MC^2: Map Concurrency Characterization for MapReduce on the Cloud

Mohammad Hammoud and Majd F. Sakr
Carnegie Mellon University in Qatar
Education City, Doha, State of Qatar
Emails: {mhhammou, msakr}@qatar.cmu.edu

***Abstract*—MapReduce is now a pervasive analytics engine on the cloud. Hadoop is an open source implementation of MapReduce and is currently enjoying wide popularity. Hadoop offers a high-dimensional space of configuration parameters, which makes it difficult for practitioners to set for efficient and cost-effective execution. In this work we observe that MapReduce application performance is highly influenced by map concurrency. Map concurrency is defined in terms of two configurable parameters, the number of available map slots and the number of map tasks running over the slots. We show that some inherent MapReduce characteristics enable well-informed prediction of map concurrency. We propose Map Concurrency Characterization (MC^2), a standalone utility program that can predict the best map concurrency for any given MapReduce application. By leveraging the generated predicted information, MC^2 can judiciously guide Map phase configuration and, consequently, improve Hadoop performance. Unlike many of relevant schemes, MC^2 does not employ simulation, dynamic instrumentation, and/or static analysis of unmodified job code to predict map concurrency. In contrast, MC^2 utilizes a simple, yet effective mathematical model, which exploits the MapReduce characteristics that impact map concurrency. We implemented MC^2 and conducted comprehensive experiments on a private cloud and on Amazon EC2 using Hadoop 0.20.2. Our results show that MC^2 can correctly predict the best map concurrencies for the tested benchmarks and provide up to 2.2X speedup in runtime.**

I. Introduction

MapReduce [8] is now a popular choice for big data processing and is highly recognized for its elasticity, scalability and fault-tolerance. For instance, Google utilizes MapReduce to process 20PB of data per day [8]. Amazon added a new service, called Amazon Elastic MapReduce to enable businesses, researchers, data analysts, and developers to easily process vast amounts of data [2]. In essence, Amazon Elastic MapReduce has created a market for pay-as-you-go analytics on the cloud [19].

MapReduce provides minimal abstractions, hides architectural details, and automatically parallelizes computation by running multiple map and/or reduce tasks over distributed data across multiple machines. MapReduce incorporates two phases, Map and Reduce phases, and allows programmers to write sequential map and reduce functions that are transformed by the framework into concurrent map and reduce tasks. Hadoop [13] is an open source implementation of MapReduce. Hadoop's adoption by academic, governmental, and industrial organizations is growing at a fast pace [19]. For example, industry's premier web vendors such as Facebook, Yahoo! and Microsoft have already advocated Hadoop [22]. Academia is currently using Hadoop for seismic simulation, natural language processing, and web data mining, among others [15], [38].

Nonetheless, Hadoop users are faced with a main challenge on the cloud. In particular, they lack the ability to run MapReduce applications in the most economical way, while still achieving good performance. The approach of renting more nodes so as to enhance performance is not cost-effective in the cloud's pay-as-you-go environment [26]. As such, in addition to elasticity, scalability and fault-tolerance, an ideal analytics engine should provide high-performing and cost-effective execution framework for big data applications on the cloud.

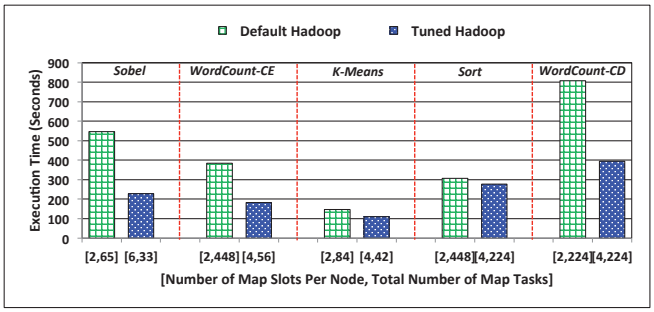

Figure 1. The execution times of various benchmarks under default Hadoop and Hadoop with two tuned parameters, # of map tasks and # of map slots (the two parameters that define map concurrency).

Hadoop has more than 190 configuration parameters out of which 10-20 parameters can have significant impact on job performance [17]. Today, the burden falls on Hadoop users to specify effective settings for all these parameters. Hadoop's default configuration settings do not necessarily provide the best performance. Thus, they might lead to some inefficiency when Hadoop is deployed on the cloud. Fig. 1 depicts the execution times of various MapReduce applications run on a private cloud[1] under two Hadoop configurations, the default one and a one with two tuned parameters, the number of map tasks and the number of map slots. As shown, the tuned configuration provides Hadoop with speedups of 2.3X, 2.1X, 1.3X, 1.1X and 2X for Sobel, WordCount-CE, K-Means, Sort, and WordCount-CD, respectively. Clearly, this demonstrates that: (1) Hadoop's default configuration is not optimal, (2) the numbers of map tasks and slots (or what we refer to as *map concurrency*) have a strong impact on Hadoop performance, and (3) for effective execution, Hadoop might require different configuration settings for different applications.

Selecting an appropriate Hadoop configuration is not a trivial task. Exhausting all possible options for a single parameter, let alone all parameters, is a complex, time-consuming, and quite expensive process. Furthermore, even if an optimal configuration is located for a specific application,

[1]The experimentation environment and all our benchmarks are described in Section VI.

it might not be applicable to other applications (see Fig. 1). Therefore, pursuing a brute-force scan over every parameter per every application is clearly an inefficient approach. Indeed, setting Hadoop parameters for efficient execution is a form of art, which typically requires extensive knowledge of Hadoop internals [24]. Most practitioners of big data analytics (e.g., computation scientists, systems researchers, and business analysts) lack the expertise to tune Hadoop and improve performance [19]. Consequently, they tend to either run Hadoop using the default configuration, thus potentially missing a promising optimization opportunity on the cloud, or learn the internal intricacies of MapReduce to select satisfactory Hadoop configuration settings, or hire expertise to accomplish the mission. We argue that practitioners need not do all that. Specifically, we suggest that a simple, accurate and fast scheme can be devised to effectively guide Hadoop configuration on the cloud.

As map concurrency greatly influences MapReduce performance, in this work we focus on optimizing the Map phase in MapReduce. Optimizing the Reduce phase is also crucial and has been left for future exploration. We propose Map Concurrency Characterization (MC^2), a highly accurate predictor that predicts the best map concurrencies for MapReduce applications. MC^2 is based on a simple mathematical model that leverages two main MapReduce characteristics: (1) data shuffling (i.e., moving Map phase output to Reduce phase) and (2) total overhead for setting up all map tasks in a job. This is contrary to many current related schemes that incorporate simulation, dynamic instrumentation and/or static analysis of unmodified MapReduce application code to accomplish a similar objective. MC^2 is a standalone utility program that only requires some information about the given application and speedily enough (in microseconds) can predict the best map concurrency for the application without involving Hadoop.

In this paper we make the following contributions:

- We show a strong dependency between the execution times of MapReduce applications and map concurrency.
- We characterize MapReduce to figure out the core characteristics that impact map concurrency.
- We develop a general mathematical model that leverages the discovered concurrency characteristics and allows estimating runtimes of MapReduce applications.
- We propose MC^2, a novel predictor that effectively utilizes the suggested mathematical model to predict the best map concurrency for any given MapReduce application.
- We present a strategy that can serve in reducing cost and improving performance in a cloud setting.
- We offer a timely contribution to data analytics on the cloud, especially as Hadoop usage continues to grow beyond companies like Google, Microsoft, Facebook and Yahoo!.

The rest of the paper is organized as follows. An overview of Hadoop is presented in Section II. We characterize MapReduce for map concurrency in Section III. Section IV presents our suggested mathematical model. We describe MC^2 in Section V. Section VI discusses our evaluation methodology and results. Finally, we provide a summary of prior work in Section VII and conclude in Section VIII.

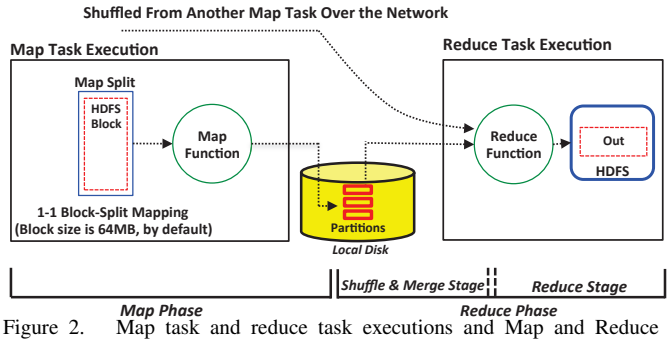

Figure 2. Map task and reduce task executions and Map and Reduce phases. Reduce phase includes two stages, Shuffle and Merge stage and Reduce stage.

II. Hadoop Overview

A. Hadoop Architecture and MapReduce Phases

Hadoop is an open source implementation of MapReduce. Hadoop presents MapReduce as an analytics engine and under the hood uses a distributed storage layer referred to as Hadoop Distributed File System (HDFS). HDFS mimics Google File System (GFS) [21]. MapReduce adopts a tree-style, master-salve architecture. The master is denoted as JobTracker and each slave node is called a TaskTracker. The JobTracker is responsible for scheduling map and reduce tasks at specific TaskTrackers in a Hadoop cluster, monitoring them and re-executing failed ones.

A MapReduce job typically includes two phases, a Map phase and a Reduce phase. Nonetheless, a job can still have only a Map phase and will, consequently, be referred to as a *Reduce-Less job* [7]. In the presence of a Reduce phase, map tasks in the Map phase produce and store intermediate outputs on local disks (means not on HDFS) and partition them to designated reduce tasks. Each reduce task pulls its corresponding partitions in a process known as *shuffling*, merges them, applies on the merged outcome a user-defined reduce function, and stores final results in HDFS (see Fig. 2). Thus, the Reduce phase is usually broken up into a Shuffle and Merge stage and a Reduce stage as shown in Fig. 2. In the absence of a Reduce phase, map tasks write their outputs directly to HDFS.

B. HDFS Blocks and Map Splits

The input data to a MapReduce job is divided by HDFS into fixed-size pieces denoted as chunks or blocks. The user-defined function in a map task operates on one or many HDFS blocks encapsulated in what is termed as a *split* (see Fig. 2). For data locality reasons, a common practice in Hadoop is to have each split encompassing only one block. Specifically, when a split includes more than one block, the probability of these blocks to exist on the same node- where a map task will run- becomes low, leading thereby to a network transfer of at least one block per every map task. In contrary, with a *one-to-one* mapping between splits and blocks, a map task can be scheduled at a node where a block exists and, consequently, results in an improved data locality and a reduced network traffic.

The number of HDFS blocks in a MapReduce job can be computed by dividing the input dataset size by the specified HDFS block size. The HDFS block size is a parameter that can be statically set by a user before running a job (by default, the HDFS block size is 64MB). As each split usually

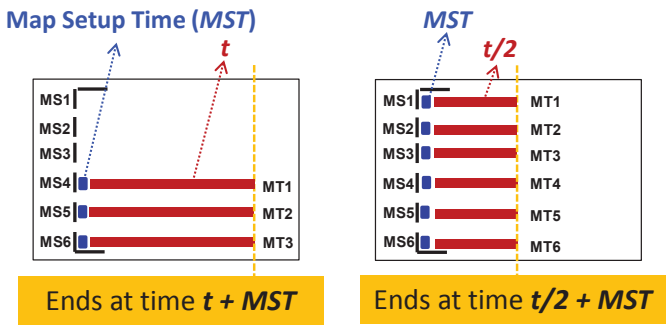

Figure 3. The execution of a MapReduce job with map wave numbers less than 1. MS_i and MT_i stand for Map Slot i and Map Task i, respectively. The small square before each task denotes the overhead required to setup the task. Before the number of map waves is doubled in (b), we assume that each map task takes time, $\boldsymbol{t}$, in (a).

contains a single block, the number of splits in a job becomes equal to the number of blocks. Moreover, as each map task is responsible for only one split, the number of splits becomes equal to the number of map tasks. This makes the number of map tasks equals to the number of HDFS blocks. Therefore, the number of map tasks can be calculated by dividing the input dataset size by the specified HDFS block size.

III. Map Concurrency in MapReduce

Each map task in a MapReduce job is scheduled by the JobTracker (JT) at a TaskTracker (TT) on what is denoted as a *map slot*. A TT in a Hadoop cluster is configured with a set of map slots to indicate the maximum number of map tasks that can run at a time on TT. The number of map slots per a TT can be statically set by a user before running a job. Because each map task runs as a separate process on a TT, a higher number of map slots translates into a higher concurrency, provided that enough map tasks are available to occupy (i.e., execute over) the slots. A higher concurrency can result in an improved performance. A caveat, however, is that a large number of occupied map slots can potentially result in resource contention and, accordingly, degrade overall performance. On the other hand, a very small number of occupied map slots, whereby resource contention is totally avoided, can result in an underutilized system and a degraded performance. Thus, to obtain optimum performance, the numbers of map tasks and slots per a TT must be judiciously selected so as concurrency is highly exploited and resources are maximally utilized, but not contended [27]. In this work, we avoid contending TTs and focus on inferring the best achievable concurrency in terms of the number of map tasks for a *given* number of map slots. Predicting the optimal number of map slots *together* with the number of map tasks is beyond the scope of this paper and is left for future exploration.

We refer to the maximum number of map tasks that can run concurrently at a given time within a Hadoop cluster as *map wave*. Clearly, the number of map waves in a MapReduce job can be computed by dividing the number of map tasks by the aggregate number of map slots in a Hadoop cluster. The process of selecting a number of map waves for a MapReduce job would dictate its achievable map concurrency. As a first step towards inferring the best achievable map concurrencies for MapReduce applications, we suggest *characterizing* map concurrency in MapReduce. Specifically, within the confines of MapReduce, we define map concurrency characterization as the process of observing, identifying and explaining various MapReduce runtime responses to different values of map wave numbers. We distinguish between two main cases: (1) when the number of map tasks is less than or equal to the total number of map slots (i.e., the number of map waves is less than or equal to 1), and (2) when the number of map tasks is greater than the total number of map slots (i.e., the number of map waves is greater than 1). We refer to the former case as ***CASE-I*** and to the latter case as ***CASE-II***. We next characterize map concurrency in MapReduce under *CASE-I* and *CASE-II*.

A. Characterizing Map Concurrency: CASE-I

Let us first consider *CASE-I*. Fig. 3 demonstrates the execution of a MapReduce job with various map task numbers. MS_i and MT_i stand for Map Slot i and Map Task i, respectively. *CASE-I* characterization is simple and does not impact the Reduce phase; hence, we only show the Map phase in the figure. The small square before each map task indicates the setup overhead required for initializing the task and launching a host Java Virtual Machine (JVM)[2]. We refer to the required time for setting up a map task as *Map Setup Time* (*MST*). We further assume that map tasks start and finish at nearly close times, with each taking initially time, $\boldsymbol{t}$, to commit. As long as *CASE-I* holds and we double the number of map tasks, *MST* remains constant while map time $\boldsymbol{t}$ is assumingly cut by half. Map time $\boldsymbol{t}$ is supposed to be cut by half because upon doubling the number of map tasks, the input HDFS block of each task is also cut by half. Therefore, as we maintain *CASE-I* and scale up the number of map tasks, we expect the Map phase to finish earlier. As shown in Fig. 3, if the Map phase in Fig. 3 (a) ends at ($\boldsymbol{t}$ + *MST*), after doubling the number of map tasks, we expect the Map phase in Fig. 3 (b) to end at time ($\boldsymbol{t}$/2 + *MST*). An earlier commit of the Map phase translates to an overall improvement in the job execution time.

In summary, a main observation pertaining to *CASE-I* is that, as map slots are occupied with map tasks, the Map phase runtime is expected to decrease. This is mainly due to: (1) exploiting more map concurrency, and (2) attaining better system utilization. Once the number of map tasks becomes greater than the number of map slots, we hit *CASE-II*, which we next characterize.

B. Characterizing Map Concurrency: CASE-II

In *CASE-II*, the number of map waves in a MapReduce job does not only have an impact on the overall job initialization cost but further on what is known in Hadoop MapReduce as *early shuffle* [15], [16]. As a mechanism to improve performance, Hadoop applies early shuffle by scheduling reduce tasks before *every* corresponding partition is available so as to overlap the Map and Reduce phases, and consequently, decrease the turnaround times of jobs. More precisely, Hadoop activates the Shuffle and Merge stage in the Reduce phase after only 5% of map tasks are done, thus allowing the interleave between the Map phase and the

[2]Hadoop runs each task in its own JVM to isolate it from the rest of running tasks. Hadoop allows *task JVM reuse* in which more than one task can use the same JVM; yet sequentially (i.e., tasks do not run concurrently in a single JVM). Clearly, this can be useful for tasks that share state. Enabling *task JVM reuse* will reduce the JVM setup overhead for the tasks that reuse JVMs, but not the initialization overhead.

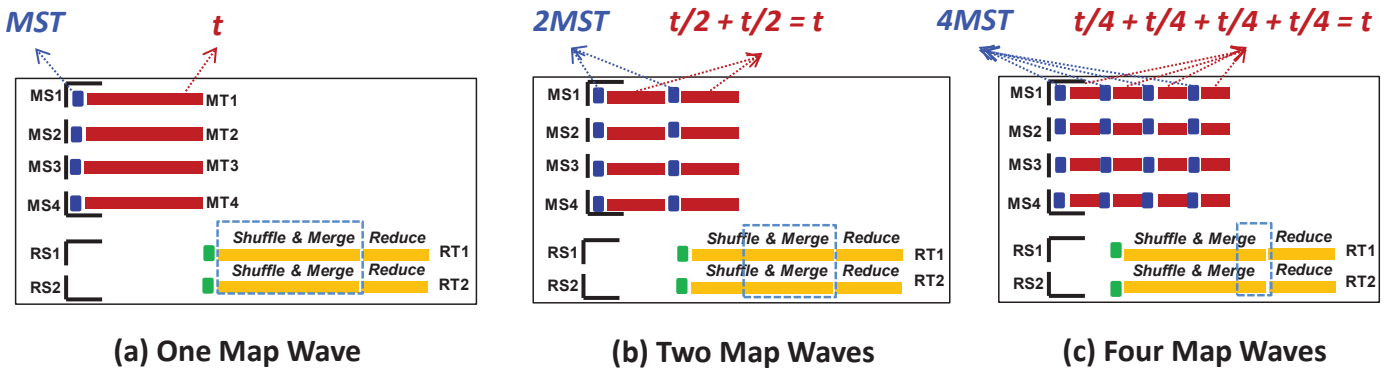

Figure 4. The execution of a MapReduce job with various map wave numbers. MS_i, MT_i, RS_i, and RT_i stand for Map Slot i, Map Task i, Reduce Slot i, and Reduce Task i, respectively. The small square before each task denotes the overhead required to setup the task.

Shuffle and Merge stage. The Reduce stage in the Reduce phase is not allowed to overlap with the Map phase. This is because the user-defined reduce function should be typically applied to the whole/merged reduce input partition, and not fragments of it, so as application correctness is maintained.

Fig. 4 demonstrates the execution of a MapReduce job with various map wave numbers. MS_i, MT_i, RS_i, and RT_i stand for Map Slot i, Map Task i, Reduce Slot i, and Reduce Task i, respectively. We start with one map wave (i.e., *CASE-I*. See Fig. 4 (a)) and characterize map concurrency as we scale up the number of map waves, thus shifting directly to *CASE-II* (see Figures 4 (a) and 4 (b)))[3]. As in *CASE-I*, the small square before each map/reduce task indicates the setup overhead required for initializing the task and launching a host Java Virtual Machine (JVM). Again, we refer to the time needed for setting up a map task as *Map Setup Time* (*MST*) and to the initial time taken by each map task as $\boldsymbol{t}$.

With only one map wave and the assumption that all map tasks start and finish at comparable times, early shuffle cannot be triggered before the entire Map phase is committed (see Fig. 4 (a)). As a result, all data will be shuffled after the Map phase is done. This will increase the time the Reduce stage has to wait before it can execute the user-defined reduce function. In contrary, with two map waves, data shuffling can be ensued while the Map phase is still running (see Fig. 4 (b)). This will decrease the duration the Reduce stage has to wait before it can proceed. In principle, the more the number of map waves is, the earlier the early shuffle process can be activated, and the more overlap between the Map and the Reduce phases can be leveraged (see Fig. 4 (c)). We note that what gets improved in the Reduce phase as early shuffle is activated earlier is, in fact, its response time and not its execution time. This is mainly because: (1) the amount of data to shuffle and merge remains the same, and (2) the actual required time for the Reduce stage is not affected. Nonetheless, the improvement in Reduce response time translates to an overall improvement in job performance.

A critical point to notice is that the gain from the overlap between the Map and the Reduce phases diminishes geometrically as the number of map waves is monotonically scaled up (see the blue rectangles around the Shuffle and Merge stages across Figures 4 (a), 4 (b) and 4 (c)). Specifically, with only 1 map wave, there will be no opportunity to hide shuffle latency since 100% of map tasks will be done before data can be shuffled. With two map waves, however, there will be an opportunity to hide shuffle latency under 50% (i.e., 0% + (100% ÷ 2)) of map tasks. With three map waves, shuffle latency can be hidden under 75% (i.e., 50% + (50% ÷ 2)) of map tasks. Furthermore, with four map waves, shuffle latency can be hidden under 87% (i.e., 50% + 25% + (25% ÷ 2)) of map tasks. In general, upon every single scale-up in the number of map waves, shuffle latency can be hidden under the previous number of map tasks plus half of it. Clearly, this is a geometric sequence with a common ratio of 1/2.

Alongside, as we scale up the number of map waves, the *gain* from early shuffle will be offset by a *loss* from growing *MST*s. In particular, as the number of map waves is doubled, *MST* is also doubled. This shall add to the runtimes of jobs, especially when map tasks have lengthy initializations and jobs have large numbers of short-lived tasks. Conversely, the map time, $\boldsymbol{t}$, will remain constant as the number of map waves is increased (see Fig. 4). In conclusion, the number of map waves for an application must be carefully chosen so as a decent gain from early shuffle is obtained and a minimal loss from an increased total *MST* is avoided. We refer to the number of map waves that achieve such a goal as *the best number of map waves* for the application.

Finally, we note that the preference of *when* exactly the early shuffle process must be activated varies across applications. Specifically, applications induce different amounts of shuffle data. Hence, the more the amount of data an application shuffles, the earlier the early shuffle process must be triggered. This allows hiding more shuffle latency under the Map phase and diminishing further the Reduce response time. Since the shuffle process can be activated earlier by increasing the number of map waves, it can be argued that with a larger amount of shuffle data, a larger number of map waves will be favored. We next suggest a mathematical model that accounts for shuffle data and facilitates locating the best number of map waves for any MapReduce application.

IV. A Mathematical Model

We first present a general mathematical model that can predict the runtimes of MapReduce jobs. In the next section we utilize this model to estimate the best number of map waves for any MapReduce application. In developing our mathematical model, we assume that: (1) map tasks start and finish at nearly close times, and (2) map time is longer

[3]In CASE-II, the number of map waves could be decimal (e.g., 1.5 or 3.2 or 2.7- just as examples). Our presented characterization is general and applies to any number of map waves that exceeds 1.

than map setup time (which is typical for MapReduce applications). Fig. 6 demonstrates our approach in modeling map concurrency in MapReduce. First, as previously, we define *Map Setup Time* (*MST*) as the time required for initializing a map task. We measure the incurred *MST* cost as the number of map waves is increased. Specifically, with a single map wave, *MST* will be counted only once since all *MST*s of all map tasks in a wave will be performed in parallel. On the other hand, with two map waves, *MST* will be counted twice. In general, as we increase the number of map waves, *MST* will scale linearly. We state the total *MST* with any number of map waves in equation (1). The *Number of Map Waves* factor in equation (1) should be always an integer because irrespective of how many map tasks a wave includes, only a single *MST* will be counted for that wave[4]. Hence, we use the ceiling function to transform the real number of map waves to the next smallest integer.

$$\textbf{Total MST} = \lceil \textbf{Number of Map Waves} \rceil \times \textbf{MST} \quad \textbf{(1)}$$

Second, we define *Hidden Shuffle Time* (*HST*) as the shuffle time that is hidden under the Map phase due to utilizing early shuffle. As implied by *CASE-I* and *CASE-II*, with a number of map waves that is less than 1, early shuffle cannot be exploited (see Section III). With two map waves, however, early shuffle can be exploited, yet under only one map wave, or 50% of map tasks. More precisely, with two map waves, the data shuffling that is carried under the second map wave is for the first map wave, which is already done (see Fig. 4 (b)). Therefore, with two map waves, shuffle latency is hidden for the first map wave, and not the second. Likewise, with three map waves, shuffle latency is hidden for the first and second map waves, and not the third. In general, with ***N*** map waves, shuffle latency is hidden for the first ***N - 1*** map waves, and not the last. As an outcome, we define *HST* as follows:

$$\textbf{HST} = \lceil \textbf{Number of Map Waves} - 1 \rceil \times (\textbf{Shuffle Data} \div \textbf{Number of Map Waves}) \times \textbf{Shuffle Rate} \quad \textbf{(2)}$$

As specified in equation (2), *Shuffle Data* is the total intermediate output of an application. We divide *Shuffle Data* by the number of map waves to obtain the amount of shuffle data per a map wave. Furthermore, we apply the ceiling function to (*Number of Map Waves -1*) because latency is always hidden for all the map waves, except the last, which might not be *fully loaded*. In particular, given our assumption that map tasks start and finish at comparable times, all the map waves in a job will be full of map tasks except the last, which might include a number of map tasks smaller than the number of map slots. Thus, we subtract 1 from the *Number of Map Waves* factor to exclude the last wave, and then use the ceiling function in case the remaining number of map waves is decimal (i.e., one wave is not full of map tasks). *Shuffle Rate* is the speed at which data is shuffled over the cluster's network, usually quoted in bits per second.

[4]The number of map waves evaluates to decimal when the remainder of dividing the number of map tasks by the number of map slots is not zero. This means that the last map wave in the respective job will include a number of map tasks that is less than the number of map slots. Regardless of how many map tasks the last wave includes, only 1 MST will be counted for it.

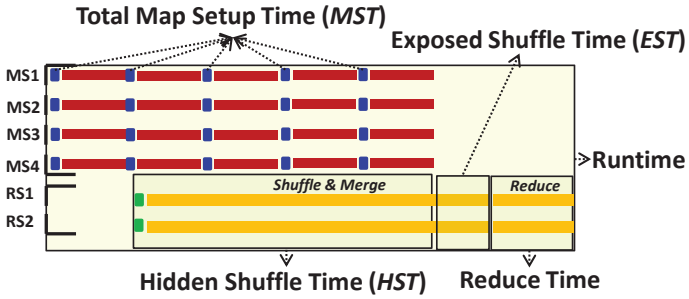

Figure 5. Our approach in modeling the runtime of a job in MapReduce. For each highlighted component we define a time metric.

Third, we define *Exposed Shuffle Time* (*EST*) as the shuffle time that cannot be hidden under the Map phase. Clearly, this is the time needed to shuffle the intermediate data of the last map wave. *EST* is stated in equation (3). To account for the fact that the last map wave might not be fully loaded, we incorporate in equation (3) the factor α. As the amount of data to shuffle per a wave will be less if the wave is not fully loaded, α can determine exactly how loaded is the last map wave. Specifically, if the last map wave is fully loaded, α can be set to 1. Otherwise, α can be set to $(Number\ of\ Map\ Waves - \lfloor Number\ of\ Map\ Waves \rfloor)$, which captures the load at the last map wave. Lastly, the *Shuffle Rate* in equation (3) is defined as in equation (2).

$$\textbf{EST} = (\textbf{Shuffle Data} \div \textbf{Number of Map Waves}) \times \alpha \times \textbf{Shuffle Rate} \quad \textbf{(3)}$$

Finally, as demonstrated in Fig. 6, the runtime of a job can be defined in terms of *MST*, *HST*, *EST* and the Reduce stage time. In particular, the runtime of a job can be defined as follows:

$$\textbf{Runtime} = \textbf{Single Map Wave Time} + \textbf{Total MST} + \textbf{HST} + \textbf{EST} + \textbf{Reduce Time} \quad \textbf{(4)}$$

As its name suggests, *Single Map Wave Time* in equation (4) is the time taken by a single map wave in a job. Besides, *Reduce Time* is the time needed for the Reduce stage to apply the user-defined reduce function on a merged input partition. Clearly, by using equation (4), the runtime of any job can be estimated if the *Number of Map Waves*, *Shuffle Data*, *Shuffle Rate*, *MST*, *Single Map Wave Time* and *Reduce Time* factors are all provided. We next discuss how such a model can be effectively used to locate the best number of map waves for any given MapReduce application.

V. MC^2: Map Concurrency Characterization

Our developed mathematical model can be utilized to predict the best number of map waves for MapReduce applications. As stated earlier, by inspecting the *Total MST*, *HST*, *EST* and *Runtime* equations, we realize that six factors are required before such equations can be evaluated. The key factor among these six factors is the number of map waves, which we are actually seeking for. We note that as our objective is not to estimate runtimes of applications, but rather to predict the best number of map waves for a given application: (1) we can fix all the model's factors except the *Number of Map Waves* one, and (2) measure the *Runtime* equation for a range of map

wave numbers (e.g., 1.0 to 15.0) and select the minimum. In principle, as we attempt to optimize performance, the minimum runtime will always provide the best number of map waves. We propose Map Concurrency Characterization (MC^2), a predictor that effectively realizes such an objective.

A notice, however, is that as we vary the number of map waves to measure the *Runtime* equation, the *Single Map Wave Time* factor in the equation also indirectly varies, but inversely. In particular, as the number of map waves increases, the single map wave time decreases and vice versa. As such, to use the *Runtime* equation the way we suggest in MC^2, the *Single Map Wave Time* factor should be divided by the *Number of Map Waves* factor. In addition, the *Single Map Wave Time* changes not only with different numbers of map waves, but also with different cluster map slot configurations. Specifically, the map wave gets larger with a larger number of total map slots and smaller otherwise. Assuming enough available cluster resources, a larger map wave is supposed to take less time than a smaller one. Therefore, in order to apply MC^2 to different cluster map slot configurations, the *Single Map Wave Time* should be further multiplied by a factor, β, that is capable of capturing such a fact. Simply, the factor β can be defined as the total number of map slots that we begin with before start varying the number of map waves, divided by the variable total number of map slots as the number of map waves is varied[5]. Clearly, as the total number of map slots that we begin with is fixed, and the variable total number of map slots increases, β decreases. When β decreases, the *Single Map Wave Time* also decreases and vice versa. Consequently, the β factor can satisfy the goal of correctly changing the single map wave time as the number of map slots is altered. In short, with MC^2 the *Single Map Wave Time* becomes equal to (*Single Map Wave Time* $\times$ β $\div$ *Number of Map Waves*).

Fig. 6 depicts a high-level view of MC^2 with the required input parameters and the proposed output values. As illustrated in the figure, MC^2 requires *Shuffle Data*, *Shuffle Rate*, *MST*, *Single Map Wave Time*, *Reduce Time* and *Initial Map Slots Number* as input parameters. The *Initial Map Slots Number* is the total number of map slots that we begin with before start varying the number of map waves to locate the best one. The rest of the parameters are defined as in equations (1), (2), (3), and (4). The output curves of MC^2 reflect the values of the *Runtime*, *Total MST*, *EST* and *HST* equations for a range of map wave numbers. By scaling up the number of map waves, *HST* (or the gain from early shuffle) keeps increasing as long as it does not get dominated by *MST* (or the loss from map task setup overhead). *HST* and *EST* are inversely proportional, thus when *HST* increases, *EST* decreases. The local minimum of the *Runtime* curve is the spot at which *HST* is maximally leveraged and *MST* is minimally incurred. We denote this spot as the *sweet spot*, or the best number of map waves for the given application.

The input application parameters *Shuffle Data*, *MST*, *Single Map Wave Time*, *Reduce Time* and *Initial Map Slots Number* can be figured out by applying static profiling. First, *Shuffle Data* is independent of map concurrency. In particular, the same amount of data will be shuffled in an application irrespective of the configured number of map waves. Accordingly, a single run of the application under the default Hadoop configuration would suffice to acquire a representative value for *Shuffle Data*. *Shuffle Data* is the intermediate output size and can be collected from a Hadoop built-in counter. Second, as we aim to predict the best number of map waves and not the application's runtime, *Single Map Wave Time* and *Reduce Time* need not be fully accurate and can be approximated. Specifically, from a single run of the application, using the default Hadoop configuration, we can compute the number of map waves and divide the application's map time by this number of waves to obtain *Single Map Wave Time*. The *Reduce Time* can be approximated by selecting any reduce task in the Hadoop web user interface (UI) (or logs) and subtract the task's finish time from its shuffle end time. Third, *Initial Map Slots Number* is simply the total number of map slots of the Hadoop cluster used in running the given application (i.e., acquiring the profile). Fourth, *MST* can be approximated by a small number of seconds (e.g., 5 or 10 seconds) [17]. Lastly, *Shuffle Rate* is the data transfer rate of the underlying cluster's network.

[5]Different numbers of map waves could entail different numbers of map slots.

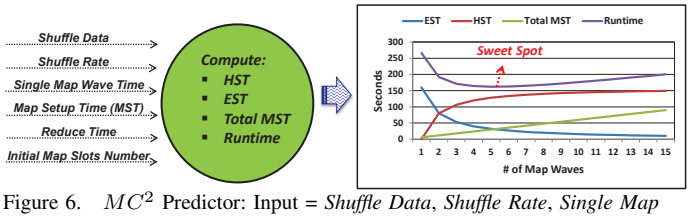

Figure 6. MC^2 Predictor: Input = *Shuffle Data*, *Shuffle Rate*, *Single Map Wave Time*, *Map Setup Time* (*MST*), *Reduce Time*, and *Initial Map Slots Number*. Output = Expect responses of Hidden Shuffle Time (*HST*), Expose Shuffle Time (*EST*), Total *MST* and *Runtime* as map concurrency is varied. The **sweet spot** is the best number of map waves (or implicitly, the best HDFS block size) for the given application.

To this end, we note that MC^2 scans a range of map wave numbers without involving Hadoop. In particular, all MC^2's computations occur without executing Hadoop, hence, the definition ***standalone utility program***. In MC^2, the range of map wave numbers is configurable and can be statically set by users. Finally, once the best number of map waves for a specific application is located, the respective HDFS block size can be easily calculated and, subsequently, configured in Hadoop (see Section II-B). The HDFS block size is a job, not a cluster, configuration parameter and can be set differently for different MapReduce applications. For instance, two applications can set HDFS block sizes to 512MB and 64MB, respectively and run on the same Hadoop cluster.

VI. Quantitative Evaluation

A. Methodology

We evaluate MC^2 on our cloud computing infrastructure and on Amazon EC2 [1]. Our infrastructure is comprised of a dedicated 14 physical host IBM BladeCenter H with identical hardware, software and network capabilities. The BladeCenter is configured with the VMware vSphere 4.1 virtualization environment and VMware ESXi 4.1 hypervisor [34]. The vSphere system is configured with a single virtual machine (VM) per each BladeCenter blade. Each VM is configured with 8 v-CPUs and 8GB of RAM. The disk storage per each VM is provided via two locally connected 300GB SAS disks. The major system software on each VM is 64-bit Fedora 13 [10], Apache Hadoop 0.20.2 [13] and Sun/Oracle's JDK 1.6 [9], Update 20. Table I summarizes the configuration of our private cloud.

Table I
OUR PRIVATE TESTBED.

Category	Configuration
Hardware	
Chassis	IBM BladeCenter H
Number of Blades	14
Processors/Blade	2 x 2.5GHz Intel Xeon Quad Core (E5420)
RAM/Blade	8 GB RAM
Storage/Blade	2 x 300 GB SAS Defined as 600 GB RAID 0
Number of Switches	3 (organized in a tree-style way)
Software	
Virtualization Platform	vSphere 4.1/ESXi 4.1
VM Parameters	8 vCPU, 8 GB RAM
	1 GB NIC
	60 GB Disk (mounted at /)
	450 GB Disk (mounted at /hadoop)
OS	64-Bit Fedora 13
JVM	Sun/Oracle JDK 1.6, Update 20
Hadoop	Apache Hadoop 0.20.2

Contrary to our private cluster, Amazon EC2 is a shared heterogeneous cloud. We provisioned on Amazon EC2 a Hadoop cluster with standard 20 large (i.e., *m1.large*) instances, each with 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units), 7.5GB memory, 850GB instance storage, high I/O Performance, and 64-bit Ubuntu AMI image. We adopted an Amazon EC2 cluster with a size different than that of our private one so as to demonstrate MC^2's versatility.

To account for various shuffle data sizes, we use WordCount with the combiner function being enabled (WordCount-CE) and disabled (WordCount-CD). In addition, we use the Apache Mahout *K*-Means clustering workload [29], Sort and an in-house developed image processing benchmark, Sobel. Sort and WordCount are two main benchmarks utilized for evaluating Hadoop at Yahoo! [8], [38]. *K*-Means is a well-known clustering algorithm for knowledge discovery and data mining [20]. Sobel is a state-of-the-art edge detection algorithm used widely in various scientific domains such as bio-medicine [25] (Sobel is part of NIH's ImageJ package [23]) and astronomy [30], among others.

On our private cloud, we ran Sort over a 28GB dataset generated using the RandomWriter in Hadoop. WordCount-CE and WordCount-CD were run over 28GB and 14GB datasets, respectively, generated using the RandomTextWriter in Hadoop. For *K*-Means we generated a random 5.5GB dataset of 2D data points. Besides, we selected 4 random centroids and fixed that for all runs. Per each run we set 5 iterative jobs similar to [20]. Finally, for Sobel we obtained an image dataset from celebi [6], the public repository of medical imaging. Our image dataset consists of 30080 dynamic high-resolution 4D PNG images (the original images were in the Dicom format) from a cardiac study acquired on a 64 detector CT scanner. After bundling all the images in a sequence file (as required by Hadoop), we obtained a total image dataset size of 4.3GB.

On Amazon EC2, we ran Sort, WordCount-CE and WordCount-CD over datasets of size 20GB, generated using the RandomWriter and the RandomTextWriter in Hadoop. For *K*-Means and Sobel we utilized similar datasets as ones used on our private cloud, but with sizes of 11.1GB and 8.7GB (or 60168 images), respectively. We varied dataset sizes in order to better test and verify the promise of MC^2, especially with the shuffle data being one of MC^2's inputs (the size of the dataset influences the amount of data to shuffle). To this end, we accounted for variances across runs by running each benchmark 3 times on our private cloud and 5 times on Amazon EC2, and reported the average.

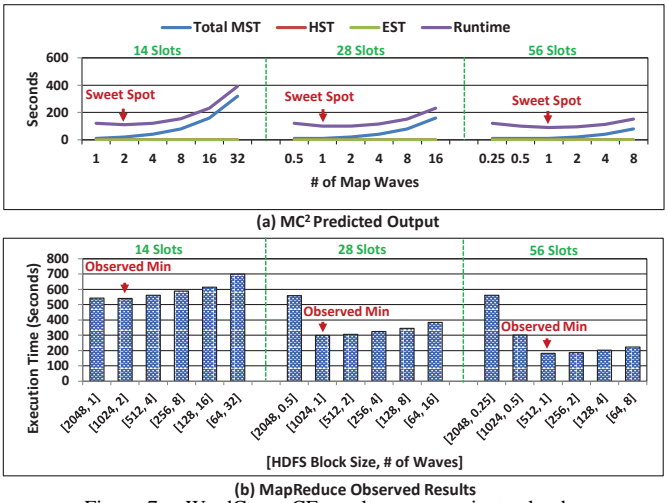

Figure 7. WordCount-CE results on our private cloud.

B. MC^2 on Our Private Cloud

We first evaluate MC^2 by: (1) running each benchmark on our private cloud with different numbers of map waves, (2) collecting results and locating the best number of map waves of each benchmark, (3) using the MC^2 standalone utility program to predict each benchmark's best number of map waves (i.e., the ***sweet spot***), and (4) matching the best empirically located number of map waves, or what we refer to as the ***observed minimum***, with the sweet spot of each benchmark. To execute MC^2 we use a single static profile for each workload collected under the default Hadoop configuration. The required MC^2 input parameter values are obtained in a way similar to what is described in Section V. *MST* is approximated to 10 seconds for all benchmarks[6]. As the number of map waves is defined in terms of the numbers of map tasks and slots, we ran each benchmark on our private cloud with 1, 2, and 4 map slots per TaskTracker (or 14, 28, and 56 total map slots). Furthermore, we used HDFS blocks of sizes 2048MB or 1024MB, 512MB, 256MB, 128MB and 64MB[7]. As discussed in Section II-B, by varying the sizes of HDFS blocks, we vary the number of map tasks. Figures 7, 8, 9, 10 and 11 demonstrate the results for WordCount-CE, K-Means, Sort, WordCount-CD and Sobel, respectively.

Let us start with a small recap on MC^2. MC^2 does not predict the actual runtimes of MapReduce applications (i.e., how many seconds a certain application will take to finish), but rather the optimal map concurrencies for applications. When the best map concurrency for an application is located and, subsequently, configured on Hadoop, it translates to an overall runtime reduction. In addition, as described in Section V, MC^2 uses approximated values for its input parameters to compute the *Total MST*, *HST*, *EST* and *Runtime*

[6]A more accurate approach is to approximate a different *MST* for different benchmarks using static profiling. For simplicity, in this work we assume a single *MST* across all our workloads, especially that they all exhibit long-lived tasks. For jobs that finish in less than 1 minute, we recommend using different approximated *MST*s for different jobs.

[7]We use ranges of map slots and HDFS block sizes that are typically utilized in Hadoop settings at small and large scale levels.

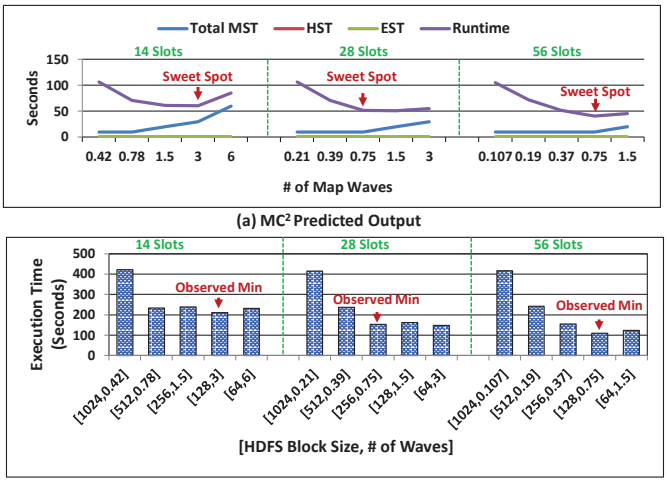

Figure 8. K-Means results on our private cloud.

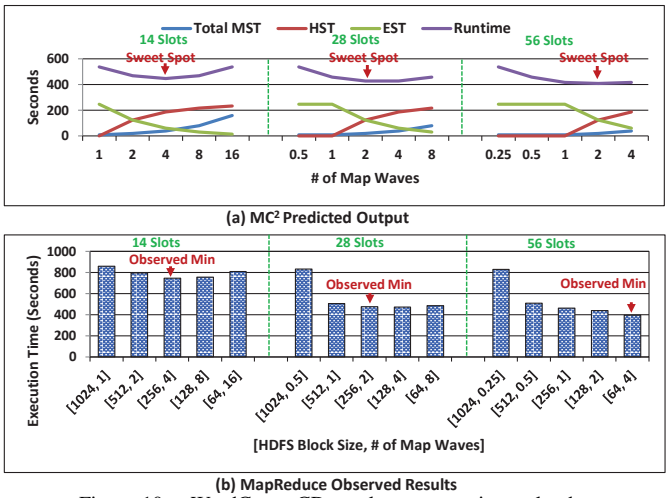

Figure 10. WordCount-CD results on our private cloud.

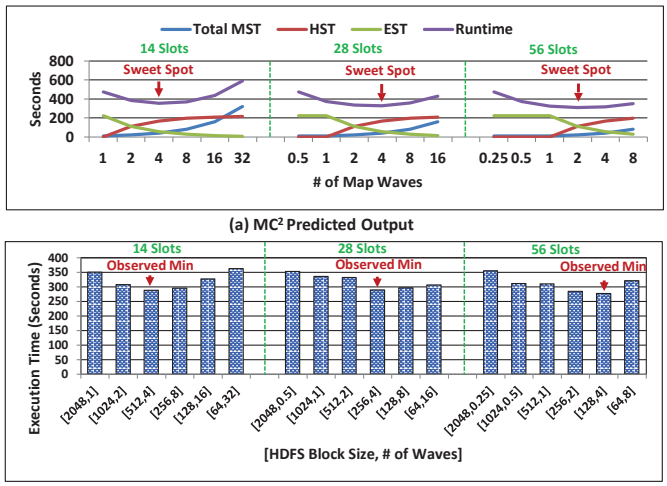

Figure 9. Sort results on our private cloud.

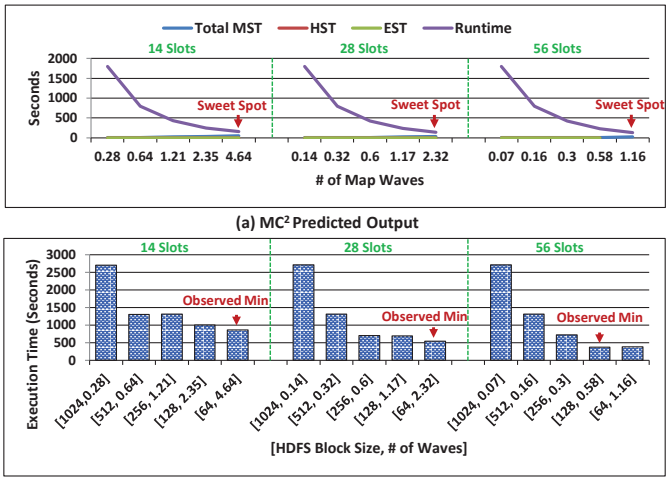

Figure 11. Sobel results on our private cloud.

equations. Accordingly, as shown in Figures 7, 8, 9, 10 and 11, the y-axes of MC^2 predicted outputs do not closely match the y-axes of MapReduce observed results.

As demonstrated in Fig. 7, MC^2 correctly predicts the best numbers of map waves for WordCount-CE under all various configurations. In particular, MC^2 predicts 2, 1, and 1 map waves for the 14, 28 and 56 map slot configurations, respectively. Some map wave numbers in the figure are in decimal because the remainder of dividing the number of HDFS blocks (or map tasks) of WordCount-CE's dataset (i.e., 28GB) by the cluster's total number of map slots is not always zero. The exhibited observed minimums in Fig. 7 (b) match exactly the MC^2 predicted sweet spots. Moreover, the configuration corresponding to the *best*[8] sweet spot (i.e., the 56 map slot and 512MB block configuration) greatly outperforms the default Hadoop configuration (i.e., the 28 map slot and 64MB block configuration). Specifically, the best sweet spot provides a speedup of 2.1X versus default Hadoop. Clearly, this shows that the default Hadoop configuration is not necessarily the optimal and substantiates the effectiveness of MC^2 in locating the best number of map waves for WordCount-CE.

Likewise, Figures 8, 9, 10 and 11 show that MC^2 successfully predicts the best numbers of map waves for K-Means, Sort, WordCount-CD and Sobel under all various configurations, except under the 56 map slot configuration for Sort, WordCount-CD and Sobel. Although these miss-predictions cause performance degradations of 2.4%, 10.8% and 0.7% for Sort, WordCount-CD and Sobel, respectively versus the configurations under the observed minimums, they do not incur degradations versus default Hadoop. In fact, the Hadoop configurations under the miss-predicted sweet spots outperform default Hadoop by 7.9%, 10.2%, and 43.8% for Sort, WordCount-CD and Sobel, respectively. As depicted in Figures 7, 8, 9, 10 and 11, there is indeed no single degradation under any of the best sweet spot configurations against default Hadoop. In summary, MC^2 provides speedups of 2.1X, 1.34X, 1.07X, 1.1X, and 1.43X versus default Hadoop for WordCount-CE, K-Means, Sort, WordCount-Cd and Sobel, respectively. The speedups vary according to where the default Hadoop configuration is from the best sweet spot.

MC^2 might sometimes result in miss-predictions for two

[8]The best sweet spot of an application is the minimum sweet spot among all the located sweet spots of the application.

main reasons. First, the mathematical model that MC^2 utilizes assumes that map tasks start and finish at comparable times. This is not always true, especially when some tasks render slow. Hadoop addresses slow tasks by launching corresponding speculative tasks so as to avert job delays. Incorporating effects of speculative execution within our mathematical model is a promising and worth-exploring future direction. Second, our mathematical model assumes that resource contention is avoided. For the infrastructure of our private cloud and the provisioned virtual machines, setting 4 or more map slots might stress resources; hence, the observed miss-predictions with Sort, WordCount-CD and Sobel under the 56 map slot configuration (the 56 map slot configuration uses 4 map slots per a TaskTracker). In contrary, WordCount-CE and K-Means do not incur miss-predictions under such a configuration due to being less CPU, memory and I/O bound than Sort, WordCount-CD and Sobel. Finally, we note that even if a miss-prediction occurs, it is typically the case that the missed correct sweet spot (gleaned from the observed minimum) is very close to the miss-predicted sweet spot at the *Runtime* curves. As such, users can always conjecture what potential degradation (if any) they might experience if a miss to the correct sweet spot happens. MC^2 users are always expected to observe speedups versus default Hadoop, unless default Hadoop is the optimal configuration and a miss-prediction occurs. We next evaluate MC^2 on Amazon EC2.

C. MC^2 on Amazon EC2

Table III
RUNTIME SPEEDUPS PROVIDED BY MC^2 VERSUS DEFAULT HADOOP.

Benchmark	Private Cloud	Amazon EC2
WordCount-CE	2.1X	1.2X
K-Means	1.34X	1.13X
Sort	1.07X	1.1X
WordCount-CD	1.1X	2.2X
Sobel	1.43X	1.04X

To evaluate MC^2 on a shared heterogeneous environment, we ran each of our benchmarks on a Hadoop cluster composed of 20 Amazon EC2 large instances. Each instance was configured with 2 and 4 map slots (or totals of 20 and 40 map slots). Besides, HDFS block sizes of 1024MB, 512MB, 256MB, 128MB and 64MB were used. For this set of experiments, we approximated *MST* to 2 seconds across all our benchmarks, assuming less initialization time because of the chosen faster virtual machines on Amazon EC2. Due to page constraints, we summarize all our MC^2 and Amazon EC2 results in Table II. As shown in the table, MC^2 correctly predicts the best numbers of map waves for WordCount-CE, K-Means, Sort, WordCount-CD and Sobel under all various configurations, except under the 80 map slot configuration for WordCount-CE, K-Means and Sobel. The miss-predicted sweet spots lead to 0.53%, 16.2% and 2.2% performance degradations as compared to the observed minimums for the three benchmarks, respectively. Nonetheless, the miss-predicted sweet spots result in 16.3%, -9.9%[9] and 4.4% performance improvements/degradations for WordCount-CE, K-Means and Sobel, respectively versus default Hadoop. In summary, on Amazon EC2 we did not observe any degradation under any of the best sweet spot configurations (which users would select) against default Hadoop. In particular, MC^2 provided speedups of 1.2X, 1.13X, 1.1X, 2.2X, and 1.04X over default Hadoop for WordCount-CE, K-Means, Sort, WordCount-CD and Sobel, respectively. Again, the speedups vary according to where the default Hadoop configuration is from the best sweet spot. To this end, Table III exhibits the runtime speedups provided by MC^2 over default Hadoop for all our benchmarks on Amazon EC2 and on our private cloud.

VII. RELATED WORK

There has been recently a large body of work that focused on optimizing Hadoop configuration for improved performance. In this short article, it is not possible to do justice to every related work. Hence, we only outline proposals that are most relevant to MC^2.

Babu makes a case for techniques to automate the process of configuring MapReduce parameters [4]. He discusses the applicability of different approaches (e.g., the database query-optimizer-style approach) to meet this goal. Rizvandi *et al.* present a preliminary step towards modeling the relationship between the number of map/reduce tasks and application runtimes [33]. They suggest employing a multivariate linear regression model to predict MapReduce performance (or CPU utilization as in [31]) for various applications. Yang *et al.* evaluate the correlation between application characteristics, configuration parameters and workload execution times [37]. Similar to [33] and [31], they suggest using a regression model. Ganapathi *et al.* propose using Kernel Canonical Correlation Analysis (KCCA) to predict runtimes of (only) Hive queries [11]. To aid users in making better use of cloud resources, Wieder *et al.* suggest utilizing dynamic linear programming to model each phase in MapReduce independently and, accordingly, determine optimal scheduling and resource configurations [35], [36]. As compared to [4], [11], [31], [33], [35]–[37], MC^2 uses no regression, KCCA or dynamic programming models. In contrary, MC^2 depends uniquely on only MapReduce concurrency characteristics and guides accurately Map phase configuration within milliseconds.

To automatically find good configuration settings for arbitrary MapReduce jobs, Herodotou and Babu introduce the Cost-Based Optimizer (CBO) [17]. CBO assumes two components: (1) a profiler and (2) a what-if engine. The profiler uses dynamic instrumentation to collect runtime monitoring information from unmodified MapReduce programs. The what-if engine utilizes a mix of simulation and model-based estimation to answer questions about job executions. To optimize Hadoop configuration, CBO enumerates and searches through the parameter space of Hadoop, and makes appropriate calls to the what-if engine. Herodotou *et al.* present Elastisizer, a system to which users can express

[9]In this case (i.e., for K-Means) the runtime provided by default Hadoop was even better than the one provided by the observed minimum with 80 map slots, and not with 40 map slots (which is the default number of map slots). Indeed, the located sweet spot under the 40 map slot configuration offered Hadoop a performance improvement of 13.1% versus default Hadoop. This sweet spot is the best sweet spot generated by MC^2 for K-Means, and which the users will naturally select.

Table II
SUMMARIZED RESULTS ON AMAZON EC2.

Benchmark	# of Map Slots	[HDFS Block Size, # of Map Waves] Range	Observed Minimum	Predicted Sweet Spot
WordCount-CE	40	[1024, 0.5], [512, 1], [256, 2], [128, 4], [64, 8]	[512, 1]	[512, 1]
	80	[1024, 0.25], [512, 0.5], [256, 1], [128, 2], [64, 4]	[512, 0.5]	[256, 1]
K-Means	40	[1024, 0.27], [512, 0.52], [256, 1.05], [128, 2.1], [64, 4.17]	[128, 2.1]	[128, 2.1]
	80	[1024, 2.08], [512, 1.05], [256, 0.52], [128, 0.26], [64, 0.27]	[64, 0.27]	[256, 0.52]
Sort	40	[1024, 0.5], [512, 1], [256, 2], [128, 4], [64, 8]	[128, 4]	[128, 4]
	80	[1024, 0.25], [512, 0.5], [256, 1], [128, 2], [64, 4]	[64, 4]	[64, 4]
WordCount-CD	40	[1024, 0.5], [512, 1], [256, 2], [128, 4], [64, 8]	[128, 4]	[128, 4]
	80	[1024, 0.25], [512, 0.5], [256, 1], [128, 2], [64, 4]	[64, 4]	[64, 4]
Sobel	40	[1024, 0.22], [512, 0.42], [256, 0.82], [128, 1.62], [64, 3.25]	[64, 3.25]	[64, 3.25]
	80	[1024, 0.11], [512, 0.21], [256, 0.41], [128, 0.81], [64, 1.62]	[128, 0.81]	[64, 1.62]

the problem of determining cluster resources and MapReduce configurations that best meet performance and cost needs [18]. Elastisizer makes use of the what-if engine proposed in [17].

Guided by the work on self-tuning database systems, Herodotou *et al.* propose Starfish, a self-tuning system for big data analytics [19]. Starfish exploits Elastisizer [18] to automate Hadoop provisioning decisions. In essence, the CBO, Elastisizer and Starfish schemes pose main challenges such as developing efficient strategies to search through the high-dimensional parameter space of Hadoop, and generating job profiles with minimal overhead. Moreover, they rely on a mix of mechanisms (e.g., simulation and model-based estimation) to find good configurations. In contrast, MC^2 requires no simulation, dynamic instrumentation and/or sampling, and simply depends on some MapReduce characteristics to optimize map concurrency.

To effectively configure MapReduce parameters, Koehler *et al.* propose the usage of an adaptive framework, which depends on autonomic computing concepts and utility functions [28]. As per the Map phase, the framework focuses on optimizing the number of map slots at cluster nodes. Similarly, Kambatla *et al.* suggest a signature-based predictor to predict the optimum number of map and reduce slots at Hadoop cluster nodes [27]. As compared to [28] and [27], MC^2 focuses on map concurrency in terms of the number of map tasks for a *given* number of map slots. Specifically, MC^2 does not estimate the optimal number of map slots for a Hadoop cluster.

VIII. CONCLUDING REMARKS AND FUTURE DIRECTIONS

In this work we observed a strong dependency between map concurrency and MapReduce performance. We realized that a good configuration for map concurrency can be determined by simply depending on two main MapReduce characteristics, data shuffling and map task setup overhead. We built MC^2, a simple standalone utility predictor that leverages these two characteristics and correctly predict the best map concurrencies for MapReduce applications. We showed that MC^2 works successfully on a private cloud and on Amazon EC2. MC^2 makes timely contributions to cloud data analytics, and serves in reducing cost and improving MapReduce performance on the cloud.

After verifying the promise of MC^2, we set forth three main future directions: (1) extending MC^2 to predict the best possible number of map slots for Hadoop clusters so as to avoid resource contention and enhance system utilization, (2) incorporating the effects of speculative execution in our mathematical model in order to make it more robust to slow tasks, and (3) characterizing reduce concurrency so as to guide Reduce phase configuration for improved performance.

REFERENCES

[1] Amazon Elastic Compute Cloud, "http://aws.amazon.com/ec2/."

[2] Amazon Elastic MapReduce, "http://aws.amazon.com/elasticmapreduce/."

[3] Amazon EC2 Instance Types, "http://aws.amazon.com/ec2/instance-types/."

[4] S. Babu, "Towards Automatic Optimization of MapReduce Programs," *SOCC,* 2010.

[5] BTrace: A Dynamic Instrumentation Tool for Java, "http://kenai.com/projects/btrace"

[6] Celebi Software, "http://www.celebisoftware.com/."

[7] S. Chen and S.W. Schlosser, "Map-Reduce Meets Wider Varieties of Applications," *Intel Research Pittsburgh, Tech. Rep. IRP-TR-08-05,* May 2008.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing On Large Clusters," *OSDI,* Dec. 2004.

[9] "http://download.oracle.com/javase/6/docs/."

[10] "http://fedoraproject.org/."

[11] A. Ganapathi *et al.*, "Statistics-Driven Workload Modeling for the Cloud," *ICDEW,* 2010.

[12] "http://ganglia.sourceforge.net/."

[13] Hadoop, "http://hadoop.apache.org/."

[14] Hadoop Tutorial, "http://developer.yahoo.com/hadoop/tutorial/."

[15] M. Hammoud and M.F. Sakr, "Locality-Aware Reduce Task Scheduling for MapReduce," *CloudCom,* 2011.

[16] M. Hammoud *et al.*, "Center-of-Gravity Reduce Task Scheduling to Lower MapReduce Network Traffic," *CLOUD,* 2012.

[17] H. Herodotou and S. Babu, "Profiling, What-If Analysis, and Cost-Based Optimization of MapReduce Programs," *VLDB,* 2011.

[18] H. Herodotou *et al.*, "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics," *SOCC,* 2011.

[19] H. Herodotou *et al.*, "Starfish: A Self-Tuning System for Big Data Analytics," *CIDR,* 2011.

[20] S. Huang *et al.*, "The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis," *ICDEW,* 2010.

[21] S. Ghemawat, *et al.*, "The Google File System," *SOSP,* Oct. 2003.

[22] S. Ibrahim, *et al.*, "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud," *CloudComm,* Dec. 2010.

[23] NIH ImageJ, "http://rsbweb.nih.gov/ij/"

[24] Impetus, "Whitepaper: Deriving Intelligence from Large Data Using Hadoop and Applying Analytics."

[25] I. N. Bankman, "Handbook of Medical Image Processing and Analysis," *Elsevier; 2nd Edition,* 2009.

[26] D. Jiang *et al.*, "The Performance of MapReduce: An In-Depth Study," *VLDB,* 2010.

[27] K. Kambatla *et al.*, "Towards Optimizing Hadoop Provisioning in the Cloud," *HotCloud,* 2009.

[28] M. Koehler *et al.*, "An Adaptive Framework for the Execution of Data-Intensive MapReduce Applications in the Cloud," *IPDPSW,* 2011.

[29] Mahout Homepage, "http://mahout.apache.org/."

[30] J.L. Starck and F. Murtagh, "Handbook of Astronomical Data Analysis," *Springer-Verlag; 2nd Edition* 2006.

[31] N.B. Rizvandi *et al.*, "Preliminary Results on Modeling CPU Utilization of MapReduce Programs," *Tech. R. 665, The University of Sydney,* 2011.

[32] N.B. Rizvandi *et al.*, "On Using Pattern Matching Algorithms in MapReduce Applications," *ISPA,* 2011.

[33] N.B. Rizvandi *et al.*, "Preliminary Results: Modeling Relation between Total Execution Time of MapReduce Applications and Number of Mappers/Reducers," *Tech. R. 679, The University of Sydney,* 2011.

[34] "http://www.vmware.com/."

[35] A. Wieder *et al.*, "Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud" *SIGACT-SIGOPS,* 2010.

[36] A. Wieder *et al.*, "Conductor: Orchestrating the Clouds" *LADIS,* 2010.

[37] H. Yang *et al.*, "MapReduce Workload Modeling with Statistical Approach," *Journal of Grid Computing,* 2012.

[38] M. Zaharia *et al.*, "Improving Mapreduce Performance in Heterogeneous Environments," *OSDI,* 2008.