PRISM: Fine-Grained Resource-Aware Scheduling for MapReduce

Qi Zhang, *Student Member, IEEE*, Mohamed Faten Zhani, *Member, IEEE*, Yuke Yang, Raouf Boutaba, *Fellow, IEEE*, and Bernard Wong

Abstract—MapReduce has become a popular model for data-intensive computation in recent years. By breaking down each job into small *map* and *reduce* tasks and executing them in parallel across a large number of machines, MapReduce can significantly reduce the running time of data-intensive jobs. However, despite recent efforts toward designing resource-efficient MapReduce schedulers, existing solutions that focus on scheduling at the task-level still offer sub-optimal job performance. This is because tasks can have highly varying resource requirements during their lifetime, which makes it difficult for task-level schedulers to effectively utilize available resources to reduce job execution time. To address this limitation, we introduce PRISM, a fine-grained resource-aware MapReduce scheduler that divides tasks into phases, where each phase has a constant resource usage profile, and performs scheduling at the phase level. We first demonstrate the importance of phase-level scheduling by showing the resource usage variability within the lifetime of a task using a wide-range of MapReduce jobs. We then present a phase-level scheduling algorithm that improves execution parallelism and resource utilization without introducing stragglers. In a 10-node Hadoop cluster running standard benchmarks, PRISM offers high resource utilization and provides $1.3\times$ improvement in job running time compared to the current Hadoop schedulers.

Index Terms—Cloud computing, MapReduce, Hadoop, scheduling, resource allocation

1 Introduction

Businesses today are increasingly reliant on large-scale data analytics to make critical day-to-day business decisions. This shift towards data-driven decision making has fueled the development of MapReduce [10], a parallel programming model that has become synonymous with large-scale, data-intensive computation. In MapReduce, a job is a collection of *Map* and *Reduce* tasks that can be scheduled concurrently on multiple machines, resulting in significant reduction in job running time. Many large companies, such as Google, Facebook, and Yahoo!, routinely use MapReduce to process large volumes of data on a daily basis. Consequently, the performance and efficiency of MapReduce frameworks have become critical to the success of today's Internet companies.

A central component to a MapReduce system is its job scheduler. Its role is to create a schedule of Map and Reduce tasks, spanning one or more jobs, that minimizes job completion time and maximizes resource utilization. A schedule with too many concurrently running tasks on a single machine will result in heavy resource contention and long job completion time. Conversely, a schedule with too few concurrently running tasks on a single machine will cause the machine to have poor resource utilization.

The job scheduling problem becomes significantly easier to solve if we can assume that all map tasks (and similarly, all reduce tasks) have homogenous resource requirements in terms of CPU, memory, disk and network bandwidth. Indeed, current MapReduce systems, such as Hadoop MapReduce Version $1.x$, make this assumption to simplify the scheduling problem. These systems use a simple slot-based resource allocation scheme, where physical resources on each machine are captured by the number of identical slots that can be assigned to tasks. Unfortunately, in practice, run-time resource consumption varies from task to task and from job to job. Several recent studies have reported that production workloads often have diverse utilization profiles and performance requirements [8], [20]. Failing to consider these job usage characteristics can potentially lead to inefficient job schedules with low resource utilization and long job execution time.

Motivated by this observation, several recent proposals, such as resource-aware adaptive scheduling (RAS) [15] and Hadoop MapReduce Version 2 (also known as Hadoop NextGen and Hadoop Yarn) [7], have introduced resource-aware job schedulers to the MapReduce framework. However, these schedulers specify a fixed size for each task in terms of required resources (e.g. CPU and memory), thus assuming the run-time resource consumption of the task is stable over its life time. However, this is not true for many MapReduce jobs. In particular, it has been reported that the execution of each MapReduce task can be divided into multiple phases of data transfer, processing and storage [12]. A *phase* is a sub-procedure in the task that has a distinct purpose and can be characterized by the uniform resource consumption over its duration. As we shall demonstrate in Section 2.2, the phases involved in the same task can have different resource demand in terms of CPU, memory, disk and network usage. Therefore, scheduling tasks based on fixed resource requirements over their durations will often

• *The authors are with the David. R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada. E-mail: {q8zhang, mfzhani, y274yang, rboutaba, bernard}@uwaterloo.ca.*

Manuscript received 21 Mar. 2014; revised 9 Oct. 2014; accepted 14 Nov. 2014. Date of publication 8 Jan. 2015; date of current version 10 June 2015. Recommended for acceptance by R. Ranjan, L. Wang, A. Zomaya, D. Georgakopoulos, G. Wang, and X.-H. Sun. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TCC.2014.2379096

cause either excessive resource contention by scheduling too many simultaneous tasks on a machine, or low utilization by scheduling too few.

In this paper, we present PRISM, a *P*hase and *R*esource *I*nformation-aware *S*cheduler for *M*apReduce clusters that performs resource-aware scheduling at the level of task phases. Specifically, we show that for most MapReduce applications, the run-time task resource consumption can vary significantly from phase to phase. Therefore, by considering the resource demand at the phase level, it is possible for the scheduler to achieve higher degrees of parallelism while avoiding resource contention. To this end, we have developed a phase-level scheduling algorithm with the aim of achieving high job performance and resource utilization. Through experiments using a real MapReduce cluster running a wide-range of workloads, we show PRISM delivers up to 18 percent improvement in resource utilization while allowing jobs to complete up to $1.3\times$ faster than current Hadoop schedulers. Finally, even though PRISM is currently designed for Hadoop MapReduce, we believe our solution can be applied to Dryad [19] and other parallel computing frameworks as well.

The rest of this paper is organized as follows. Section 2 provides a basic overview of MapReduce scheduling and job execution. We describe the phase-level task usage characteristics and our motivation in Section 3. Section 4 introduces PRISM and describes its architecture. The phase-level scheduling algorithm is presented in details in Section 5. Our experimental evaluation of PRISM is provided in Section 6. Finally, we summarize existing work related to PRISM in Section 7, and draw our conclusion in Section 8.

2 Background

This section provides an overview of Hadoop MapReduce and various phases in a MapReduce job.

2.1 Hadoop MapReduce

MapReduce [10] is a parallel computing model for large-scale data-intensive computations. A MapReduce job consists of two types of tasks, namely map and reduce tasks. A map task takes as input a key-value block stored in the underlying distributed file system and runs a user-specified map function to generate intermediary key-value output. Subsequently, a reduce task is responsible for collecting and applying a user-specified reduce function on the collected key-value pairs to produce the final output.

Currently, the most popular implementation of MapReduce is Apache Hadoop MapReduce [1]. A Hadoop cluster consists of a large number of commodity machines with one node serving as the master and the others acting as slaves. The master node runs a resource manager (also known as a job tracker) that is responsible for scheduling tasks on slave nodes. Each slave node runs a local node manager (also known as a task tracker) that is responsible for launching and allocating resources for each task. To do so, the task tracker launches a Java Virtual Machine (JVM) that executes the corresponding map or reduce task. The original Hadoop MapReduce (i.e. version $1.x$ and earlier) adopts a slot-based resource allocation scheme. The scheduler assigns tasks to each machine based on the number of available slots on that machine. The number of map slots and reduce slots determines respectively the maximum number of map tasks and reduce tasks that can be scheduled on the machine at a given time.

As a Hadoop cluster is usually a multi-user system, many users can simultaneously submit jobs to the cluster. The job scheduling is performed by the resource manager in the master node, which maintains a list of jobs in the system. Each slave node monitors the progress of each running task and available resources on the node, and periodically (usually between 1-3 seconds) transmit a heartbeat message to convey this information to the master node. The resource scheduler will use the provided information to make scheduling decisions. Currently, Hadoop MapReduce supports several job schedulers such as the Capacity scheduler [2] and Fair scheduler [3]. These schedulers make job scheduling decisions at task level. They determine which task should be scheduled on which machine at any given time, based on the number of unoccupied slots on each machine.

While this simple slot-based allocation scheme is simple and easy to implement, it does not take run-time task resource consumption into consideration. As different tasks may have different resource requirements, this simple slot-based resource allocation scheme can lead to resource contention if the scheduler assigns multiple tasks that have high demand for a single resource.

Motivated by this observation, Hadoop Yarn (also known as the Hadoop Version 2 and Hadoop NextGen) [7] enables resource-aware task scheduling in Hadoop MapReduce clusters. While still in alpha version, it offers the ability to specify the size of the task container (i.e. a resource reservation for a task process) in terms of CPU and memory usage.[1]

2.2 MapReduce Job Phases

Current Hadoop job schedulers perform task-level scheduling, where tasks are considered as the finest granularity for scheduling. However, if we examine the execution of each task, we can find that a task consists of multiple phases, as illustrated in Fig. 1. In particular, a map task can be divided into two main phases: *map* and *merge*.[2] The input of a MapReduce job is stored as data blocks (usually of size 64 or 128 MB) in the Hadoop Distributed File System (HDFS) [4], where data blocks are stored across multiple slave nodes. In the map phase, a mapper fetches a input data block from the Hadoop Distributed File System [4] and applies the user-defined map function on each record. The map function generates records that are serialized and collected into a buffer. When the buffer becomes full (i.e., content size exceeds a pre-specified threshold), the content of the buffer will be written to the local disk. Lastly, the mapper executes a merge phase to group the output records based on the intermediary keys, and store the records in multiple files so that each file can be fetched a corresponding reducer.

Similarly, the execution of a reduce task can be divided into three phases: *shuffle*, *sort*, and *reduce*. In the shuffle phase, the reducer fetches the output file from the local

1. Other resources such as disk and network I/O are yet to be supported by Hadoop Yarn.

2. We use the same phase names as in the Hadoop implementation.

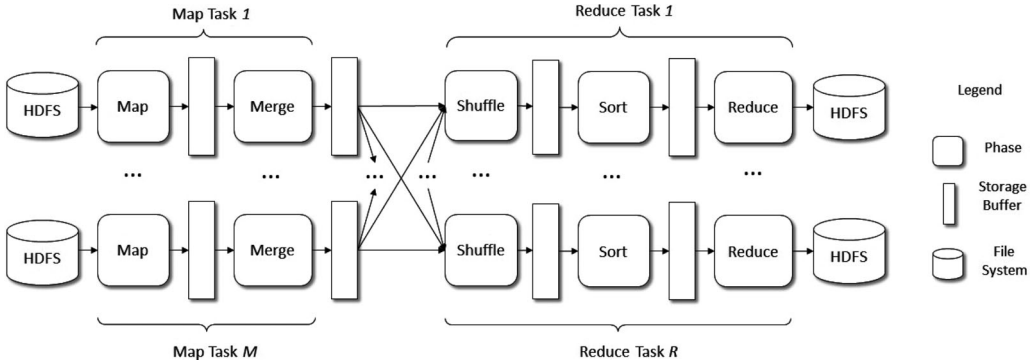


Fig. 1. Phases involved in the Execution of a Typical MapReduce Job (both version 1 and 2).

storage of each map task and then places it in a storage buffer that can be either in memory or on disk depending on the size of the content. At the same time, the reducer also launches one or more threads to perform local merge sort in order to reduce the running time of the subsequent sort phase. Once all the map output records have been collected, the sort phase will perform one final sorting procedure to ensure all collected records are in order. Finally, in the reduce phase, records are processed according the user-defined reduce function in the sorted order, and the output is written to the HDFS.

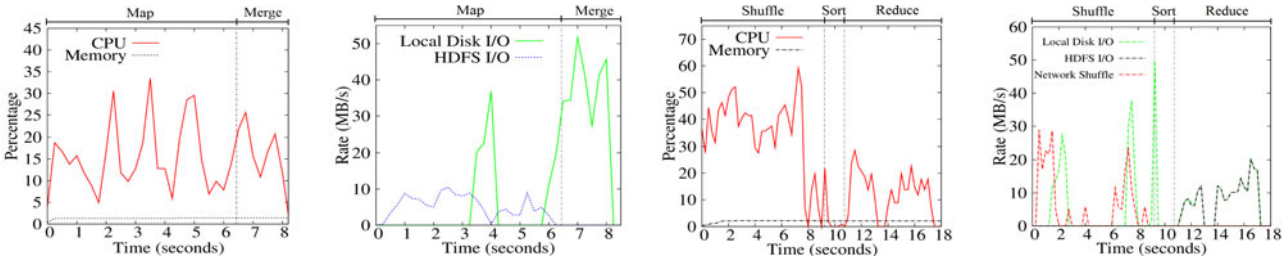
Different phases can have different resource consumption characteristics. For instance, the shuffle phase often consumes significant network I/O resources as it requires collecting outputs from all completed map tasks. In contrast, the map and reduce phases mainly process the records on local machines, thus they typically demand greater CPU resources than network bandwidth. In the next section, we provide empirical evidence to show that the run-time task resource consumption can change significantly across phase boundaries.

3 Phase-Level Resource Requirements

In this section we experimentally analyze the run-time task resource requirements in each phase for various Hadoop jobs. We deployed Apache Hadoop 0.20.2 on a 16 node cluster, with one node acting as the master managing the other 15 slave nodes. Each machine has a Quad-core Xeon CPU with 12 GB of memory and 1 TB local disk storage. We modified the default task tracker in Hadoop 0.20.2 to monitor the execution of phases inside each task.

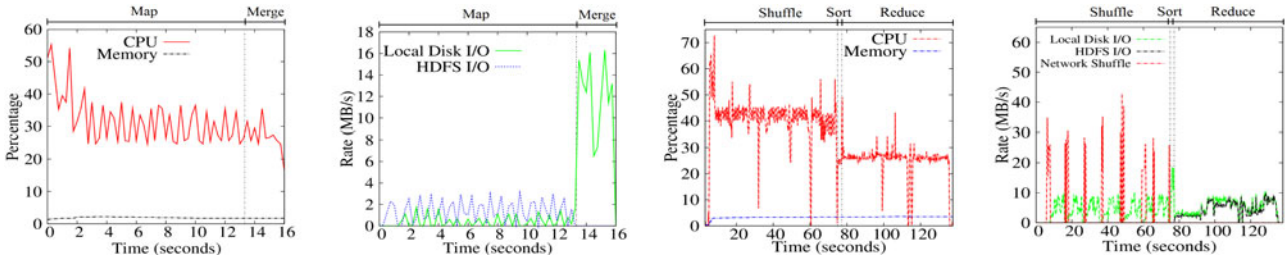
In our experiments, we evaluate the phase-level resource requirements across various jobs, including the standard examples provided by the Hadoop MapReduce distribution Gridmix2 [5] and the PUMA Benchmarks [6]. The CPU and memory usage are collected using the linux `top` command, whereas I/O usage are obtained by reading MapReduce I/O Counters at run-time. Fig. 2 shows the resource consumption over time of a single map and reduce task for the `sort` job. Despite the fact that CPU usage usually show high variances, the average CPU usage of the map tasks remain relatively stable over time as shown in Fig. 2a. However, at the same time, the I/O usage increases significantly as each the task progresses from the map phase to the merge phase. The low I/O usage is the result of the map phase incrementally reading the input key-value pairs from the HDFS system. In contrast, the merge phase has high I/O usage because it is responsible for grouping all intermediary key-values pairs within a short period of time. Similarly, Figs. 2c and 2d show that the run-time resource consumption of the reduce task changes from the shuffle phase to the reduce phase. The reason is that the shuffle phase fetches the intermediary key-values pairs from the map tasks, and performs partial merge on the fetched key-value pairs. As a result, it consumes both CPU and network I/O resources. However, once the reduce phase begins, the reducer only needs to focus on applying the reduce function to each key-value pair to produce the final output. Because the reduce function of the `sort` job is just a simple pass-through function, the CPU usage of the reduce phase is lower than that of the shuffle phase.

We also analyze the `InvertedIndex` job in the PUMA benchmark. Fig. 3 shows that the map tasks and reduce tasks of the `InvertedIndex` job have different running times compared to the `sort` job. Furthermore, unlike in the `sort` job, the map tasks of the `InvertedIndex` job consume almost $8\times$ less I/O resources during map phase than



(a) MAP CPU and memory usage (b) MAP IO Usage (c) REDUCE CPU and memory usage (d) REDUCE IO Usage

Fig. 2. Job Profile for `sort`.



(a) Map CPU and mem. usage (b) MAP IO Usage (c) REDUCE CPU and memory usage (d) REDUCE IO Usage

Fig. 3. Job Profile for `InvertedIndex`.

the merge phase. These observations suggest that the run-time task resource consumption is dependent on the phase in which the task is currently executing. Therefore, ignoring the phase-level resource characteristics will lead to poor resource allocations decisions, which in turn will cause the job scheduler to make inefficient job scheduling decisions.

We would like to mention that while it is tempting to divide certain phases into more fine-grained phases to achieve even more uniform resource usage in each phase, we found that doing so is cost prohibitive because (1)certain fine-grained phases (e.g. partition and spill [12]) are tightly coupled with each other thus scheduling them individually will require major change to the MapReduce implementation, and (2) a more fine-grained splitting as this will significantly increase the complexity of the system and the scheduling overhead may outweigh the gain attained by phase-level scheduling.

4 PRISM

The task run-time resource usage analysis described in the previous section suggests that simply allocating a fixed-sized container for each task can lead to inefficient scheduling decisions. At run-time, if the resource allocated to a task is higher than the current resource usage, then idle resources are wasted. On the contrary, if the resource allocated to the task is much less than the actual task resource demand, the resource can become a performance bottleneck and slow down task execution. This motivates us to design a fine-grained, phase-level scheduling scheme that allocates resources according to the phase that each task is currently executing. By exploiting fine-grained phase-level resource characteristics, it is possible to better "bin-pack" tasks on machines to achieve higher resource utilization compared to task-level schedulers.

A key issue that must be addressed in phase-level scheduling is that once a task has completed a phase, the subsequent phase of the task may not be scheduled immediately if the machine does not have sufficient resources to run the subsequent phase. Thus, the execution of a phase may be "paused" in order to avoid resource contention, at the cost of delaying the completion of the task. However, we believe this trade-off is beneficial, as the improvement of the overall cluster utilization often implies the cluster is more productive in terms of executing tasks (i.e. by allowing more tasks to be executed simultaneously without causing resource contention). Thus the average job running time will be improved compared to task-level resource-aware schedulers.

Based on this motivation, we present PRISM, a fine-grained resource-aware scheduler that performs scheduling at phase-level. Unlike existing MapReduce schedulers that only allow job owners to specify resource requirements at task-level, PRISM allows the job owners to specify phase-level resource requirements. An overview of the PRISM architecture is shown in Fig. 4. PRISM consists of three main components: a phase-based scheduler at the master node, local node managers that coordinate phase transitions with the scheduler, and a job progress monitor to capture phase-level progress information. The phase-level scheduling mechanism used by PRISM is illustrated by Fig. 5. Similar to the current Hadoop implementation, each node manager periodically sends a heartbeat message to the scheduler. when a task needs to be scheduled, the scheduler replies to the heartbeat message with a task scheduling request (Step 1). The node manager then launches the task (Step 2). Each time a task finishes executing a particular phase (e.g. shuffle phase of the reduce task), the task asks the node manager for a permission to start the next phase (e.g. reduce phase of the task) (Step 3). The local node manager then forwards the permission request to the scheduler through the regular heartbeat message (Step 4). Given a job's phase-level resource requirements and its current progress information, the scheduler decides whether to start a new task, or allow a paused task to begin its next phase (e.g., the reduce phase), and then informs the node manager about the scheduling decision (Step 5). Finally, once the task

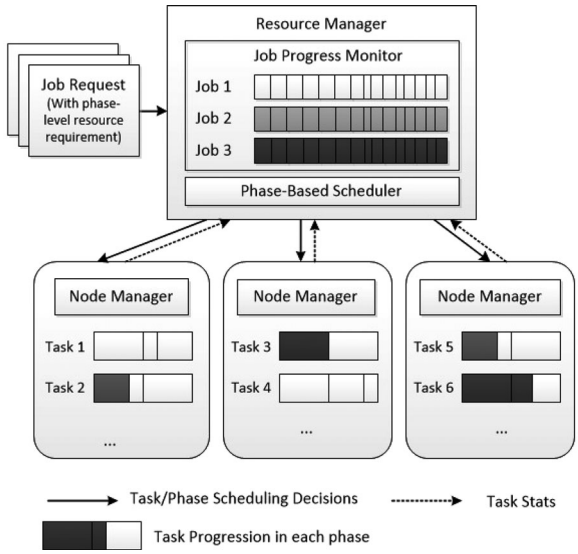


Fig. 4. System architecture.

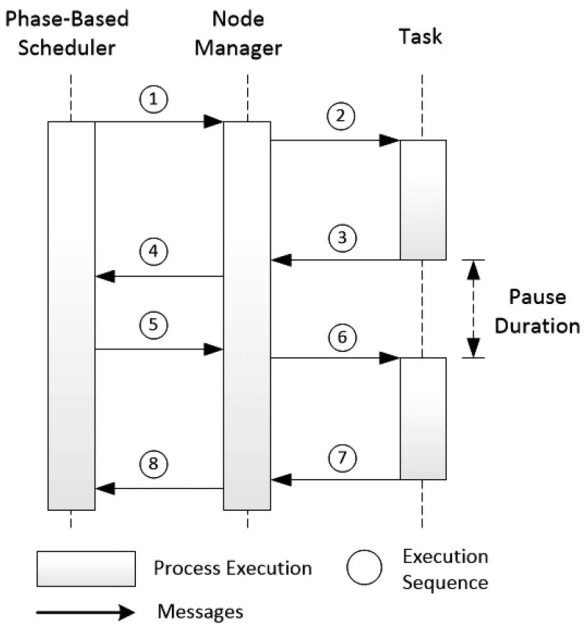


Fig. 5. Phase-level scheduling mechanism.

is allowed to execute the next phase, the node manager grants the permission to the task process (Step 6). Once the task is finished, the task status is received by the node manager (Step 7) and then forwarded to the scheduler (Step 8).

To perform phase-level scheduling, PRISM requires phase-level resource information for each job. In this work, we do not study the problem of job profiling as existing state-of-the-art job profilers, such as Starfish [12], can already provide accurate resource information that can be used by PRISM. While the accuracy of the profiles can affect the performance of PRISM, we believe PRISM is ideal for environment where jobs that are executed repeatedly with the same input size, which is common in many production clusters [17], [18]. In these environments, the accuracy of the job profiles can be improved over time. In the absence of phase-level resource information (i.e. a new job that has no profile), PRISM can fall back to use task-level resource information specified for Hadoop Yarn. In this case, each phase has the same resource requirement as the task itself.

Finally, even though the flexibility of phase-based scheduling should allow the scheduler to improve both resource utilization and job performance over existing MapReduce schedulers, realizing such a potential is still a challenging problem. This is because pausing the task execution at run-time may delay the completion of the current and subsequent tasks, which may increase the job completion time (these delayed tasks are commonly referred to as stragglers [10]). Thus, the scheduler must avoid introducing stragglers when switching between phases. In the following sections, we will describe how PRISM overcomes this challenge.

5 Scheduler Design

In this section, we describe in detail the design of PRISM's phase-based scheduling algorithm. We first describe the design rationale of the scheduling algorithm in Section 5.1, and then provide the details of our algorithm in Section 5.2.

5.1 Design Rationale

The responsibility of a MapReduce job scheduler is to assign tasks to machines with consideration for both efficiency and fairness [8], [14]. To achieve efficiency, job schedulers must maintain high resource utilization in the cluster. Job running time is another possible measure for efficiency, as a lower job running time implies that resources are more efficiently utilized for job execution. In contrast, fairness ensures that resources are fairly divided among jobs such that no job will experience starvation due to unfair resources allocation. However, simultaneously achieving both fairness and efficiency in the context of multi-resource scheduling has been shown to be challenging, as there is usually a trade-off between these objectives [14].

Fair scheduling algorithms such as Hadoop Fair Scheduler [3], Quincy [13] and dominant resource fairness (DRF) [11] generally run an iterative procedure by identifying users that experience the highest degree of unfairness (i.e. deficit) in each iteration, and schedule tasks that belong to those users to improve the overall fairness of the system. However, directly applying a fair scheduling algorithm for phase-level scheduling is insufficient. In particular, given a set of phases that can be scheduled on a machine, the scheduling algorithm must consider their inter-dependencies in addition to their resource requirements. For example, due to the sequential ordering of phases in a task, the scheduler needs to consider possible cascading delays when postponing the start of a phase. In many cases, such delays can also propagate to phases in other tasks, causing them to be delayed as well. For example, even though the execution of a shuffle phase of a reduce task can overlap with the execution of a merge phase of a map task, the shuffle phase cannot finish unless all merge phases of the map tasks have finished. Thus, when choosing between scheduling merge phases and shuffle phases, it is preferable to give sufficient resources to merge phases to allow them to finish faster, instead of allocating most of the resources to the shuffle phase and delay the completion of merge phases.

The above examples demonstrate the importance of achieving a fairness-performance trade-off for phase scheduling. In other words, it is necessary to provide fairness without significantly delaying the execution of each phase. While there are many possible alternatives to address this problem, in PRISM we have adopted a heuristic solution as follows: Given a set of phases that can be scheduled on a machine, the scheduler assigns a utility value to each phase which indicates the benefit of scheduling the phase. The scheduler will then schedule the phases in decreasing value of their utility. The utility value is phase-dependent, because phases have different dependencies. If a phase is map or shuffle, scheduling the phase implies scheduling a new map or reduce task. In this case, the utility of the phase is determined by the increase in parallelism from running an additional task. For other phases, the utility is determined by the urgency to complete the phase. A simple metric for measuring urgency is the number of seconds that a task has been paused due to phase-level scheduling. If the task has been paused for a long time, it becomes urgent to schedule its remaining phases in order to avoid creating a staggler.

5.2 Algorithm Description

We formally introduce our scheduling algorithm in this section. Upon receiving a heartbeat message from a node manager reporting resource availability on the node, the scheduler must select which phase should be scheduled on the node. Suppose there are J jobs in the system. Specifically, each job $j \in J$ consists of two types of tasks: map tasks M and reduce task R. Let $\tau(t) \in \{M, R\}$ denote the type of a task t. Given a phase i of a task t that can be scheduled on a machine n, we define the utility function of assigning a phase i to machine n as:

$$U(i,n) = U_{fairness}(i,n) + \alpha \cdot U_{perf}(i,n), \tag{1}$$

where $U_{fairness}$ and U_{perf} represent the utilities for improving fairness and job performance, respectively, and α is an adjustable weight factor. If we set α close to zero, then the algorithm would greedily schedule phases according to the improvement in fairness. Notice that considering job performance objectives will not severely hurt fairness. When a job is severely below its fair share, scheduling any phase with non-zero resource requirement will only improve its fairness.

Now we describe each term in Eq. (1). We define

$$U_{fairness}(i,n) = U^{before}_{fairness}(i,n) - U^{after}_{fairness}(i,n), \tag{2}$$

where $U^{before}_{fairness}(i,n)$ and $U^{after}_{fairness}(i,n)$ are the fairness measures of the job before and after scheduling i on n. The actual form of $U_{fairness}(i,n)$ is dependent on the fairness metric used. For example, if DRF is used in a homogenous MapReduce cluster, then the fairness utility $U^{before}_{fairness}$ can be computed as [11]:

$$U^{before}_{fairness}(i,n) = \max_{\{j,j'\} \in J} \left| \min_{r \in R} \left(\frac{c_{jr}}{C_r} \right) - \min_{r \in R} \left(\frac{c_{j'r}}{C_r} \right) \right|, \tag{3}$$

where c_{jr} denotes the task usage of resource r of job $j \in J$ before phase i is scheduled, and C_r denotes the capacity of resource r on a single machine. $U^{after}_{fairness}(i,n)$ can be computed in a similar way, except in this case c_{jr} represents the task usage after phase i is scheduled. Essentially, $U_{fairness}(i,n)$ measures the improvement in fairness due to the scheduling decision. $U_{perf}(i,n)$ is, on the other hand, more difficult to compute. As mentioned previously, if i is the leading phase (i.e. the first phase) of a task t, then $U_{perf}(i,n)$ measures the gain in parallelism in terms of the number of running map tasks (or reduce tasks). Otherwise, if i is a subsequent phase of task t, then $U_{perf}(i,n)$ measures the gain in shortening the running time of task t. Formally, we define

$$U_{perf}(i,n) = \begin{cases} U_{leading\_phase}(i,n) & i \text{ is a leading phase,} \\ U_{subsequent\_phase}(i,n) & \text{Otherwise.} \end{cases} \tag{4}$$

Even though PRISM does not dictate a particular function for computing the utility of a phase, in our implementation, we have chosen $U_{perf}(i,n)$ to be

$$U_{leading\_phase}(i,n) = \frac{N_{remaining}}{\max\{N_{current}, \epsilon\}} - \frac{N_{remaining}}{N_{current}+1}, \tag{5}$$

where $N_{remaining}$ denotes the number of remaining tasks of type $\tau(t)$ (i.e. the number of remaining tasks of the same type as t), and $N_{current}$ denotes the number of tasks of type $\tau(t)$ that are running. The variable ϵ is used to prevent dividing by 0. Intuitively, $U_{leading_phase}(i,n)$ measures the gain in parallelism if the number of running tasks is increased from $N_{current}$ to $N_{current}+1$.

On the other hand, let T^t_{wait} denote the number of seconds that task t has been paused due to phase-based scheduling. The utility for scheduling a non-leading phase i of task t is a function $p(\cdot)$ of T^t_{wait}:

$$U_{subsequent\_phase}(i,n) = p\big(T^t_{wait}\big). \tag{6}$$

There are many possible choices for $p(\cdot)$. For example, we can define $p(\cdot)$ as a linear function (i.e. $p(T^t_{wait}) = a \cdot T^t_{wait} + b$ for constants a and b), which would increase the utility of scheduling i to increase linearly with the number of seconds that the task has been paused. However, in our implementation, we have chosen $p(\cdot)$ to be a quadratic function of the form $p(T^t_{wait}) = a \cdot p(T^t_{wait})^2$, where $a = 0.1$. The intuition to using a quadratic function is to increase the urgency for scheduling i if i has been paused for a long time. However, PRISM can adopt any type of utility function $p(\cdot)$ as long as it is a monotonically increasing function.

Similar to Hadoop Fair Scheduler, PRISM supports locality-aware scheduling. As each job profile captures HDFS I/O usage, local disk (for writing intermediary output) and network usage for shuffle phase, at run-time the scheduler can use the data locality information to determine the disk and network usage of each task and allocate right amount of resources. To given higher preference to data-local tasks, in the scheduling algorithm the tasks of each job is sorted so that data local tasks appear before non-data local tasks. In each iteration each job provides one candidate phase to the scheduler, and phase of a new non-data local task is provided only if the there is no data local task that can be scheduled on the machine. This scheme is easy to implement and can work with locality-aware schemes such as delay scheduling [20].

Algorithm 1 summarizes the scheduling algorithm used by our phase-based scheduler. Specifically, upon receiving the status message from a node manager running on machine n, the algorithm computes the utilization u of the machine using job's phase-level resource requirement (Line 2). it then computes a set of candidate phases (i.e. the phases are schedulable on the machine) (Lines 4-9), and selects phases in an iterative manner. In each iteration, for each schedulable phase $i \in P(j)$ of each job j, it computes the utility function $U(i,n)$ according to equation (1) (Line 16). Then we select the phase with the highest utility for scheduling (Lines 22-23), and update the resource utilization of the machine (Line 25). Afterwards, the algorithm repeats by recomputing the utility of all the phases in the candidate set, and select the next best phase to schedule. The algorithm ends when the candidate set is empty, which means there is no suitable phase to be scheduled. The scalability of our algorithm can be achieved by only examining (1) the jobs that have tasks running on the machine, and (2) top k jobs with the lowest fairness

TABLE 1
Job Characteristics of the Gridmix Workload

Job Type	Num. of tasks		Running Time (s)		
	Map	Reduce	Map	Reduce	Job
MonsterQuery	16	30	12.38	17.7	156
WebDataScan	16	20	18.96	14.66	96
Combiner	8	5	99.54	16.68	162

TABLE 2
Job Characteristics of the PUMA Workload

Job Type	Num. of tasks		Running Time (s)		
	Map	Reduce	Map	Reduce	Job
Sort	16	8	10.87	24.67	97
Self-join	16	4	7.76	22.86	74
inverted-index	24	10	50.5	15.72	132
classification	24	10	4.73	10.15	70

measure. Assuming each machine can run at most N tasks, to further improve the scalability, the scheduler only needs to consider the top N schedulable phases of each job. The ranking of the phases is determined by data locality as mentioned previously. Thus, the overall running time of the algorithm is $O(N^2k)$. In our experiment, we set $k = 20$ and found the running time of the algorithm is usually less than 10 ms, which reasonable in most of the deployment scenarios.

Algorithm 1. Phase-Level Scheduling Algorithm

1: Upon receiving a status message from machine n:
2: Obtain the resource utilization of machine n
3: $PhaseSelected \leftarrow \{\emptyset\}$
4: $CandidatePhases \leftarrow \{\emptyset\}$
5: **repeat**
6: **for** each job $j \in jobsthathasksonn$ **do**
7: **for** each scheduable phase $i \in j$ **do**
8: $CandidatePhases \leftarrow CandidatePhases \cup \{i\}$
9: **end for**
10: **end for**
11: **for** each job $j \in$ top k jobs with highest deficit n **do**
12: **if** exist schedulable data local task **then**
13: $CandidatePhases \leftarrow CandidatePhases \cup \{$first phase of the local task $i\}$
14: **else**
15: $CandidatePhases \leftarrow CandidatePhases \cup \{$first phase of the non-local task $i\}$
16: **end if**
17: **end for**
18: **if** $CandidatePhases \neq \emptyset$ **then**
19: **for** $i \in CandidatePhases$ **do**
20: **if** i is not schedulable on n given current utilization **then**
21: $CandidatePhases \leftarrow CandidatePhases \backslash \{i\}$
22: continue;
23: **end if**
24: Compute the utility $U(i, n)$ as in equation (1)
25: **if** $U(i, n) \leq 0$ **then**
26: $CandidatePhases \leftarrow CandidatePhases \backslash \{i\}$
27: **end if**
28: **end for**
29: **if** $CandidatePhases \neq \emptyset$ **then**
30: $i \leftarrow$ task with highest $U(i, n)$ in the $CandidatePhases$
31: $PhaseSelected \leftarrow PhaseSelected \cup \{i\}$
32: $CandidatePhases \leftarrow CandidatePhases \backslash \{i\}$
33: Update the resource utilization of machine n
34: **end if**
35: **end if**
36: **until** $CandidatePhases == \emptyset$
37: return $PhaseSelected$

Lastly, it should be mentioned that PRISM can naturally tolerate task failures. As phase utilities are recomputed when the scheduler tries to assign new phases to a machine, the PRISM will still make consistent decisions in spite of task failures. PRISM also supports speculative re-execution. Speculative re-execution refers to launching multiple copies of a task, if the task is progressing slowly (i.e. below 25 percentile of all tasks) and is likely to delay the overall job completion [21]. In our implementation, even though the scheduling of a speculative task will not improve task-level parallelism, additional resources consumed will be used to improve fairness if other tasks have finished and the job is below its fair share.

6 Experiments

We have implemented PRISM in Hadoop 0.20.2. Implementing this architecture requires minimal change to the existing Hadoop architecture (around 1,000 lines of code). We deployed PRISM in a compute cluster which consists of 10 compute nodes. Each compute node has four-core 2.13 GHz Intel Xeon E5606 processors, 8 GB RAM, 100 GB of local high speed hard drive, and runs 64-bit Ubuntu OS. The network interface card (NIC) installed on each node is capable of handling up to 1Gb/s of network traffic. Each node is connected to a top-of-rack switch and can communicate with others via a $1\,\mathrm{Gb/s}$ link.

We have chosen two benchmarks to evaluate the performance of PRISM: Gridmix 2 and PUMA. Gridmix 2 [5] a standard benchmark included in the Hadoop distribution. For Gridmix 2 we have chosen three jobs for performance evaluation: `MonsterQuery (MQ)`, `WebDataScan (WDS)` and `Combiner (CM)`. Similarly, PUMA [6] is a MapReduce benchmark developed at Purdue University. We have selected four jobs for performance evaluation: `sort (SRT)`, `self-join (SJ)`, `inverted-index (II)` and `classification (CL)`. We chose these jobs because they contain a variety of resource usage characteristics. For example, `sort` and `MonsterQuery` are I/O intensive jobs, whereas `Combiner` and `self-join` are more CPU intensive. Table 1 and 2 summarize the characteristics of each job used in our evaluation. A mixture of jobs with different resource requirements allows us to better evaluate the performance of PRISM.

To evaluate the benefit brought by phase-level scheduling, it is necessary to compare PRISM to existing task-level resource-aware schedulers. In our experiments, we have chosen Hadoop Yarn 2.0.4 as a competitive task-level resource-aware scheduler. Hadoop Yarn 2.0.4 is a recent version of Hadoop NextGen that allows the users to specify both CPU (i.e. number of virtual cores (vCores)) and memory (i.e. GB of RAM) requirements of each task. Ideally, we

Fig. 6. # of Slots versus Map running time.

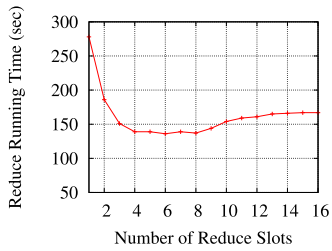


Fig. 7. # of Slots versus Job running time.

would like to compare PRISM with Hadoop Yarn running a fair scheduler. However, Hadoop Yarn is yet to support fair scheduling with consideration to resource requirements. Therefore, in our experiments we compare PRISM with Hadoop Yarn Capacity scheduler, as the Capacity scheduler takes resource requirements into consideration when making scheduling decisions. Lastly, to evaluate the fairness of our scheduler, in our implementation, we adopt the same fairness metric as in the Hadoop fair scheduler 0.20.2, and use Hadoop 0.20.2 as a baseline for comparing both fairness and scheduler performance.

6.1 Capturing Job Performance Requirements

Even though job profiling is not the main focus of this work, for analysis purposes, we have implemented a simple job profiler that captures the CPU, memory and I/O usage of both tasks and compute nodes. Writing our own profiler allows us to better analyze the fine-grained resource characteristics of individual phases. In our implementation, we monitor the execution of each task and record the start and end time of every phase in the task log file. As for monitoring run-time resource usage, we rely on linux `top` command to record CPU and memory usage once per second. Network I/O is more difficult to profile. In our current implementation, we modified the Hadoop source code to print the values of I/O counters. The actual disk and network I/O usage over-time can be obtained from Linux utilities such as `iotop` and `nethogs`.

We rely on Hadoop Yarn's capability of providing resource isolation between tasks to obtain accurate phase-level resource usage information. Specifically, we run each job used in the experiment in Hadoop Yarn and collect the run-time resource usage of each phase. However, in order to run Hadoop Yarn, we must first specify the task-level resource requirement, which is not available (and not reported in the literature) for the jobs we consider. Instead of using arbitrary values for task container size, we determine the task-level resource requirements using an approach similar to the one in [15]: We run each job in Hadoop 0.20.2 with different number of slots allocated to map and reduce tasks. Specifically, we first vary the number of map slots to find an optimal number that minimizes the map completion time. Using this number, we then vary the number of reduce slots to find an optimal number of reduce slots that minimizes the overall job completion time. Then we compute the task size using the optimal number of map and reduce slots per machine. For instance, Figs. 6 and 7 shows the result for adjusting the number of maps slots and reduces for the `sort` job, respectively. Both figures show that the job running time has a non-linear relationship with the number of slots used. When the number of slots is small (e.g. two slots) the job running time becomes long due to the low degree of task-level parallelism imposed by the slot allocation. On the other hand, when the number of slots is large (e.g. 12 slots) the running time again becomes high due to multiple tasks competing for bottleneck resources. For the `sort` job, we found setting the number of map slots and reduce slots to 8 and 6 respectively achieves the optimal running time. The profile for the `sort` job is shown in Fig. 8. The same process is repeated to create the profiles for all other jobs. Notice we adopt this approach mainly because we do not want to set task size used by Yarn to arbitrary values. In practice, the task size is specified by the user, and we can simply just collect the phase-level information with low overhead.

In our experiments, to compare the performance of all three schedulers (i.e., PRISM, Fair Scheduler and Yarn), we set the task container size used by Yarn according to the optimal number of slots found by Hadoop 0.20.2. The default configuration of Yarn specifies 16 virtual cores per machine. Yarn also requires that the number of vCores per task must take integer values in the job request. In our experiments, we have found the default configuration of Yarn produces lower performance compared to both the Fair Scheduler and PRISM. This is due to the large rounding errors for converting the number of vCores to integer values. Therefore, we modified the default configuration so that each machine provides 128 vCores. This significantly reduces rounding errors, allowing Yarn to produce comparable performance against both the fair scheduler and PRISM.

6.2 Evaluation Using Individual Jobs

In our first experiment, our goal is to demonstrate the benefit of phase-level scheduling. For this purpose, we run a single `sort` job in a small cluster consisting of only three nodes using Fair Scheduler, Yarn and PRISM.[3] The input size is set to 5 GB. The number of map and reduce slots used by Fair Scheduler is set to 8 and 6 as discussed in previous section.

The experiment results for Hadoop fair scheduler, Yarn and PRISM are shown in Figs. 10, 11, and 12, respectively. In particular, the fair scheduler is able to complete the job execution in 149 seconds, whereas Yarn finishes the job in 152 seconds. In contrast, PRISM achieves the same in just 125 seconds (as shown in Fig. 12a), resulting in a 19 percent reduction in job running time. To understand the reason

3. We choose three nodes in this experiment mainly to allow us to visualize the execution of the job, as well as to demonstrate the scenarios where PRISM outperforms the fair scheduler

Job: sorter
Input size: 5GB, Map Count: 40, Reduce Count: 56
Map stage completion: 63s, Reduce stage completion: 147s

Phase	Map	Merge	Shuffle	Sort	Reduce
t_i (s)	7.43	1.25	9.07	0.64	9.69
$CPU(\%)$	17.42	14.35	21.58	7.5	8.21
Mem (%)	1.35	1.40	2.11	2.37	2.33
LFS(MB/s)	3.98	34.31	5.71	11.17	5.29
HDFS(MB/s)	7.17	0	0	0	5.30
Shuffle(MB/s)	0	0	7.16	0	0

Fig. 8. An example Job Profile: Sort Job.

behind the performance gain, we first plotted the CPU/Memory usage as well as disk/network I/O usage in Figs. 10b and 10c for Fair Scheduler, in Figs. 11b and 11c for Yarn, and in Figs. 12b and 12c for PRISM. We found Yarn achieves highest utilization while performing slightly worse than the fair scheduler. The main reason is that Yarn has an additional scheduling overhead. Specifically, in order to run a new MapReduce job, scheduler need to run a job controller called Application Master [7], which will be responsible for monitoring and managing the job execution. This Application Master also consumes cluster resources at runtime, which reduce the resource capacity available for task scheduling. In contrast, PRISM delivers higher utilization for all resources. The CPU utilization of PRISM is always better than that of Fair scheduler except near the end of the execution.

We also plotted Figs. 10d, 11d and 12d to show the number of phases scheduled over time by each scheduler. For clarity of presentation, we only show the plot for the three major phases: map, shuffle and reduce. Both PRISM and Yarn are able to achieve higher degree of parallelism during the map stage (seven and six map tasks running concurrently on average) than the Fair Scheduler. During the reduce stage, as shuffle phases consumes more resources than reduce phases, PRISM recognizes the potential resource bottleneck, and thus delays the start of the reduce phases, allowing more shuffle phases to be scheduled. This makes the shuffle phases to run faster than the Fair scheduler and Yarn (81 seconds for PRISM, 86 seconds for Yarn and 89 seconds for the Fair Scheduler). As reduce phases consume less resources, they can be scheduled in large quantity without causing resource contention. Given the flexibility to separate shuffle phases from reduce phases, PRISM is able to find better schedule for both shuffle and reduce phases without cause resource contention, resulting in better in job running time. This confirms our intuition in Section 4, which states that PRISM relies on improving resource utilization to improve job completion times, at the cost of slightly delaying the completion of individual tasks. As resources are better utilized for task execution without causing resource contention, making trade-off can lead to higher job performance than existing Hadoop resource-aware schedulers.

After demonstrating the behavior of PRISM using Figs. 10, 11 and 12, we also evaluated the performance using data set ranging from 25 to 100 GB. The results are shown in Fig. 13. Even though PRISM achieves slightly worse data locality compared to Fair scheduler and Yarn, it still outperformed both Fair scheduler and Yarn. The explanation is that PRISM tries to find best opportunities for bin-packing, and as a result, it may schedule tasks on machines that have low locality. However, as mentioned in Section 5.2, because our algorithm prioritize data-local tasks when making scheduling decisions, the decrease in data locality is small. We also varied the number of nodes in the cluster and run sort with 100 GB input. the results are shown in Fig. 14. Despite having slightly worse data locality, PRISM delivers better performance than Fair scheduler and Yarn.

We also performed the same experiment for the remaining jobs in the Gridmix 2 and the PUMA benchmark. The results are shown in Fig. 9. PRISM outperforms both the fair scheduler and Yarn for all the jobs. The reduction in job running time ranges between 5-38 percent. Furthermore, we have found that PRISM generally achieves higher reduction in job running time for reduce intensive jobs (e.g. `sort` and `self-join`, where reducers consume more resources than mappers). The reason is that for reduce-intensive jobs, both shuffle and reduce phases take longer time to run and often have drastically different resource consumption characteristics. As a result, PRISM is able to find better schedules compared to both Yarn and fair scheduler, and therefore the gain becomes higher. While one may think that the performance improvement is due not only to the scheduling but also due to how data is distributed across tasks, we argue this is not case, as we make no effort in optimizing data placement and the assignment of data to tasks. Furthermore, We tested our algorithm with different input sets and the performance gain remain the same. This confirms the performance gain achieved by PRISM is independent of data distribution.

6.3 Evaluation Using Benchmarks

We now present our evaluation result using both PUMA and Gridmix 2 benchmarks. In the PUMA benchmark, we vary the number of jobs between 50 to 200 to create batch

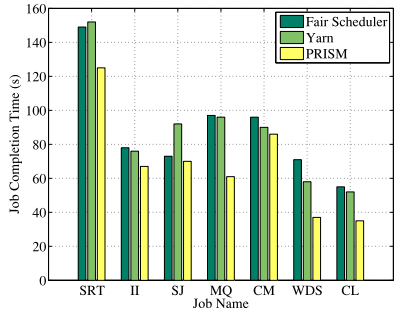
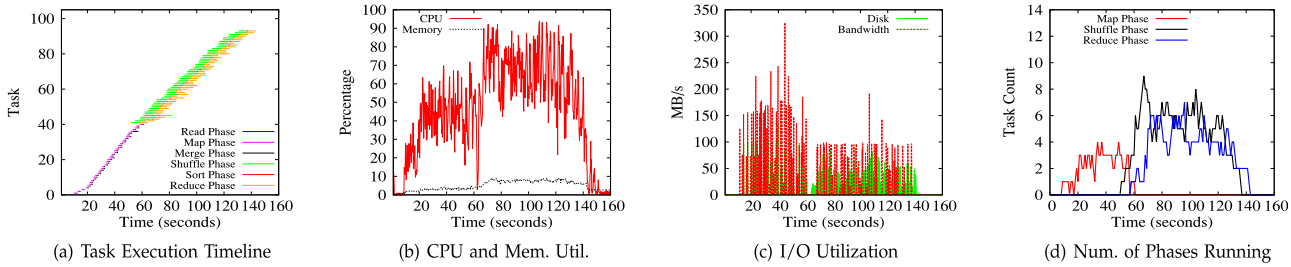
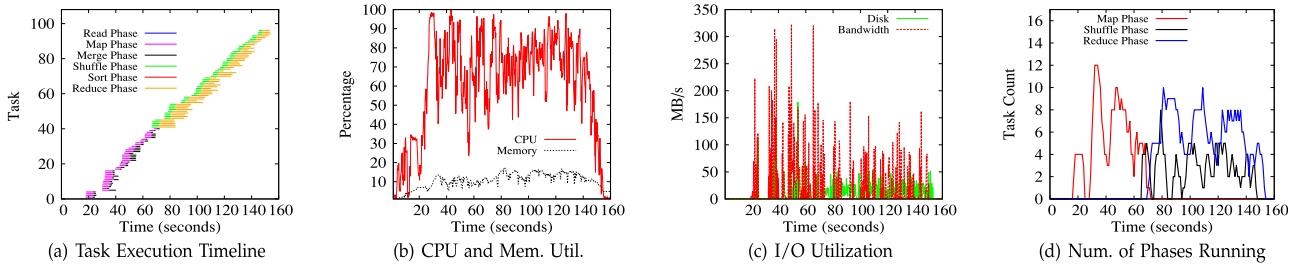


Fig. 9. Running time of each job.



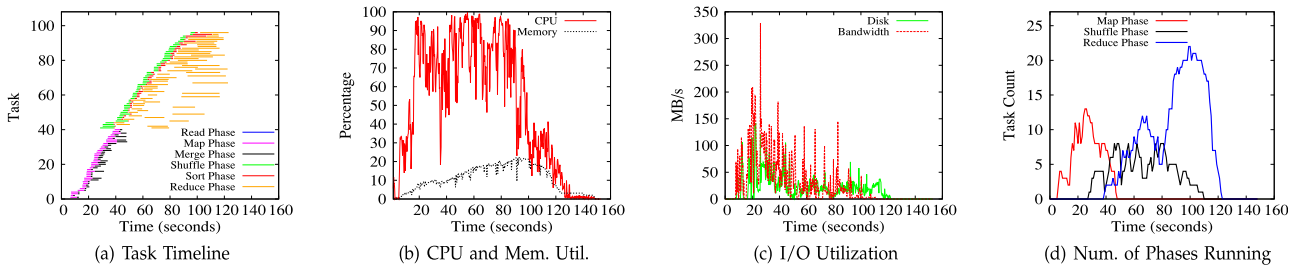
(a) Task Execution Timeline (b) CPU and Mem. Util. (c) I/O Utilization (d) Num. of Phases Running

Fig. 10. Sorting 5 GB data with fair-scheduler.



(a) Task Execution Timeline (b) CPU and Mem. Util. (c) I/O Utilization (d) Num. of Phases Running

Fig. 11. Sorting 5 GB data with Yarn.



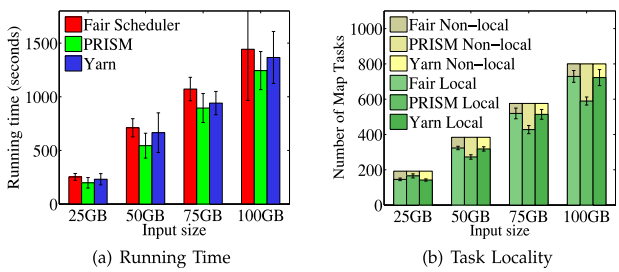
(a) Task Timeline (b) CPU and Mem. Util. (c) I/O Utilization (d) Num. of Phases Running

Fig. 12. Sorting 5 GB data with PRISM.

workload of different size, and run each of the batch workload three times using Fair scheduler, Yarn and PRISM. To provide an accurate evaluate the performance gain, in our experiments, all the jobs in the batch are simultaneously submitted to the job tracker and executed concurrently in the cluster. The results for job completion time is shown in Fig. 15a. It can be seen that PRISM outperforms both Fair scheduler and Yarn in all scenarios. Furthermore, Yarn generally outperforms the Fair scheduler for large workloads, because it is more resource-aware. The locality of tasks are shown in Fig. 16a. Once again, PRISM achieves lower task locality, while deliver better performance than Fair scheduler and Yarn. Figs. 17a, 18a and 19a shows the resource utilization of the cluster during the execution of each batch for each scheduler respectively. It can be seen from the diagrams that PRISM generally provider higher resource utilization than the Fair Scheduler, and One interesting observation is that PRISM achieves higher I/O throughput than Hadoop Yarn, but slightly lower CPUutilization. The reason is that CPU is the performance bottleneck of the workload. As PRISM tries to mitigate CPU contention while improving utilization of idle resources, it achieves higher I/O throughput than Yarn. On average, PRISM is able to reduce job running time by up to 24 percent. The benefit of PRISM mainly comes from the fact that PRISM achieves higher degree of parallelism through better scheduling of phases, resulting in shorter job running time.

Similarly, we vary the number of jobs in the Gridmix 2 benchmark from 25 to 100 to create multiple batches of Gridmix 2 workload. Each batch is then executed three times using all three schedulers. The results for average job running time, data locality and resource utilizations are shown in Figs. 15b, 16b, 17b, 18b and 19b, respectively. The results are similar to that of the PUMA workload. These results suggest that PRISM is able to achieve shorter job running time while maintaining high resource utilization for large workloads containing a mixture of jobs, which are common in production clusters.

So far we have only analyzed the aggregate workload running time and resource utilization. However, these objectives should not be achieved at the cost of introducing poor job fairness. Therefore, we have also measured the *application normalized performance (ANP)* and the *unfairness*



(a) Running Time (b) Task Locality

Fig. 13. Running sort with different input size.

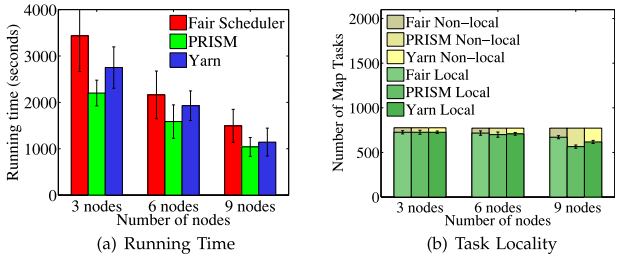


Fig. 14. Running sort with different clusters.

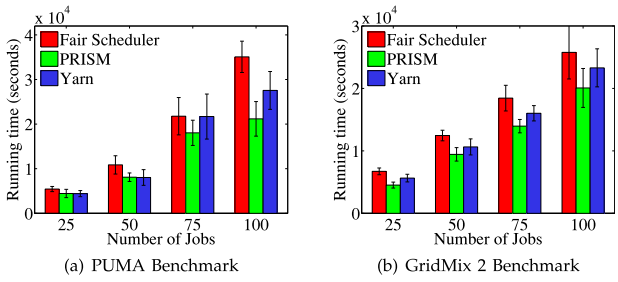


Fig. 15. Benchmark running time.

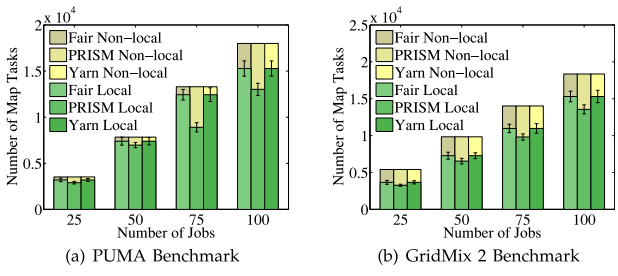


Fig. 16. Benchmark locality.

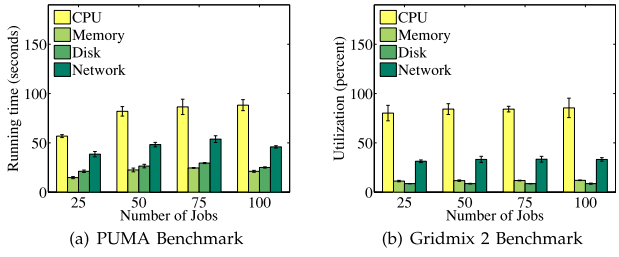


Fig. 17. Utilization using fair scheduler.

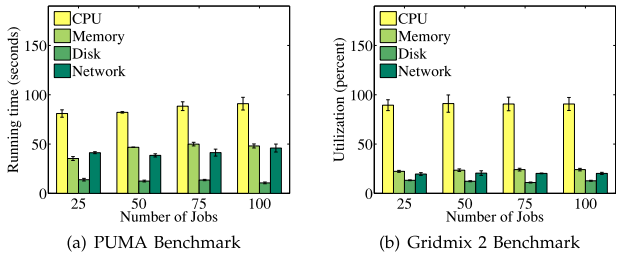


Fig. 18. Utilization using yarn.

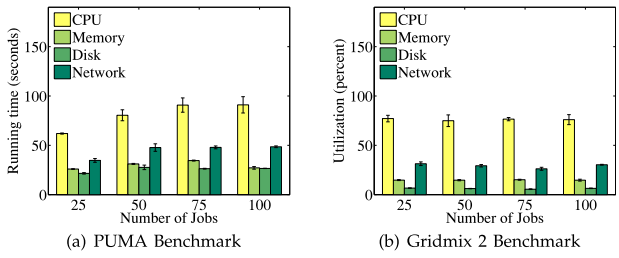


Fig. 19. Utilization Using PRISM.

as introduced by Isard et al. in Quincy [13]. The ANP of a job is the ratio between the ideal job running time (when the job is given sufficient capacity to run at full speed) to actual job running time. Thus, the higher the ANP value is, the better the scheduler performs in terms of improving job running time. The unfairness, on the other hand, is the coefficient of variation (CV) of the ANP values across all jobs in the batch. The intuition is that a fair scheduler should ensure all jobs experience similar speed up rate regardless of the current utilization of the cluster. Therefore, a small CV of ANP values indicates a high level of fairness achieved by the scheduler. The results of ANP and unfairness for both PUMA and Gridmix workload are shown in Figs. 20a, 20b, 20c and 20d respectively. Specifically, Figs. 20a and 20b show that PRISM is able to achieve high ANP values compared to both Fair Scheduler and Yarn. However, it delivers slightly higher unfairness than the Fair Scheduler, as shown in Figs. 20c and 20d. We believe this is due to the fact that PRISM tries to find a balance between performance and resource-awareness. Thus due to resource constraints, it is not possible to achieve ideal fairness values. However, as the difference is relatively small between these two schedulers, we believe sacrificing a small amount of fairness for the sake of improving resource utilization and job running time is beneficial to the overall performance of the cluster. Finally, we found that Yarn achieves the worst unfairness values. This is because it uses the capacity scheduler, which does not take fairness into consideration when making scheduling decisions.

7 Related Work

The original Hadoop MapReduce implements a slot-based resource allocation scheme, which does not take run-time task resource consumption into consideration. As a result, several recent works reported the inefficiency introduced due to such simple design, and proposed solutions. For instance, Polo et al. proposed RAS [15], an adaptive resource-aware scheduler that uses job specific slots for scheduling. However, RAS still performs scheduling at task-level, and does not consider the task resource usage variations at run time. Subsequently, Hadoop Yarn [7] represents a major endeavor towards resource-aware scheduling in MapReduce clusters. It offers the ability to specify the size of each task container in terms of requirements for each type of resources. In this context, A key challenge is to define the notion of fairness when multiple resource types are considered. Ghodsi et al. proposed dominant resource fairness as a measure of fairness in the presence of multiple resource types, and provided a simple scheduling algorithm for achieving near-optimal DRF. However, the DRF scheduling algorithm still focuses on task-level scheduling, and does not consider change in resource consumption within individual tasks. Their subsequent model, namely

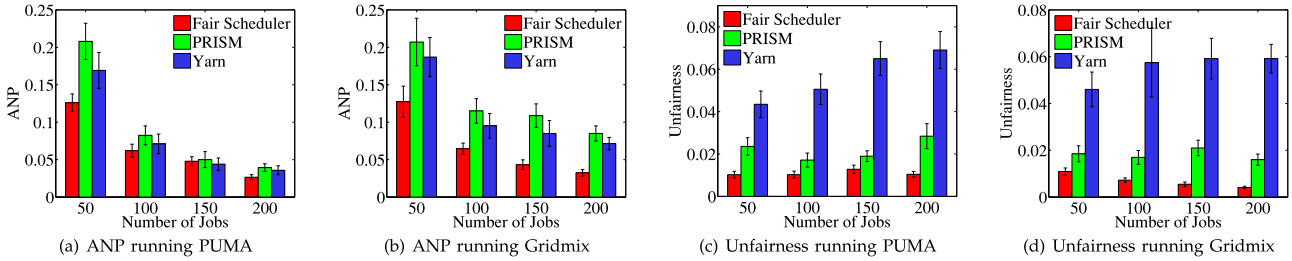


Fig. 20. Fairness result using benchmarks.

dominant resource fair queueing (DRFQ), aims at achieving DRF for packet scheduling over time. However, DRFQ algorithm is mainly designed for packet scheduling, which is different from the task-level "bin-packing" type of scheduling model we consider in this paper. Thus it cannot be directly applied to MapReduce scheduling.

Using profiles to improve MapReduce job performance has received considerable attention in recent years [12]. For instance, Verma et al. [17] developed a framework that profiles task running times and use the job profiles to achieve deadline-ware scheduling in MapReduce clusters. Herodotou et al. recently developed Starfish [12], a job profiler that collects fine-grained task usage characteristics that can be used for fine-tuning job configuration parameters. However, the goal of profiling in these studies is to optimize job parameters, rather than optimizing job schedules.

Another related research direction is MapReduce pipelining. In particular, MapReduce Online [9] is a framework for stream-based processing of MapReduce jobs. it allows partial outputs of each phase to be sent directly to the subsequent phase, thus enables overlaps execution of phases. ThemsisMR [16] is another scheme that modifies MapReduce phases to improve I/O efficiency. However, both of these solutions does not deal with scheduling. Furthermore, they are not resource-aware. While introducing resource-awareness in MapReduce Online is another interesting alternative, the scheduling model for MapReduce online is much different from the current MapReduce. It will require further investigation to identify scheduling issues for MapReduce online.

8 Conclusion

MapReduce is a popular programming model for data intensive computing. However, despite recent efforts toward designing resource-efficient MapReduce schedulers, existing work mainly focuses on designing task-level schedulers, and is oblivious to the fact that the execution of each task can be divided into phases with drastically different resource consumption characteristics. To address this limitation, we introduce PRISM, a fine-grained resource-aware scheduler that coordinates task execution at the level of phases. We first demonstrate how task run-time usage can vary significantly over time for a variety of MapReduce jobs. We then present a phase-level job scheduling algorithm that improves job execution without introducing stragglers. In a 16-node Hadoop cluster running standard benchmarks, we demonstrated that PRISM offers high resource utilization and provides $1.3\times$ improvement in job running time compared to the current Hadoop schedulers.

Lastly, we believe there are many interesting avenues for future exploration. In particular, we would like to study the problem of meeting job deadlines under phase-level scheduling. Also, in this paper we assume all machines have identical hardware and resource capacity. It is interesting to study the profiling and scheduling problem for machines with heterogenous performance characteristics. Finally, improving the scalability of PRISM using distributed schedulers is also an interesting direction for future research.

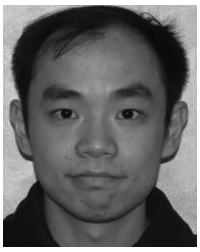
Acknowledgments

This work was completed as part of the Smart Applications on Virtual Infrastructure (SAVI) project funded under the National Sciences and Engineering Research Council of Canada (NSERC) Strategic Networks grant number NETGP394424-10.

References

[1] Hadoop MapReduce distribution [Online]. Available: http://hadoop.apache.org, 2015.
[2] Hadoop Capacity Scheduler [Online]. Available: http://hadoop.apache.org/docs/stable/capacity_scheduler.html/, 2015.
[3] Hadoop Fair Scheduler [Online]. Available: http://hadoop.apache.org/docs/r0.20.2/fair_scheduler.html, 2015.
[4] Hadoop Distributed File System [Online]. Available: hadoop.apache.org/docs/hdfs/current/, 2015.
[5] GridMix benchmark for Hadoop clusters [Online]. Available: http://hadoop.apache.org/docs/mapreduce/current/gridmix.html, 2015.
[6] PUMA benchmarks [Online]. Available: http://web.ics.purdue.edu/fahmad/benchmarks/datasets.htm, 2015.
[7] The Next Generation of Apache Hadoop MapReduce [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2015.
[8] R. Boutaba, L. Cheng, and Q. Zhang, "On cloud computational models and the heterogeneity challenge," *J. Internet Serv. Appl.*, vol. 3, no. 1, pp. 1–10, 2012.
[9] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2010, p. 21.
[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[12] 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. Conf. Innovative Data Syst. Res.*, 2011, pp. 261–272.
[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, and K. Talwar, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS Symp. Oper. Syst. Principles*, 2009, pp. 261–276.
[14] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. "Multi-resource allocation: Flexible tradeoffs in a unifying framework," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2012, pp. 1206–1214.

[15] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for MapReduce clusters," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2011, pp. 187–207.
[16] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat, "ThemisMR: An I/O-Efficient MapReduce," in *Proc. ACM Symp. Cloud Comput.*, 2012, p. 13.
[17] A. Verma, L. Cherkasova, and R. Campbell, "Resource provisioning framework for MapReduce jobs with performance goals," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2011, pp. 165–186.
[18] D. Xie, N. Ding, Y. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *Proc. ACM SIGCOMM*, 2012, pp. 199–210.
[19] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. USENIX Symp. Oper. Syst. Des. Implementation*, 2008, pp. 1–14.
[20] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[21] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. USENIX Symp. Oper. Syst. Des. Implementation*, 2008, vol. 8, pp. 29–42.



Qi Zhang received the BASc, MSc, and PhD degrees from the University of Ottawa, Canada, Queen's University, Canada, and the University of Waterloo, Canada, respectively. He is currently working toward the postdoctoral fellowship at the University of Toronto, Canada. His current research focuses on resource management for cloud computing systems. He is also interested in related areas including big-data analytics, software-defined networking, network virtualization, and management. He is a student member of the IEEE.

Mohamed Faten Zhani received the engineering and MS degrees from the National School of Computer Science, Tunisia, in 2003 and 2005, respectively. He received the PhD degree in computer science from the University of Quebec in Montreal, Canada, in 2011. Since then, he has been a postdoctoral research fellow at the University of Waterloo. His research interests include cloud computing, virtualization, Big data, and software-defined networking. He is a member of the IEEE.

Yuke Yang received the BEng degree from Xi'an Jiaotong University, China, in 2011 and the MMath degree from the University of Waterloo, Canada, in 2013, respectively. Her research interests include big data systems and analytics, cloud computing, database systems, and machine learning.

Raouf Boutaba received the MSc and PhD degrees in computer science from the University Pierre and Marie Curie, Paris, France, in 1990 and 1994, respectively. He is currently a professor of computer science with the University of Waterloo, Waterloo, ON, Canada. His research interests include control and management of networks and distributed systems. He is a fellow of the IEEE and the Engineering Institute of Canada.

Bernard Wong received the BASc degree in computer engineering from the University of Waterloo, and the MS and PhD degrees in computer science from Cornell University. He is currently an assistant professor in the School of Computer Science at the University of Waterloo. His research interests span distributed systems and networking, with particular emphasis on problems involving decentralized services, self-organizing networks, and distributed storage systems.