Benchmarking Approach for Designing a MapReduce Performance Model

Zhuoyao Zhang
University of Pennsylvania
zhuoyao@seas.upenn.edu

Ludmila Cherkasova
Hewlett-Packard Labs
lucy.cherkasova@hp.com

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

ABSTRACT

In MapReduce environments, many of the programs are reused for processing a regularly incoming new data. A typical user question is how to estimate the completion time of these programs as a function of a new dataset and the cluster resources. In this work[1], we offer a novel performance evaluation framework for answering this question. We observe that the execution of each 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* and depend on the amount of data processed by the phase and the performance of underlying Hadoop cluster. First, we design *a set of parameterizable microbenchmarks* to measure 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 offer *a MapReduce performance model* that estimates the program completion time for processing a new dataset. The evaluation study justifies our approach and the proposed framework: we are able to accurately predict performance of the diverse set of twelve MapReduce applications. 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.

Categories and Subject Descriptors: C.4 [Computer System Organization] Performance of Systems, D.2.6.[Software] Programming Environments.

General Terms: Measurement, Performance, Design.

Keywords: MapReduce, benchmarking, job profiling, performance modeling

1. INTRODUCTION

MapReduce and Hadoop represent an economically compelling alternative for efficient large scale data processing and cost-effective analytics over "Big Data" in the enterprise. There is a slew of interesting applications associated with live business intelligence that require completion time guarantees. While there were a few research efforts to design different models and approaches for predicting performance of MapReduce applications [6, 5, 8, 9, 10], 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 [9], and deriving some scaling factors for these execution times when the original MapReduce application is applied for processing a larger dataset [10, 8]. Some other efforts [6, 5, 8] 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.

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. In some sense, it combines the useful rationale of the detailed *phase* profiling method [6] in order to more accurately estimate durations of map and reduce tasks, and then apply fast and efficient analytical models designed in [9]. 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. 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*, i.e., 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 [6, 5, 8], profiling is done for all the phases (including the generic ones) for each application separately. Then these 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, e.g., having only 5 nodes. This test cluster employs a similar hardware and Hadoop configuration as a production cluster, and its advantage is that the benchmarking process does not interfere with the production jobs, and therefore, can be performed much faster while testing a large set of diverse processing patterns. The input parameters of microbenchmarks im-

[1]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.

ICPE'13, April 21-24, 2013, Prague, Czech Republic.

pact the amount of data processed by different phases of map and reduce tasks. We concentrate on profiling of general (non-customized) phases of 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 time 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 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, i.e., the ratio of the map (reduce) output to the map (reduce) input. 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 in a small 5-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 are within 10% of the measured ones.

The rest of this paper is organized as follows. Section 2 provides MapReduce background and presents our phase profiling approach. Section 3 describes microbenchmarks and objectives for their selection. Sections 4 , 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 outlines related work. Section 8 presents conclusion and future work directions.

2. MAPREDUCE PROCESSING PIPELINE

In the MapReduce model [4], 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 reduce function then merges or aggregates all the values associated with the same key. MapReduce jobs are executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*.

The execution of each map (reduce) task is comprised of a specific, well-defined sequence of processing phases (see Fig. 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 are *generic* (i.e., defined by Hadoop code), and mostly depend 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 plan to run a set of microbenchmarks that create different amounts of data for processing per map (reduce) tasks and for processing by their phases. We need to profile a 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.

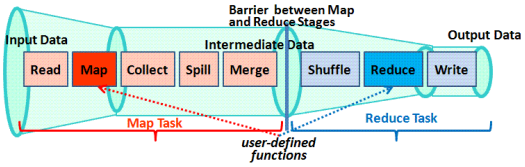


Figure 1: MapReduce Processing Pipeline.

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 (HDFS). However, written data files might be of arbitrary size, e.g., 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. We measure the duration of the read phase as well as 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.

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 (∼*320 MB*) is written to disk. After all the data is shuffled, Hadoop merge-sorts first 10 spilled files and writes them in the new sorted file. Then it merge-sorts 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 *3.2 GB* 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 HDFS.

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:

- *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 above, each executed task has a constant overhead for setting and cleaning up. We account for these overheads separately for each task.

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 [6] approach based on *BTrace* – a dynamic instrumentation tool for Java [1]. 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 to 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.

In this work, first, we use profiling to create *a platform performance model*. We execute a set of microbenchmarks (described in the next 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. This profiling is done on a small test cluster (5-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 due to its simplicity and low overhead, and that the modified Hadoop version can be easily deployed in this test environment.

The *MapReduce performance model* needs additional measurements that characterize the execution of user-defined map and reduce functions of a given job. 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 *BTrace* approach. Remember, this approach does not require Hadoop or application changes, and can be switched on for profiling a targeted MapReduce job of interest. Since we only profile map and reduce phase executions the extra overhead is relatively small.

3. MICROBENCHMARKS AND PLATFORM PROFILE

We generate and perform 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* (M^{inp}): 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* (M^{sel}): 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 processeded 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 control the number of reduce tasks to expedite the training set generation with the large amount of intermediate data per reduce task.

Thus, each microbenchmark MB_i is parameterized as

$$MB_i = (M_i^{inp}, M_i^{sel}, N_i^{map}, N_i^{red}).$$

Each created benchmark uses input data consisting of 100 byte key/value pairs generated with *TeraGen* [3], 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 5 worker nodes (see setup details in Section 6). We run benchmarks with the following parameters: M^{inp}={2MB, 4MB, 8MB, 16MB, 32MB, 64MB}; 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}; $M^{inp} = 64MB$, $M^{sel} = 5.0$ and $N^{red} = 5$ which result in different intermediate data size per reduce tasks ranging from 1 GB to 12 GB.

We generate the *platform profile* by running a set of our microbenchmarks on the small 5-node test cluster that is similar to a given production Hadoop cluster. 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*:

- *Map task processing:* in the collected platform profiles, we denote the measurements for phase durations and the amount of processed data for *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 2 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 2 shows fragments for *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.

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 2: A fragment of a platform profile for *read* and *collect* phases.

4. PLATFORM PERFORMANCE MODEL

Now, we describe how to create a *platform performance model* M_{Phases} which 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 $M_1, M_2, ..., M_5$, 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 (see Figure 2).

Below, we explain how to build a submodel M_i, where $1 \leq i \leq 6$. By using measurements from the collected platform profiles, we form a set of equations which express a phase duration as a linear function of processed data. 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 $i = 1, 2, \cdots, 6$):

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

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 non-negative Least Squares Regression). To decrease the impact of occasional bad measurements and to improve the overall model accuracy, we employ robust linear regression [7].

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

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 [9]. 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 lower and upper bounds serves as a good prediction: it is within 10% of the measured completion time as shown in [9].

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. 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;
- 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 $M_{Phases} = (M_1, M_2, ..., M_5, M_6)$ by applying a correspoding function to the data amount processed by the phase. Note, that the data size for *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 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 *shuffle, reduce*, and *write* phases. Similarly to the map task computation described above, 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:

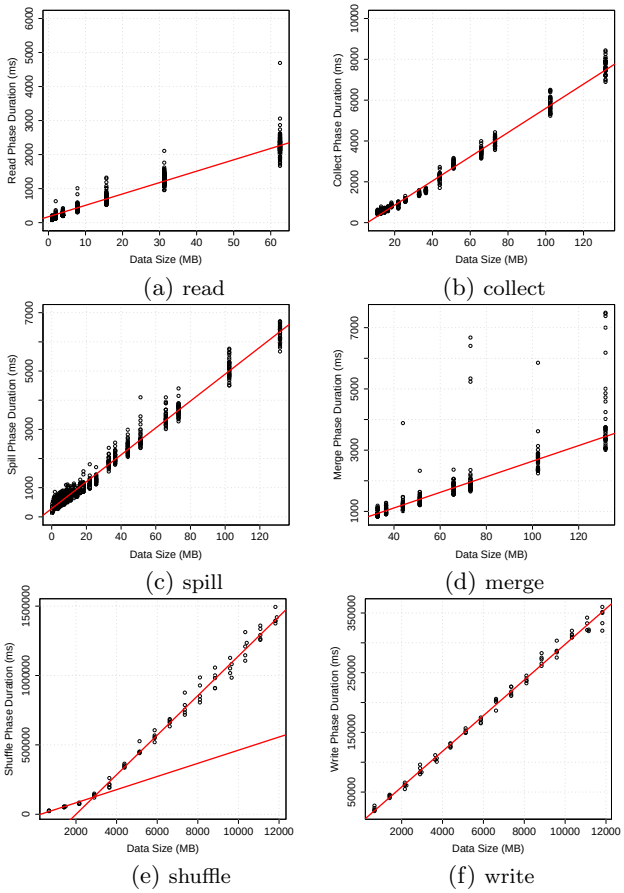
$$Data_{shuffle} = (M^{inp} \times M^{sel} \times N^{map})/N^{red}, \qquad (2)$$

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

6. EVALUATION

All experiments are performed on 66 HP DL145 G3 machines. Each machine has four AMD 2.39GHz 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 2 map and 2 reduce slots. The file system blocksize is set to 64MB. The replication level is set to 3. We disabled speculative execution since it did not lead to significant improvements in our experiments.

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 5-node test cluster that uses the same hardware and configuration as the large 66-node production cluster. Figure 3 shows the relationships between the amount of processed data and the execution durations of different phases for a given Hadoop cluster. Figures 3 (a)-(f) reflect 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 red line



(a) read (b) collect (c) spill (d) merge (e) shuffle (f) write

Figure 3: Benchmark results.

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 piece-wise function comprised 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.

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 d_i^{measrd}. The relative error is defined as follows:

$$error_i = \frac{|d_i^{measrd} - d_i^{pred}|}{d_i^{measrd}}$$

We calculate the relative error for all the data points in the platform profile. Table 1 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 that 20%. For the *shuffle* phase, 76% of reduce tasks have the relative error less that 10%, and 96% of reduce tasks have the relative error less that 20%.

In summary, almost 80% of all the predicted values are within 15% of the corresponding measurements. Thus the derived platform performance model fits well the collected experimental data.

Table 1: Relative error distribution

phase	error $\leq$ 10%	error $\leq$ 15%	error $\leq$ 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%

Next, we validate the accuracy of the proposed MapReduce performance model by predicting completion times of 12 applications made available by the Tarazu project [2].

Table 2 gives a high-level description of these 12 applications with the job settings (e.g, 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 will present 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.

Table 2: Application characteristics.

Application	*Input data (type)*	*Input (GB) small*	*#Map, Reduce tasks*	*Input (GB) large*	*#Map, Reduce tasks*
TeraSort	Synthetic	2.8	44, 20	31	495, 240
WordCount	Wikipedia	2.8	44, 20	50	788, 240
Grep	Wikipedia	2.8	44, 1	50	788, 1
InvIndex	Wikipedia	2.8	44, 20	50	788, 240
RankInvIndex	Wikipedia	2.5	40, 20	46	745, 240
TermVector	Wikipedia	2.8	44, 20	50	788, 240
SeqCount	Wikipedia	2.8	44, 20	50	788, 240
SelfJoin	Synthetic	2.1	32, 20	28	448, 240
AdjList	Synthetic	2.4	44, 20	28	508, 240
HistMovies	Netflix	3.5	56, 1	27	428, 1
HistRatings	Netflix	3.5	56, 1	27	428, 1
KMeans	Netflix	3.5	56, 16	27	428, 50

Figure 4 shows the comparison of the measured and predicted job completion times [2] for 12 applications (with a small input dataset) executed using 5-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.

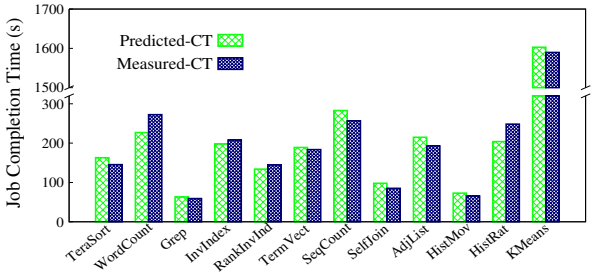


Figure 4: Predicted vs. measured completion times of 12 applications on the *small 5-node test cluster*.

The next question to answer is whether the *platform performance model* constructed using a small 5-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 input dataset) on the 66-node production cluster. Figure 5 shows the comparison of the measured and predicted job completion

[2] All the experiments are performed five times, and the measurement results are averaged. This comment applies to the results in Figure 4, 5.

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 (a worst case is *WordCount* that exhibits 17% of error). Note the split at Y-axes for accomodating the *KMeans* completion time in the same figure.

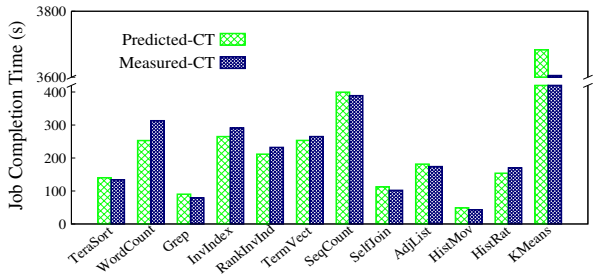


Figure 5: Predicted vs. measured completion times of 12 applications (with a large input dataset) on the *large 66-node production cluster*.

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, since these measurements do not interfere with production workloads while 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.

7. RELATED WORK

In past few years, performance modeling and workload management in MapReduce environments have received much attention, and different approaches [6, 5, 8, 9, 10] were offered for predicting performance of MapReduce applications.

Starfish [6] 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 problem [5]. 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* [6]. We build a light-weight 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 [8] 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 cannot be directly applied for job performance optimization and parameter tuning problems.

ARIA [9] 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 [10] 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.

In our current work, we consider a more detailed profiling of MapReduce jobs via six generic and two customized execution phases. The proposed models aim to estimate durations of these phases and accordingly the durations of map and reduce tasks when the application is executed on the new dataset. Once, we obtain predicted map and reduce task durations (by applying the proposed platform performance model and using the extracted job profile), we can utilize performance models designed in [9] for predicting the job completion time as a function of allocated resources.

8. 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, e.g., by tuning the number of reduce tasks and tailoring the resource allocation to meet the required completion time guarantees.

9. REFERENCES

[1] BTrace: A Dynamic Instrumentation Tool for Java. `http://kenai.com/projects/btrace`.

[2] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar. Tarazu: Optimizing MapReduce on heterogeneous clusters. In *Proc. of ASPLOS*, 2012.

[3] Apache. Hadoop: TeraGen Class. `http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/examples/terasort/TeraGen.html`.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[5] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proc. of ACM Symposium on Cloud Computing*, 2011.

[6] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of 5th Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[7] P. Holland and R. Welsch. Robust regression using iteratively reweighted least-squares. *Communications in Statistics-Theory and Methods*, 6(9):813–827, 1977.

[8] F. Tian and K. Chen. Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds. In *Proc. of IEEE Conference on Cloud Computing (CLOUD 2011)*.

[9] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. *Proc. of the 8th ACM International Conference on Autonomic Computing (ICAC)*, 2011.

[10] A. Verma, L. Cherkasova, and R. H. Campbell. Resource Provisioning Framework for MapReduce Jobs with Performance Goals. *Proc. of the 12th ACM/IFIP/USENIX Middleware Conference*, 2011.