results are important for performance modeling in MapReduce environments. Providing a more reliable, well performing, balanced environment enables reproducible results, consistent job executions, and supports more accurate performance modeling and predictions.

The problem of predicting the application performance on a new or different hardware has fascinated researchers and been an open challenge for a long time [39, 40]. In 1995, Larry McVoy and Carl Staelin introduced the *lmbench* [39]—a suite of operating system microbenchmarks that provides an extensible set of portable programs for system profiling and the use in cross-platform comparisons. Each microbenchmark was purposely created to capture some unique performance properties and features that were present in popular and important applications of that time.

DOI: 10.1002/cpe

Although such microbenchmarks can be useful in understanding the end-to-end behavior of a system, the results of these microbenchmarks provide little information to indicate how well a particular application will perform on a particular system.

A different approach is to use a set of specially generated microbenchmarks to characterize the *relative performance* of the processing pipelines of two underlying Hadoop clusters: *old* and *new* ones. Herodotou *et al*. [6] attempt to derive a *relative model* for Hadoop clusters composed of different Amazon EC2 instances. They use the Starfish profiling technique and a small set of six benchmarks to exercise job processing with data compression and combiner turned on and off. The model is generated with the M5 Tree Model approach [41].

The other prior examples of successfully building relative models include a *relative fitness model* for storage devices [42] using classification and regression tree (CART) models models and a *relative model* between the native and virtualized systems [43] based on a linear regression technique. The main challenges outlined in [42, 43] for building the accurate models are the tailored benchmark design and the benchmark coverage. Both of these challenges are nontrivial: If a benchmark collection used for system profiling is not representative or complete to reflect important workload properties, then the created model might be inaccurate. Finding the right approach to resolve these issues is a nontrivial research challenge.

9. CONCLUSION

Hadoop is increasingly being deployed in enterprise private clouds and also offered as a service by public cloud providers (e.g., Amazon's Elastic Map-Reduce). Many companies are embracing Hadoop for advanced data analytics over large datasets that require completion time guarantees.

In this work, we offer a new benchmarking approach for building a MapReduce performance model that can efficiently predict the completion time of a MapReduce application. We use a set of microbenchmarks to profile generic phases of the MapReduce processing pipeline of a given Hadoop cluster. We derive an accurate platform performance model of a given cluster once and then reuse this model for characterizing performance of generic phases of different applications. The introduced MapReduce performance model combines the knowledge of the extracted job profile and the derived platform performance model to predict the program completion time on a new dataset. In the future work, we intend to apply the proposed approach and models for optimizing program performance, for example, by tuning the number of reduce tasks and tailoring the resource allocation to meet the required completion time guarantees.

ACKNOWLEDGEMENTS

This work was completed during Z. Zhang's internship at HP Labs. Prof. B. T. Loo and Z. Zhang are supported in part by NSF grants CNS-1117185 and CNS-0845552.

REFERENCES

1. Zhang Z, Cherkasova L, Loo BT. Benchmarking approach for designing a MapReduce performance model. *Proceedings of the International Conference on Performance Engineering*, Prague, Czech Republic, 2013; 253–258.
2. Das S, Sismanis Y, Beyer K, Gemulla R, Haas P, McPherson J. Ricardo: integrating R and Hadoop. *Proceedings of SIGMOD*, Indianapolis, IN, USA, 2010; 987–998.
3. Chen S. Cheetah: a high performance, custom data warehouse on top of MapReduce. *PVLDB* 2010; **3**(2): 1459–1468.
4. Fusco F, Stoecklin M, Vlachos M. NET-Fli: on-the-fly compression, archiving and indexing of streaming network taffic. *Proceedings of VLDB* 2010; **3**:1382–1393.
5. Herodotou H, Lim H, Luo G, Borisov N, Dong L, Cetin F, Babu S. Starfish: a self-tuning system for big data analytics. *Proceedings of 5th Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, 2011; 261–272.
6. Herodotou H, Dong F, Babu S. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. *Proceedings of ACM Symposium on Cloud Computing*, Cascais, Portugal, 2011; 18:1–18:14.
7. Tian F, Chen K. Towards optimal resource provisioning for running MapReduce programs in public clouds. *Proceedings of IEEE Conference on Cloud Computing (CLOUD 2011)*, Washington DC, USA; 155–162.

DOI: 10.1002/cpe

8. Verma A, Cherkasova L, Campbell RH. ARIA: automatic resource inference and allocation for MapReduce environments. *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC)*, Karlsruhe, Germany, 2011; 235–244.
9. Verma A, Cherkasova L, Campbell RH. Resource provisioning framework for MapReduce jobs with performance goals. *Proceedings of the 12th ACM/IFIP/USENIX Middleware Conference*, Lisbon, Portugal, 2011; 160–179.
10. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
11. White T. *Hadoop: The Definitive Guide*. Yahoo Press.
12. *Kick start Hadoop*. Available from: http://kickstarthadoop.blogspot.com/2011/04/word-count-hadoop-map-reduce-example.html. [Accessed on 13 November 2013].
13. Herodotou H, Babu S. Profiling, what-if analysis, and costbased optimization of mapreduce programs. *Proceedings of the VLDB Endowment* 2011; **4**(11):1111–1122.
14. *BTrace: a dynamic instrumentation tool for Java*. Available from: http://kenai.com/projects/btrace. [Accessed on 13 November 2013].
15. Apache. *Hadoop: TeraGen class*. Available from: http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/examples/terasort/TeraGen.html. [Accessed on 13 November 2013].
16. Holland PW, Welsch RE. Robust regression using iteratively reweighted least-squares. *Communications in Statistics-Theory and Methods* 1977; **6**(9):813–827.
17. Ahmad F, Chakradhar S, Raghunathan A, Vijaykumar TN. Tarazu: optimizing MapReduce on heterogeneous clusters. *Proceedings of ASPLOS*, London, England, UK, 2012; 61–74.
18. Polo J, Carrera D, Becerra Y, Torres J, Ayguadé E, Steinder M, Whalley I. Performance-driven task co-scheduling for mapreduce environments. *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium*, Osaka, Japan, 2010; 373–380.
19. Bardhan S, Menasce D. Queuing network models to predict the completion time of the map phase of MapReduce jobs. *Proceedings of the Computer Measurement Group International Conference*, Las Vegas, NV, USA, 2012.
20. Castiglionea A, Gribaudo M, Iacono M, Palmieri F. Exploiting mean field analysis to model performances of big data architectures. *Future Generation Computer Systems, Elsevier*, 2013.
21. Phan LTX, Zhang Z, Zheng Q, Loo BT, Lee I. An empirical analysis of scheduling techniques for real-time cloud-based data processing. *Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications, (SOCA '11)*, Irvine, CA, USA, 2011; 1–8.
22. Morton K, Balazinska M, Grossman D. ParaTimer: a progress indicator for MapReduce DAGs. In *Proceedings of SIGMOD*. ACM: Indianapolis, IN, USA, 2010; 507–518.
23. Ganapathi A, Chen Y, Fox A, Katz R, Patterson D. Statistics-driven workload modeling for the cloud. *Proceedings of 5th International Workshop on Self Managing Database Systems (SMDB)*, Long Beach, CA, USA, 2010; 87–92.
24. Apache. *Capacity scheduler guide*, 2010. Available from: http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html. [Accessed on 13 November 2013].
25. Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys*. ACM: Paris, France, 2010; 265–278.
26. Isard M, Prabhakaran V, Currey J, Wieder U, Talwar K, Goldberg A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*. ACM: Big Sky, MT, USA, 2009; 261-276.
27. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS OS Review* 2007; **41**(3):59–72.
28. Wolf J, Rajan D, Hildrum K, Khandekar R, Kumar V, Parekh S, Wu K-L, Balmin A. FLEX: a slot allocation scheduling optimizer for mapreduce workloads. *Proceedings of the 11th ACM/IFIP/USENIX Middleware Conference*, Bangalore, India, 2010; 1–20.
29. Sandholm T, Lai K. Dynamic proportional share scheduling in Hadoop. *LNCS: Proceedings of the 15th Workshop on Job Scheduling Strategies for Parallel Processing*, Atlanta, GA, USA, 2010; 110–131.
30. Zaharia M, Konwinski A, Joseph AD, Katz R, Stoica I. Improving mapreduce performance in heterogeneous environments. *Proceedings of OSDI*, San Diego, CA, USA, 2008; 29–42.
31. Zaharia M, Konwinski A, Joseph AD, Katz R, Stoica I. Improving MapReduce performance in heterogeneous environments. *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation, OSDI'08*, 2008.
32. Xie J, Yin S, Ruan X, Ding Z, Tian Y, Majors J, Manzanares A, Qin x. Improving MapReduce performance through data placement in heterogeneous Hadoop clusters. *Proceedings of the IPDPS Workshops: Heterogeneity in Computing*, Atlanta, GA, US, 2010; 1–9.
33. Polo J, Carrera D, Becerra Y, Beltran V, Torres J, Ayguadé E. Performance management of accelerated mapreduce workloads in heterogeneous clusters. *Proceedings of the 41st International Conference on Parallel Processing*, San Diego, CA, USA, 2010; 653–662.
34. Konwinski A, Zaharia M, Katz R, Stoica I. X-tracing Hadoop. *Hadoop Summit*, 2008.
35. Tan J, Pan X, Kavulya S, Marinelli E, Gandhi R, Narasimhan P. Kahuna: problem diagnosis for mapreduce-based cloud computing environments. *12th IEEE/IFIP Noms*, Osaka, Japan, 2010; 112–119.
36. Ananthanarayanan G, Kandula S, Greenberg A, Stoica I, Lu Y, Saha B, Harris E. *Reining in the outliers in map-reduce clusters using Mantri*, 2010.

DOI: 10.1002/cpe

37. Intel. *Optimizing Hadoop* deployments*, 2010. Available from: http://communities.intel.com/docs/DOC-4218. [Accessed on 13 November 2013].
38. Kambatla K, Pathak A, Pucha H. Towards optimizing Hadoop provisioning in the cloud. *Proceedings of the First Workshop on Hot Topics in Cloud Computing*, San Diego, CA, USA, 2009; 118.
39. McVoy L, Staelin C. lmbench: portable tools for performance analysis. *Proceedings of USENIX ATC*, San Diego, CA, 1996; 23–23.
40. Seltzer M, Krinsky D, Smith K, Zhang X. The case for application-specific benchmarking. *Proceedings of Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, 1999; 102–107.
41. Quinlan JR. Learning with continuous classes. *Proceedings of Australian Joint Conference on Artificial Intelligence*, Singapore, 1992; 343–348.
42. Mesnier M, Wachs M, Sambasivan R, Zheng A, Ganger G. Modeling the relative fitness of storage. *Proceedings of ACM SIGMETRICS*, San Diego, CA, USA, 2007; 37–48.
43. Wood T, Cherkasova L, Ozonat K, Shenoy P. Profiling and modeling resource usage of virtualized applications. *Proceedings of ACM/IFIP/USENIX Middleware*, Leuven, Belgium, 2008; 366–387.

DOI: 10.1002/cpe