The *Packing Server* for Real-Time Scheduling of MapReduce Workflows

Shen Li, Shaohan Hu, Tarek Abdelzaher
University of Illinois at Urbana-Champaign
$\{$*shenli3, shu17, zaher*$\}$*@illinois.edu*

***Abstract*—This paper develops new schedulability bounds for a simplified MapReduce workflow model. MapReduce is a distributed computing paradigm, deployed in industry for over a decade. Different from conventional multiprocessor platforms, MapReduce deployments usually span thousands of machines, and a MapReduce job may contain as many as tens of thousands of parallel segments. State-of-the-art MapReduce workflow schedulers operate in a best-effort fashion, but the need for real-time operation has grown with the emergence of real-time analytic applications. MapReduce workflow details can be captured by the *generalized parallel task model* from recent real-time literature. Under this model, the best-known result guarantees schedulability if the task set utilization stays below 50% of total capacity, and the *deadline to critical path length* ratio, which we call the stretch φ, surpasses 2. This paper improves this bound further by introducing a hierarchical scheduling scheme based on the novel notion of a *Packing Server*, inspired by servers for aperiodic tasks. The Packing Server consists of multiple periodically replenished budgets that can execute in parallel and that appear as independent tasks to the underlying scheduler. Hence, the original problem of scheduling MapReduce workflows reduces to that of scheduling independent tasks. We prove that the utilization bound for schedulability of MapReduce workflows is $U_B \cdot \frac{\varphi-\beta}{\varphi}$, where U_B is the utilization bound of the underlying independent task scheduling policy, and β is a tunable parameter that controls the maximum individual budget utilization. By leveraging past schedulability results for independent tasks on multiprocessors, we improve schedulable utilization of DAG workflows above 50% of total capacity, when the number of processors is large and the largest server budget is (sufficiently) smaller than its deadline. This surpasses the best known bounds for the generalized parallel task model. Our evaluation using a Yahoo! MapReduce trace as well as a physical cluster of 46 machines confirms the validity of the new utilization bound for MapReduce workflows.**

I. Introduction

The past decade has seen MapReduce [1–5] become the dominant distributed computing paradigm in industry. The importance of meeting deadlines of MapReduce workflows has grown in recent years as well, driven by the advent of real-time analytics [6–16]. The success of the MapReduce community in addressing real-time constraints, however, remains limited due to the inherent difficulty of the workflow scheduling problem on parallel resources. On the other hand, in real-time scheduling literature, recent results on schedulability of generalized parallel tasks do not offer high platform utilization. The prospect of improving schedulability bounds of generalized parallel tasks on multiprocessors in the subcases relevant to MapReduce workflows motivates the work reported in this paper.

The MapReduce distributed computing paradigm splits source data into independent chunks and processes them using two phases: the *map* phase applies the map function onto each chunk, generating intermediate key-value pairs, while the *reduce* phase aggregates and summarizes those key-value pairs based on their keys. The reduce phase cannot start until the map phase finishes, and both phases can be parallelized to run on a large number of slots, where a slot is a resource unit in MapReduce clusters. MapReduce deployments usually span thousands of machines, connected by a high-speed and high-bandwidth intranet. Assuming a bounded network delay (that we can subtract from the end-to-end deadline), the platform acts as a very large multiprocessor. One MapReduce job may contain tens of thousands of parallel segments [17, 18]. Due to input/output dependencies that are often required to carry out complex algorithms, MapReduce jobs usually form Directed Acyclic Graphs (DAG), called MapReduce workflows.

Many data processing algorithms used in the context of MapReduce workflows are bulk algorithms (as opposed to incremental-update algorithms). They require a bulk of data to be present at once. This leads to a periodic invocation model, where a volume of data is first collected within the current time-slice, and then the MapReduce workflow is invoked. Being able to meet workflow deadlines is often of crucial importance to businesses, because applications supported by production MapReduce workflows, such as advertisement placement optimizations, user graph storage partitions, and personalized content recommendations, usually directly affect site performance and company revenue.

The state-of-the-art MapReduce workflow schedulers, such as Oozie [19] and WOHA [18], operate in a best-effort fashion, offering little guarantees on workflow completion times. With the surge of interest in real-time workflow execution, recent work addressed scheduling extensions that offer resource partitioning [20, 21], reduce preemption cost to support prioritization [22], or take deadlines as input to the scheduler [11, 12]. However, these attempts fall short of offering timing guarantees. Administrators cannot tell, quantitatively, what maximum utilization their MapReduce clusters can bear before jobs start lagging too far behind. This calls for an analytically well-motivated scheduling and admission control policy, which is the topic of this paper.

This paper makes two contributions. From the perspective of MapReduce applications, we offer an analytic result and a run-time mechanism to guarantee schedulability of MapReduce workflows as long as a schedulability bound is met.

From the perspective of real-time foundations, we improve the best known bound for schedulability of the generalized parallel task model in a special subcase of relevance to MapReduce applications.

The contributions rest on the novel idea of the *Packing Server*. It constitutes a run-time mechanism that makes concurrent precedence-constrained workflows look like independent periodic tasks to the underlying MapReduce scheduler. We then derive a *conversion factor* that expresses a bound for schedulability of MapReduce workflows as a function of the underlying utilization bound for schedulability of independent periodic tasks.

Details of a MapReduce workflow can be captured by a generalized parallel task model [23]. Among the metrics used in existing work, utilization bounds and capacity augmentation bounds give rise to efficient admission control policies, as they only require simple information about the task set and the platform. The best-known result for schedulability of workflow tasks, that is amenable to a simple admission control policy, is a capacity augmentation bound of 2 for implicit-deadline task models using a federated scheduling strategy [24]. In order to guarantee schedulability, this bound requires the task set utilization to be less than 50% of total capacity, and the *deadline to critical path length* ratio, which we call the *stretch* (φ), to be larger than 2. This constraint (only 50% of capacity) may be too restrictive for some systems.

Fortunately, more advances have been made on schedulability of independent task sets, achieving much higher utilization bounds, especially when the number of processors is large and the individual units of work are small. For example, López *et al.* [25] proved that first-fit partitioned EDF (EDF-FF) can schedule any system of independent periodic tasks on m processors, given that the total capacity stays below $U_B = \frac{m\beta+1}{\beta+1}$, where β is inverse of the maximum individual task utilization (*i.e.*, $\beta = \frac{1}{u_{\max}}$). As another example, the global EDF scheduling [26] guarantees to meet all deadlines if the total task set utilization is less than $U_B = m\left(1 - \frac{1}{\beta}\right) + \frac{1}{\beta}$. Note that, for $\beta = \frac{1}{u_{\max}} \geq 1$, the total schedulable utilization is larger than $m/2$ (i.e., larger than 50% of total capacity). In fact, the bound approaches 100% of capacity as β increases. These observations suggest that task set abstractions or transformations that make workflows look like independent tasks may improve schedulability.

The Packing Server mechanism, presented in this paper, is inspired by the above observation. Each Packing server consists of a number of budgets dedicated to a given MapReduce workflow. The MapReduce system schedules these budgets as independent tasks. When invoked, the budget runs MapReduce segments in a manner that respects workflow precedence constraints. Hence, the original problem of analyzing schedulability of MapReduce workflows on multiprocessors is translated into the well-known problem of analyzing schedulability of independent tasks. The utilization bounds for the latter are well known both for EDF and fixed priority scheduling, as well as both for partitioned and global schedulers. We prove a *conversion factor* between the utilization bound for schedulability of MapReduce workflows achieved by our scheme, and the utilization bound of the underlying scheduler for independent tasks. Namely, the MapReduce workflow is schedulable if utilization is below $U_B \cdot \frac{\varphi-\beta}{\varphi}$. In the following, we shall use the convention of expressing utilization as *percentage of total cluster capacity*. Hence, for example, for a cluster of m machines, we shall say 50% when we mean $m/2$, and will refer to it as the *cluster utilization*.

Since the deadlines for MapReduce workflows are typically large (e.g., hours) and the clusters are big, it is common that MapReduce workflows enjoy a large stretch, φ, leading to a high utilization bound. Hence, we improve the best known results for scheduling MapReduce DAGs on multiprocessors. The paper describes how to size Packing servers, derives the schedulability bounds attained, and presents the policies used inside a server in handling MapReduce workflows. Evaluation results confirm the improved schedulability.

The remainder of this paper is organized as follows. Section II briefly introduces the MapReduce job model. Section III and IV develop the conversion factor for individual MapReduce jobs and workflows, respectively. We describe the application-level scheduling algorithm for packing work inside servers in Section V. Section VI presents evaluation results. We survey the related work in Section VII. Section VIII concludes the paper.

II. Map-Reduce Workflow Model

We refer by *workflow task sets* to those task sets that contain inter-dependent sequential jobs. This is in contrast to *independent task sets* in which no job dependencies are present. The high-level idea of our technique is to transform a MapReduce workflow task set τ into an independent task set $\overline{\tau}$, implemented as Packing server budgets, such that the schedulability of $\overline{\tau}$ is a sufficient condition for the schedulability of τ. The transformation is done at the cost of introducing increased (virtual) computation times in $\overline{\tau}$, leaving $\overline{\tau}$ at a higher utilization than τ. We show that the utilization of $\overline{\tau}$ is at most $\frac{\beta}{\varphi-\beta}$ times larger than τ, where β is a tunable parameter that controls the maximum individual server budget size. Hence, if an independent task set scheduler $\mathcal{A}$ offers a utilization bound U_B, the MapReduce workflow set τ can meet all deadlines provided that its utilization stays below $U_B \cdot \frac{\varphi-\beta}{\varphi}$.

The MapReduce literature uses terminology differently from the real-time literature, leading to a potential confusion over what is meant by such terms as jobs and tasks. In this paper, we follow the definitions common in real-time literature as much as possible. We say that a MapReduce *job* consists of a *map phase* and/or a *reduce phase*. It may be that a MapReduce job contains only a single map or reduce phase, although commonly it contains both. Each phase contains multiple *segments* that may execute in parallel. We call a segment a *mapper* or a *reducer* depending on whether it belongs to a map phase or a reduce phase. The execution of a mapper or a reducer occupies a *resource slot* (or just *slot*), which could, for example, be a core in a multi-core platform. Within one job, no reducer may start before all mappers finish. A MapReduce *pipeline* chains multiple MapReduce jobs together, resulting in a sequence of phases. In a general MapReduce *workflow*, MapReduce jobs collectively form a Directed Acyclic Graph (DAG), where each node represents a MapReduce phase and each edge points from node a to node b represents the dependency that phase b cannot start before phase a finishes.

A MapReduce workflow task τ_i is a periodic task that generates a MapReduce workflow every T_i time units with relative deadline D_i. We denote the number of segments (mappers or reducers) at the j^{th} phase of the workflow of task τ_i by m_i^j, and the worst-case computation time of an individual segment by c_i^j. A MapReduce workflow task set τ contains multiple MapReduce workflow tasks. Usually, the input of a MapReduce workflow task invocation depends on the output of the previous workflow invocation from the same task, resulting in an implicit-deadline task model ($D_i = T_i$).

We define the stretch of workflow task τ_i, denoted by φ_i, as the ratio of relative deadline D_i over critical path length, denoted by L_i. Let φ denote the minimum stretch of all workflow tasks, $\varphi = \min\{\varphi_i | \forall i\}$. Note that, if the workflow contains a single path, $path$ (i.e., it is a pipeline), $L_i = \sum_{j \in path} c_i^j$ summed over the path. The critical path in a DAG workflow is the longest execution path in the DAG. Hence, $L_i = \max_{path_k \in DAG} \sum_{j \in path_k} c_i^j$

Please note that the workflow model enjoys the same expressiveness as the generalized parallel task model [23], as any instance of the latter model can be transformed into a workflow model by constructing a single-mapper MapReduce job for each node in the generalized parallel task model. Hence, the above terminology is introduced merely for semantic convenience of mapping results to the MapReduce application world. Different from the typical multiprocessor scenario, the MapReduce platform usually spans thousands of machines [27], and a single phase may contain as many as 30 thousand segments [17, 18]. This encourages us to pay special attention to cases where m is large. Moreover, since deadlines and parallelism are large, we are interested in scenarios of large stretch, φ.

III. The Packing Server Utilization Bound

In this section, we restrict each workflow to contain a single MapReduce job. The analysis is generalized to pipelines and DAGs in Section IV.

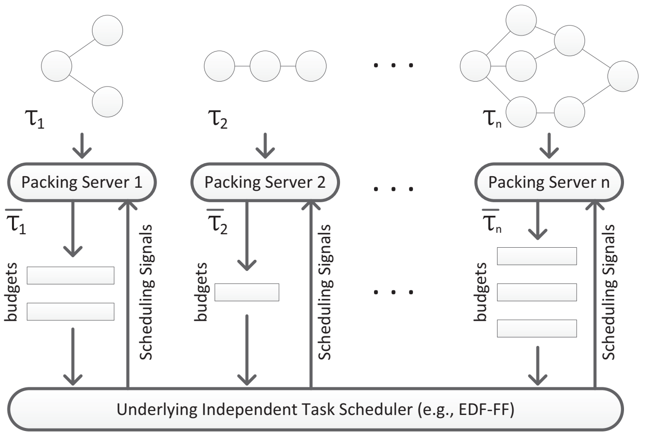


Figure 1: Packing Server Architecture

A. The Packing Server

For each task τ_i in the MapReduce task set τ, we propose to create a Packing server, $\overline{\tau_i}$, given by several parallel budgets, where the size of each budget of server $\overline{\tau_i}$ is denoted $\overline{c_i}$ and the number of budgets $\overline{m_i}$ (also called, server *concurrency*). The name, *Packing server*, was chosen because the server "packs" segments of the original workflow task into a smaller set of budgets. The set of Packing servers is collectively called set $\overline{\tau}$.

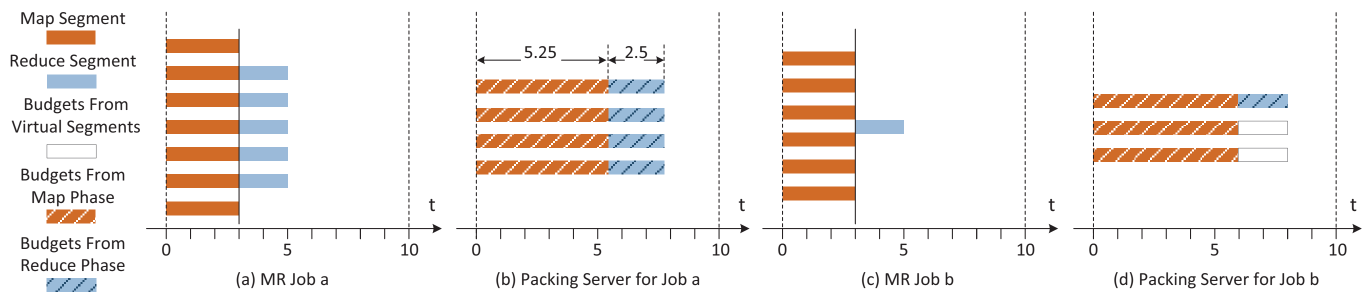
As we show later in this paper, a property of how segments are packed into budgets is that these budgets can be scheduled by the underlying scheduler as if they were *independent tasks*. They can be migrated among cores, preempted, and prioritized as the underlying scheduling policy requires, without impacting the ability of the Packing server to respect workflow synchronization (*i.e.*, segment precedence) constraints. Clearly, the budgets have to be sized such that: (i) collectively, they fit all workflow segments and (ii) individually, they fit the critical path of the workflow. Below, we describe how budget size is chosen, then prove a conversion factor bound that expresses the utilization bound for schedulability of workflows in terms of the utilization bound of the underlying scheduler.

Figure 1 shows the high-level architecture of how Packing servers work. Note that, all budgets in the same Packing server have the same budget size. A Packing server is valid if its budget size is smaller than the relative deadline of the original MapReduce job ($\overline{c_i} \leq D_i$).

B. The Case of a Single Job

We first consider the special case where the workflow task, τ_i, is composed of a single path represented by a succession of one map phase and one reduce phase. We give a smaller index to the phase with the larger number of segments. Hence, $m_i^1 \geq m_i^2$. Without loss of generality, we assume that m_i^1 is a map phase. (The discussion applies equally if m_i^1 was a reduce phase.)

In choosing server concurrency, $\overline{m_i}$, we note that some independent task scheduling algorithms on multiprocessors



Map Segment
Reduce Segment
Budgets From Virtual Segments
Budgets From Map Phase
Budgets From Reduce Phase
5.25
2.5
t
0
5
10
(a) MR Job a
(b) Packing Server for Job a
(c) MR Job b
(d) Packing Server for Job b

Figure 2: Improved Packing Server Construction Scheme

are sensitive to the maximum individual task utilization $u_{\max}$ [25, 26]. The smaller the maximum individual task utilization, the better the schedulability bound. Hence, we introduce a tunable parameter, β, to curb $u_{\max}$ of converted independent server budgets. Intuitively, the Packing server treats $D'_i = \frac{D_i}{\beta}$ as the worst-case allowable budget size, guaranteeing individual budget utilization to be upper bounded by $\frac{1}{\beta}$ (*i.e.*, $u_{\max} \le \frac{D'_i}{D_i} = \frac{1}{\beta}$). In order to pack a MapReduce job into an interval of D'_i time units, we derive two lower-bounds on Packing server concurrency, $\overline{m_i}$, that stem from the following conditions:

- *The total WCET condition*: In order to fit all original computation of the workflow task into $\overline{m_i}$ budgets of length no longer than D'_i, we should satisfy:

$$\overline{m_i} \ge \left\lceil \frac{m_i^1 c_i^1 + m_i^2 c_i^2}{D'_i} \right\rceil \tag{1}$$

- *The critical path condition*: In order to allow the MapReduce job to finish in D'_i time units, the phase with more segments (phase 1 according to our indexing) has to finish in $D'_i - c_i^2$ time units. Therefore:

$$\overline{m_i} \ge \left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil \tag{2}$$

Please note, that these two lower bounds do not dominate each other. For example, in Figure 2 (a), the MapReduce job contains $m_i^1 = 7$ mappers of WCET $c_i^1 = 3$, and $m_i^2 = 5$ reducers of WCET $c_i^2 = 2$. Deadline D_i was 10, and β is set to 1 (*i.e.*, $D'_i = D_i$). In this example, the first lower bound wins, as it results in $\overline{m_i} = \left\lceil \frac{m_i^1 c_i^1 + m_i^2 c_i^2}{D'_i} \right\rceil = 4$, whereas the second lower bound leads to $\overline{m_i} = \left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil = 3$. Figure 2 (c) depicts another example, where the original MapReduce job consists of $m_i^1 = 6$ mappers of WCET $c_i^1 = 3$, and $m_i^2 = 1$ reducers of WCET $c_i^2 = 2$. Under this configuration, the first lower bound results in $\overline{m_i} = 2$ budgets, whereas the second is $\overline{m_i} = 3$ budgets. Hence, we have the following two cases:

Case 1: When $\left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil < \left\lceil \frac{m_i^1 c_i^1 + m_i^2 c_i^2}{D'_i} \right\rceil$, the MapReduce job concentrates to $\left\lceil \frac{m_i^1 c_i^1 + m_i^2 c_i^2}{D'_i} \right\rceil$ budgets of budget size $\frac{C_i}{\lceil C_i / D'_i \rceil}$, where $C_i = m_i^1 c_i^1 + m_i^2 c_i^2$. Figures 2 (a)-(b) depict an example. As this construction strategy introduces no extra computation, the result Packing server $\overline{\tau_i}$ shares the same utilization with its original MapReduce task τ_i ($\overline{u_i} = u_i$).

Case 2: When $\left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil \ge \left\lceil \frac{m_i^1 c_i^1 + m_i^2 c_i^2}{D'_i} \right\rceil$, the reduce phase originally has less segments than $\left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil$. Therefore, we add $\left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil - m_i^2$ virtual reducers, as shown in Figures 2 (c)-(d). Together with virtual reducers, the original MapReduce job converts to $\overline{m_i} = \left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil$ budgets. In each budget, the map phase and the reduce phase (with virtual reducers) contribute $\frac{m_i^1 c_i^1}{\lceil m_i^1 c_i^1 / (D'_i - c_i^2) \rceil}$ and c_i^2 execution time respectively, resulting in the budget size of $\overline{c_i} = \frac{m_i^1 c_i^1}{\lceil m_i^1 c_i^1 / (D'_i - c_i^2) \rceil} + c_i^2$.

C. Conversion Penalty

It is crucial that we bound the utilization penalty introduced during Packing server construction, which directly affects the *conversion factor* when bridging schedulability utilization bound from independent tasks to MapReduce tasks.

Lemma 1. *The utilization ($\overline{u_i}$) of the Packing server $\overline{\tau_i}$ is at most $\frac{\varphi_i u_i}{\varphi_i - \beta}$, and the maximum individual independent task utilization is at most $\frac{1}{\beta}$, where φ_i is the stretch of MapReduce task τ_i, and $\beta \in [1, \varphi]$ is a tunable parameter.*

***Proof*.** We prove the lemma holds for the two construction cases separately:

Case 1: $\left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil < \left\lceil \frac{m_i^1 c_i^1 + m_i^2 c_i^2}{D'_i} \right\rceil$.

As the Packing server construction procedure introduces no utilization penalty in this case, the utilization of the Packing server $\overline{\tau_i}$ equals the utilization of its original MapReduce task τ_i (*i.e.*, $\overline{u_i} = u_i < \frac{\varphi_i u_i}{\varphi_i - \beta}$). Therefore, the lemma holds for case 1.

Case 2: $\left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil \ge \left\lceil \frac{m_i^1 c_i^1 + m_i^2 c_i^2}{D'_i} \right\rceil$.

The concurrency is $\left\lceil \frac{m_i^1 c_i^1}{D'_i - c_i^2} \right\rceil$. Hence, the number of virtual reduce segments is $\left(\left\lceil \frac{m_i^1 c_i^1}{D_i'-c_i^2}\right\rceil - m_i^2\right)$, each of length c_i^2. Define η_i to be $\frac{c_i^1}{c_i^2}$. Then, we have:

$$\begin{aligned}
\frac{\overline{u_i}-u_i}{u_i} &= \frac{\left(\left\lceil \frac{m_i^1 c_i^1}{D_i'-c_i^2}\right\rceil - m_i^2\right) c_i^2}{m_i^1 c_i^1 + m_i^2 c_i^2} \\
&\le \frac{\left(\left\lceil \frac{m_i^1 c_i^1}{D_i'-c_i^2}\right\rceil - 1\right) c_i^2}{m_i^1 c_i^1 + c_i^2} && \text{as } m_i^2 \ge 1 \\
&\le \frac{\left(\frac{m_i^1 c_i^1}{D_i'-c_i^2}\right) c_i^2}{m_i^1 c_i^1 + c_i^2} \\
&= \frac{m_i^1 \eta_i}{(m_i^1\eta_i+1)\left(\frac{D_i'}{c_i^2}-1\right)} \\
&= \frac{m_i^1 \eta_i}{(m_i^1\eta_i+1)\left(\frac{D_i}{\beta c_i^2}-1\right)} && \text{as } D_i' = \frac{D_i}{\beta} \\
&= \frac{m_i^1 \eta_i}{(m_i^1\eta_i+1)\left(\frac{\varphi_i\eta_i}{\beta}+\frac{\varphi_i}{\beta}-1\right)} && \text{as } \varphi_i = \frac{D_i}{c_i^1+c_i^2} \\
&= \frac{m_i^1 \eta_i}{m_i^1\eta_i+1}\cdot\frac{\beta}{\varphi_i\eta_i+(\varphi_i-\beta)} \\
&\le \frac{\beta}{\varphi_i-\beta} && (3)
\end{aligned}$$

Reorganizing the result from Inequality 3, we have:

$$\overline{u_i} \le \frac{\varphi_i u_i}{\varphi_i-\beta}. \qquad (4)$$

Hence, Lemma 1 holds for both Packing server construction cases. ■

Lemma 1 directly leads to a conversion factor bound of $\frac{\varphi-\beta}{\varphi}$, where $\varphi = \min\{\varphi_i|\forall i\}$.

IV. MapReduce Workflow Bounds

Real world MapReduce applications usually call for multiple MapReduce jobs that form a pipeline or a DAG to accomplish complex missions. In the pipeline model, MapReduce jobs are chained together one after another, resulting in a sequence of phases. The DAG model is more generalized such that the dependencies among MapReduce jobs may form a directed acyclic graph.

In this section, we first discuss how to generalize the utilization penalty bound in Lemma 1 to MapReduce Pipelines. Then we show that any MapReduce DAGs can be transformed into a pipeline with the same critical path length L_i and utilization u_i, implying that the utilization penalty bound for MapReduce pipelines also applies to MapReduce DAGs.

A. MapReduce Pipelines

MapReduce pipeline i connects n_i MapReduce jobs one after another, resulting in no more than $2n_i$ map/reduce phases. Given a number x, if $m_i^j > x$, phase j is called a *x-large phase*. Otherwise, it is a *x-small phase*. Using the similar strategy described in Section III-A, the Packing server concentrates the total WCET of each x-large phase into x identical segments, and adds $\left(x - m_i^j\right)$ virtual segments to each x-small phase j. After that, the Packing server concatenates each (virtual) segment with another (virtual) segment in the next phase, resulting in x budgets. Binary search can be used to find the minimum $x = \overline{m_i}$ such that the budget size $\overline{c_i}$ does not exceed $D_i' = \frac{D_i}{\beta}$. That is to say, using only $x = \overline{m_i}-1$ budgets would violate the deadline D_i'. Then, we have:

$$\sum_{\{j|m_i^j \ge \overline{m_i}-1\}} \frac{m_i^j c_i^j}{\overline{m_i}-1} + \sum_{\{j|m_i^j < \overline{m_i}-1\}} c_i^j \ge D_i'. \qquad (5)$$

The definitions of the deadline D_i and the stretch φ_i further lead to the following inequality:

$$D_i' = \frac{D_i}{\beta} \ge \frac{\varphi_i}{\beta}\sum_j c_i^j. \qquad (6)$$

Combining Inequalities (5) and (6), we have:

$$\sum_{\{j|m_i^j \ge \overline{m_i}-1\}} \frac{m_i^j c_i^j}{\overline{m_i}-1} + \sum_{\{j|m_i^j < \overline{m_i}-1\}} c_i^j \ge \frac{\varphi_i}{\beta}\sum_j c_i^j. \qquad (7)$$

Subtracting $\sum_{\{j|m_i^j < \overline{m_i}-1\}} c_i^j$ from both sides, we obtain:

$$\begin{aligned}
\sum_{\{j|m_i^j \ge \overline{m_i}-1\}} \frac{m_i^j c_i^j}{\overline{m_i}-1} &\ge \frac{\varphi_i}{\beta}\sum_j c_i^j - \sum_{\{j|m_i^j < \overline{m_i}-1\}} c_i^j \\
&\ge \frac{\varphi_i}{\beta}\sum_j c_i^j - \sum_j c_i^j \\
&= \left(\frac{\varphi_i}{\beta}-1\right)\sum_j c_i^j \qquad (8)
\end{aligned}$$

Based on Inequality (8), the total amount of computation requirement can be bounded from below:

$$\begin{aligned}
\sum_j m_i^j c_i^j &\ge \sum_{\{j|m_i^j \ge \overline{m_i}-1\}} c_i^j m_i^j \\
&\ge (\overline{m_i}-1)\left(\frac{\varphi_i}{\beta}-1\right)\sum_j c_i^j \qquad (9)
\end{aligned}$$

According to Inequality (8) and (9), we have:

$$
\begin{aligned}
\frac{\overline{u_i}-u_i}{u_i} &= \frac{\sum\limits_{\{j|m_i^j<\overline{m_i}\}} \left(\overline{m_i}-m_i^j\right) c_i^j}{\sum\limits_j m_i^j c_i^j} \\
&\leq \frac{(\overline{m_i}-1) \sum\limits_{\{j|m_i^j<x\}} c_i^j}{\sum\limits_j m_i^j c_i^j} && (\text{as } m_i^j \geq 1) \\
&\leq \frac{(\overline{m_i}-1) \sum\limits_{\{j|m_i^j<\overline{m_i}\}} c_i^j}{(\overline{m_i}-1)\left(\frac{\varphi_i}{\beta}-1\right)\sum\limits_j c_i^j} && (\text{Equation } 9) \\
&\leq \frac{\beta}{\varphi_i-\beta} && (10)
\end{aligned}
$$

Therefore, the same utilization penalty bound $\frac{\beta}{\varphi-\beta}$ holds for MapReduce pipelines.

B. Transforming DAGs into Pipelines

This section further generalizes the same utilization penalty bound $\frac{\beta}{\varphi-\beta}$ to MapReduce DAGs by transforming a MapReduce DAG into a MapReduce pipeline. There are many different ways to transform a MapReduce DAG into a MapReduce Pipeline. One naive solution would perform a topology sort on the DAG, and execute phases one after another according to the sorted order. However, this solution enlarges the critical path length L_i, leading to a smaller φ after conversion, and hence a larger utilization penalty bound $\frac{\beta}{\varphi-\beta}$.

Our goal is to develop a strategy that leads to the lowest utilization penalty bound. As shown above, the utilization penalty bound for a MapReduce pipeline is $\frac{\beta}{\varphi-\beta}$, which decreases with the increase of $\varphi = \frac{D_i}{L_i}$. This inspires us to design an algorithm that minimizes the pipeline critical path length L_i during Packing server constructions. The resulting pipeline length is bounded from below by the critical path length L_i of its original MapReduce DAG, where the utilization penalty introduced by Packing servers is minimized. This can be achieved by allowing each node in the DAG to start execution as soon as all its prerequisite nodes finish. To keep the result as a valid pipeline, a synchronization point is inserted when each DAG node finishes. Fig. 3 shows an example, where the original DAG job is depicted in (a) and the resulting pipeline is shown in (b). As phases are of different lengths, a node in the original DAG may break into multiple phases in the pipeline. For example, node 2 and node 5 both follow phase 1. Hence, these two nodes may start at the same time. However, as node 5 finishes sooner than node 2, its ending synchronization point breaks workflow of node 2 into two pipeline phases. The pseudo code is described in Algorithm 1.

Algorithm 1 Transform MapReduce Workflows to MapReduce Pipelines

```
Input: MapReduce DAG task set $\tau$
Output: MapReduce pipeline task set $\tau'$
 1: procedure TRANS-D($\tau$)
 2:     $\tau' \leftarrow \emptyset$
 3:     for $\tau_i \in \tau$ do
 4:         Sync $\leftarrow \{0\}$
 5:         for node $j$ in $\tau_i$ do
 6:             $l_i^j \leftarrow$ the earliest possible time that the node $j$ could start
 7:             Lay out all segments in node $j$ in time interval $[l_i^j, l_i^j + c_i^j]$
 8:             Sync $\leftarrow$ Sync $\cup \{l_i^j + c_i^j\}$
 9:         end for
10:         sort Sync following increasing order
11:         $\tau_i' \leftarrow \emptyset$
12:         for $j \leftarrow 2 \sim |Sync|$ do
13:             $\mathcal{P}_i^j \leftarrow$ create a phase encapsulates all segments (portions)
                  fall in time interval [Sync[j-1], Sync[j])
14:             $\tau_i' \leftarrow \tau_i' \cup \{\mathcal{P}_i^j\}$
15:         end for
16:         $\tau' \leftarrow \tau' \cup \tau_i'$
17:     end for
18:     schedule $\tau'$ using the MapReduce pipeline scheduling algorithm.
19: end procedure
```

Algorithm 1 loops over all workflow tasks on lines 2-17. For each task τ_i, the algorithm first computes its synchronization points on lines 4-10. As shown on lines 6-8, each node j in task τ_i associates with a synchronization point $l_i^j + c_i^j$, where l_i^j is its longest preceding WCET path. Lines 12-15 divide the workflow into a pipeline of phases using synchronization time points in set *Sync*. The result can be viewed as a single pipeline task set, τ'.

Note that, Algorithm 1 transforms a workflow task set into a pipeline task set without increasing its utilization. Therefore, the same utilization bound $U_B \cdot \frac{\varphi-\beta}{\varphi}$ applies to MapReduce workflows, where $\varphi = \min\{\varphi_i | \forall \tau_i \in \tau\}$.

V. Scheduling MapReduce Workflows

Previous sections introduce the strategy to convert a set of MapReduce workflow tasks into budgets that belong to a set of Packing servers, one per task. Those budgets can then be scheduled as independent sequential jobs by the underlying scheduler. Each budget is used to execute workflow segments. We now describe how the execution order of segments is determined.

Before proceeding with our description, it is good to understand the differences between traditional operating system scheduling and MapReduce scheduling, which in our system is based on Hadoop [2, 3]; an open-source MapReduce implementation. These differences are important to the understanding of our workflow scheduler implementation.

In an operating systems context, which has been the traditional scheduling context in real-time systems literature, *servers* typically refer to application tasks. The *underlying scheduler* typically refers to an OS kernel scheduler [28]. This OS kernel scheduler has the power to allocate physical resources to tasks. When it invokes a server, a *second-level scheduler* (implemented in user space) inside the server

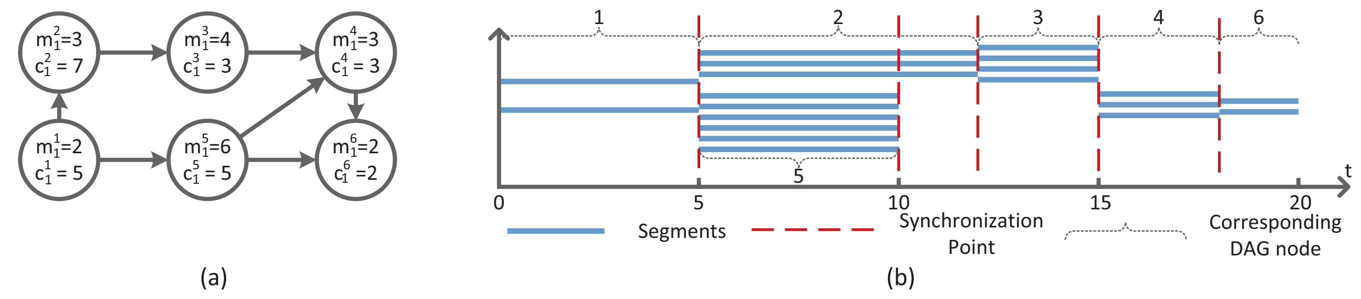


Figure 3: Transform a DAG into a pipeline: (a) shows the original DAG, and (b) shows the resulting pipeline. The pipeline consists of 6 phases containing 2, 9, 3, 4, 3, 2 segments and of lengths 5, 5, 2, 3, 3, 2 respectively.

decides on the order in which server budget is allocated to different computations.

In a MapReduce context, the picture is slightly different. First, all scheduling is done in user space, since Hadoop is an application-level implementation. Hence, the *underlying scheduler* refers to a user-level resource manager that performs coarse-grained resource assignments. In Hadoop v2 (YARN) [3], the resource manager assigns resources to the so called *application masters*. In this case, an application master acts as a Packing server. The system can be configured to have a single master per workflow task. The application master implements the second-level scheduler that decides on the order of execution of MapReduce segments of the corresponding workflow task within the server's budgets.

In an important departure from OS scheduling, in Hadoop, the application masters are assumed to be cooperative. Hence, the scheduling policy used by the Hadoop resource manager (*i.e.*, the underlying scheduler) is expressed to the application masters as an exact timeline showing when the corresponding workflow task is allowed to run and on which resources. Application masters are therefore "clairvoyant" about their exact future schedules. It is this clairvoyance that allows us to implement the abstraction of independent budgets, such that the underlying scheduler (the resource manager) does not need to know anything about workflow topologies and precedence constraints.

More specifically, once the resource manager informs application masters of their budget schedules, since each application master (*i.e.*, Packing server) of a workflow task knows when its budgets are scheduled, it can determine a sequence of synchronization time points within its budgets, *Sync*= $\{t_0, t_1, ..., t_n\}$ such that $t_0 = 0$, and the total size of scheduled budgets falling in $[t_{j-1}, t_j)$ equals the total WCET of the j^{th} phase including potential virtual segments $(c_i^j \cdot \max\{m_i^j, \overline{m_i}\})$. The time instance t_j thus becomes the time when phase $j-1$ should end and phase j begin. The application master (*i.e.*, Packing server) then packs segments of phase j into budget portions falling in $[t_{j-1}, t_j)$. The result of such packing is shown in Figure 4 for an example server composed of four budgets. Packing is done in a *best fit* manner (*i.e.*, smallest budget portion is filled first). Note that, segments within the same phase have no precedence constraints and hence can be packed in any order. Furthermore, since the execution of a segment is arbitrarily divisible, it turns out that it is always possible to pack segments such that all budgets running at time t_j finish the execution of segments of phase j simultaneously at that time. The only constraint to consider is that portions of the *same* segment cannot run in parallel on multiple processors at the same time. Hence, when scheduling a segment, the best fit policy skips time intervals where potions of the same segment have already been scheduled. The exact pseudo-code for the best fit segment packing algorithm and the proof that it always succeeds at finding a valid schedule between successive synchronization points are delegated to the appendix.

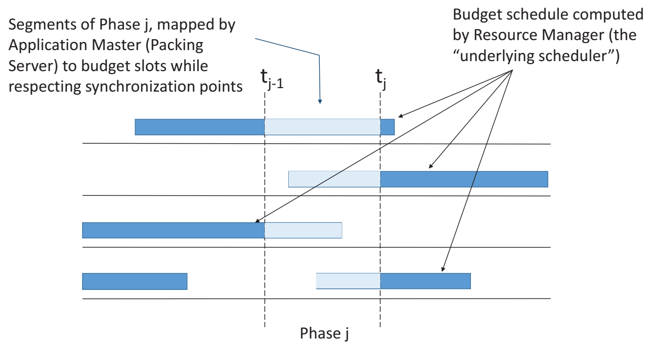

Segments of Phase j, mapped by Application Master (Packing Server) to budget slots while respecting synchronization points
Budget schedule computed by Resource Manager (the "underlying scheduler")
t_{j-1}
t_j
Phase j

Figure 4: An example of segment packing.

A. Limitations

The above discussion has been a simplified treatment of MapReduce applications. MapReduce is a complex system. A faithful analytic treatment goes beyond the scope of a single paper. It is therefore useful to outline the approximations and simplifications we made in this work.

First and foremost, we do not explicitly address data allocation. Segments of task workflows operate on data. Such data must be available on the local machine. If not, the computation time of the segment will increase. In principle,

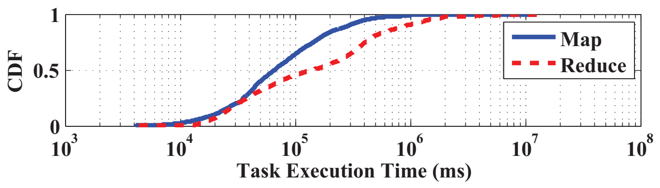


Figure 5: Yahoo! Hadoop Jobs Mapper and Reducer WCET

it is possible to plan ahead of time, such that data are distributed to machines as segments are allocated, such that each segment finds its data locally. Challenges arise in the presence of preemption and migration, when a segment might find itself resumed on a different machine. In general, moving data around is a bad idea. Hence, in practice, the underlying scheduler should consider migration and data movement costs. For example, partitioned scheduling would be highly preferable to global scheduling. The underlying scheduling policy is an orthogonal issue to our contribution and hence is not addressed in this paper.

Second, the cost of preemption in MapReduce systems is higher than that in a multi-core platform. The NatJam system [22] enables MapReduce preemption by inserting checkpoints between two key-value pairs, and writing those checkpoints into the shared distributed file system, which introduces a few milliseconds to a few seconds of delay. In real-world scenarios, this is not a big problem as MapReduce segments also take much longer time to finish compared to multi-core tasks, leading to a small relative preemption cost. For example, Figure 5 plots the distribution of mapper and reducer WCET in Yahoo! Hadoop cluster [17]. WCETs of most segments are larger than 10 seconds, and more than 50% segments take more than 1 minute to finish.

Finally, there is a data movement phase following each computational phase in a MapReduce workflow, where outputs of one set of segments are sent to the next set of segments. This movement takes place over a high-speed network interconnect. If network bandwidth is not sufficient, data movement may introduce delays that need to be explicitly accounted for. One possibility is simply to subtract those delays from end-to-end deadlines, such that the deadlines used reflect the time available for the computational part only. Better solutions will be explored in subsequent work.

VI. Evaluation

In this section, we compare our solution to two baselines. The first one is the state-of-the-art generalized parallel tasks scheduling algorithm [24], called *federated scheduling*. Among past results, federated scheduling achieves the highest-known utilization bound of 50% for generalized parallel tasks, if the stretch φ surpasses 2. The second baseline for the parallel task model is the GEDF scheduling algorithm, with a best-known utilization bound of $\frac{2}{3+\sqrt{5}} \approx$ 38.2% [26].

A. Computing the Optimal Budget Size

We begin by computing the optimal Packing server budget size (or equivalently, the optimal value of β). Packing servers may use any implicit-deadline independent task scheduling policy $\mathcal{A}$ as the underlying scheduler to schedule their budgets, while achieving a utilization bound $U_B \cdot \frac{\varphi-\beta}{\varphi}$ for schedulability of MapReduce workflows. In following experiments, we set $\mathcal{A}$ to EDF-FF [25] and (independent task) GEDF [26], respectively, as they lead to high utilization bounds when stretch, φ, is large. We compute the optimal β as follows:

- *EDF-FF:* When EDF-FF is used as the underlying policy, $\mathcal{A}$, the utilization bound guaranteed by Packing servers for MapReduce workflows becomes:

$$U_B \cdot \frac{\varphi-\beta}{\varphi} = \frac{(m\beta+1)(\varphi-\beta)}{m\varphi(\beta+1)}. \tag{11}$$

  By taking the derivative with respect to β, and setting the derivative to 0, the highest utilization bound is achieved when:

$$\beta = \sqrt{\frac{(\varphi+1)(m-1)}{m}} - 1. \tag{12}$$

- *GEDF:* When the underlying scheduling policy, $\mathcal{A}$, is set to GEDF, the schedulability bound becomes:

$$U_B \cdot \frac{\varphi-\beta}{\varphi} = \frac{(\varphi-\beta)(m\beta-m+1)}{m\varphi\beta}. \tag{13}$$

  Similarly, by equating the derivative to zero, the optimal β, that achieves the highest bound, is:

$$\beta = \sqrt{\frac{\varphi(m-1)}{m}}. \tag{14}$$

The following experiments use these two optimal β formulas to configure Packing servers when they run above EDF-FF and GEDF schedulers.

B. Schedulable Utilization

The first question we answer in the evaluation section is to empirically determine the average schedulable utilization of a MapReduce cluster, due to tasks that *meet deadlines*, under different scheduling policies. Four policies are compared: (i) Packing servers on top of EDF-FF, (ii) Packing servers on top of GEDF, (iii) the federated scheduling policy [24] (with no Packing servers), and (iv) GEDF. In industry, outputs of MapReduce workflows power a variety of services, where tardiness are usually allowed, but at the cost of diminishing monetary benefits. In our experiments, we inherit the same configuration, allowing tasks to continue execution after their deadlines, which may adversely affect the schedulability of subsequent jobs. We do *not* use admission control. Rather, we vary the total input workload utilization (as percentage of total platform capacity) on the x-axis and count on the y-axis only the utilization of tasks whose deadlines were

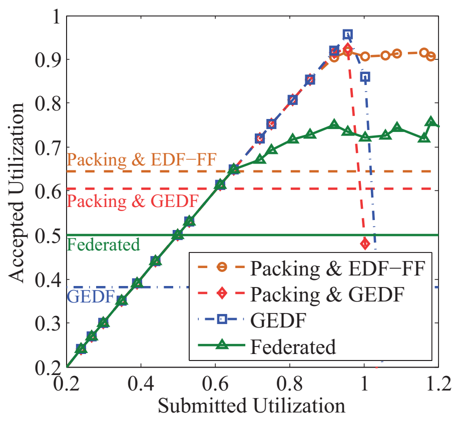


Figure 6: Schedulable Utilization, $\varphi = 20$

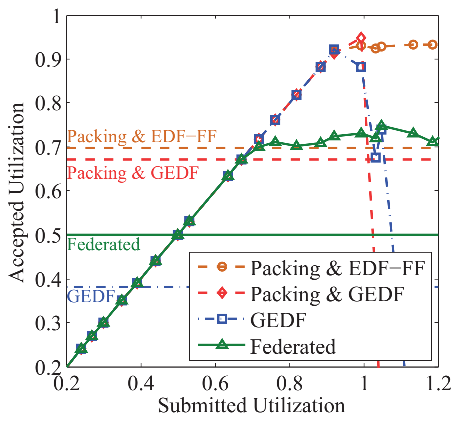


Figure 7: Schedulable Utilization, $\varphi = 30$

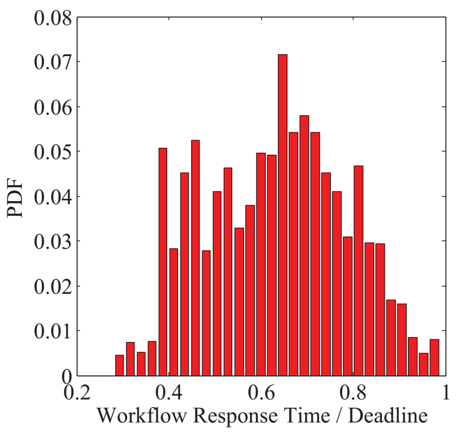


Figure 8: Admission Control

met. While we do not explicitly plot deadline misses, note that the difference between each curve and the diagonal x=y is the utilization attributed to tasks that missed deadlines. In these experiments, workflows are generated based on Yahoo! MapReduce cluster trace data [17, 18]. The Yahoo! dataset does not specify the deadline of jobs or workflows. We therefore calculate the critical path length of workflows, and set their deadlines to control the value of φ. More specifically, the number of slots m is set to 500, and the parameter φ is tuned to 20 (and 30) during our simulations, resulting in $\beta = 3.58$ (and $\beta = 4.56$) for the EDF-FF scheduler, and $\beta = 4.47$ (and $\beta = 5.47$) for the GEDF scheduler.

Figures 6-7 depict the experiment results. The horizontal lines in these figures indicate the theoretical utilization bounds for the schedulability of each of the four schemes compared. When GEDF acts as the underlying scheduler, the theoretical schedulability bounds of Packing servers are 60.3% for $\varphi = 20$, and 66.9% for $\varphi = 30$. Packing servers on top of EDF-FF have a 64% and 70% utilization bound, which are the highest under the $\varphi = 20$ and $\varphi = 30$ configurations.

We also plot the empirically determined, schedulable utilization curves for each of the four policies. Empirically, when GEDF is the underlying scheduling policy, both GEDF (alone) and Packing servers on GEDF meet deadlines of almost all tasks when the task set utilization is below 90%. However, the GEDF-based algorithms fail seriously, when the task set utilization surpasses 95%, exhibiting a "domino effect".

Federated scheduling leads to a theoretical utilization bound of 50%, which is the highest-known bound in previous work. Under MapReduce workloads, it is empirically shown to be able to schedule tasks without deadline misses up to about 70% utilization. Above that utilization deadlines are missed, although the domino effect seen by GEDF is not experienced.

Packing servers on an EDF-FF scheduler appear to be the most successful policy. The policy offers a high theoretical schedulability utilization bound, and performs very well above the bound, rising up to almost a 90% utilization without deadline misses, when serving MapReduce workloads. No domino effect is experienced above 90%. Hence, we implement this scheduler on a 46-server Hadoop cluster to verify its feasibility and validity.

C. *Meeting Deadlines in a Real Hadoop Cluster*

Next, we test the efficacy of admission control schemes based on our new bounds at eliminating all deadline misses. We implement a prototype of Packing servers on WOHA [18], a workflow-enabled variant of Hadoop v1. The experiment runs on a cluster of 40 Dell PowerEdge R620 servers and 6 Dell PowerEdge R610 servers. The 40 R620 servers form a Hadoop cluster, providing 160 reduce slots. The 6 R610 servers execute 1 resource planner, and 5 client nodes that submit workflow invocations to the Hadoop cluster. The parameter φ is set to 20. It corresponds to a utilization bound of 64.3%, which we set as the threshold for admission control. Hence, we prepare a set of workflows with a total utilization above 100%. An admission controller is used that denies a workflow if it brings the cluster utilization above 64.3%. All segments are computationally-intensive. Figure 8 shows the resulting probability density distribution of response time-to-deadline ratio of workflow invocations during a 4-hour experiment. All ratios are below 1, suggesting the validity of the utilization bound.

VII. Related Work

Workflow scheduling attracts increasing attention from both real-time and MapReduce researchers. The widespread MapReduce deployments stimulate the MapReduce community to design and improve scheduling policies for MapReduce implementations, such as Hadoop. The default scheduler executes jobs in a FIFO order, leading to poor fairness under multi-tenant scenarios. Yahoo! developed a Capacity

Scheduler [20] to offer each Hadoop cluster tenant a guaranteed resource share. Facebook's Fair Scheduler [21] organizes Hadoop jobs into pools, and fairly divides resources among these pools. Verma *et al.* [11] evaluates an EDF-based scheduling algorithm on MapReduce. Their simulation results confirm that simple deadline-based scheduling heuristics allow more jobs to meet their deadlines. All of the above solutions target job scheduling rather than workflow scheduling. Yahoo! later developed Oozie [19, 29] as a generic Hadoop workflow management tool, that submits each workflow job at the right time. WOHA [18] introduces deadline-aware scheduling of Hadoop workflows. However, these schedulers make no guarantees on whether workflow deadlines are met or not.

In real-time literature, workflow scheduling (called generalized parallel tasks scheduling) has been recently studied on multiprocessor platforms. Baruah *et al.* [30] prove that EDF can achieve a 2X speedup bound for a single recurrent workflow. Saifullah *et al.* [31] propose to arrange a workflow into stages, and then the workflow's deadline is split and assigned to each stage. If some optimal algorithm can successfully schedule the original workflow, their solution is guaranteed to satisfy the same deadline with a 4X (speedup bound) speed processors. When the workflow is restricted to a fork-join model [32, 33], Lakshmanan *et al.* [34] improve the speedup bound to 3.42. Li *et al.* [35] develop a capacity augmentation bound of $4 - \frac{2}{m}$ for workflows, which immediately leads to a simple and effective schedulability test. More recently, Li *et al.* [24] improve the capacity augmentation bound to 2 using the federated scheduling algorithm. Nevertheless, a utilization below 50% may be pessimistic for industry MapReduce clusters.

Independent tasks on multiprocessors have been studied more extensively during the past decades. Some algorithms push the schedulability bound to be much higher than 50%. The EDF-FF (first fit) algorithm is able to schedule all tasks if their total utilization stays below $U_B = \frac{m\beta+1}{\beta+1}$, where β is the inverse of the maximum individual task utilization (*i.e.*, $\beta = \frac{1}{u_{\max}}$). The global EDF guarantees schedulability if the total utilization is less than $U_B = m\left(1 - \frac{1}{\beta}\right) + \frac{1}{\beta}$. Both algorithms approach a 100% utilization bound when β and m are large, which is common on MapReduce platforms. These observations motivate us to develop the Packing server technique to be able to apply those higher bounds from independent task scheduling to MapReduce workflows.

VIII. Conclusion

This paper introduces the technique of Packing server to convert independent task set schedulability bounds to MapReduce workflows schedulability bounds. If an independent task set scheduler $\mathcal{A}$ guarantees schedulability up to total utilization U_B, the Packing servers can achieve schedulability bound of $U_B \cdot \frac{\varphi-\beta}{\varphi}$ using $\mathcal{A}$ as the underlying scheduler, where φ is the minimum deadline to critical path ratio, and $\beta \in [1, \varphi]$ is a tunable parameter that curbs the maximum converted individual independent task utilization ($u_{\max} \leq \frac{1}{\beta}$). MapReduce workflows usually yield large φ, allowing the new bound to achieve a much higher value than the best known bound of 50%. Our evaluations using Yahoo! data on a 46-server Hadoop cluster confirm the validity of the new bound and the feasibility of the Packing server system design.

Acknowledgement

This work was sponsored in part by the National Science Foundation under grants CNS 13-20209, CNS 13-02563, CNS 10-35736 and CNS 09-58314. We are also thankful to Yahoo! Inc. for sharing the access to Webscope data.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *USENIX OSDI*, 2004.

[2] "Apache Hadoop," http://hadoop.apache.org/, October 2014.

[3] "Apache Hadoop V2 (YARN): Yet Another Resource Negotiator," http://hortonworks.com/hadoop/yarn/, October 2014.

[4] "Apache spark," https://spark.apache.org/, May 2014.

[5] S. Li, T. Abdelzaher, and M. Yuan, "Tapa: Temperature aware power allocation in data center with map-reduce," in *IEEE IGCC*, 2011.

[6] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *IEEE RTAS*, 2014.

[7] J. Lee, A. Easwaran, and I. Shin, "Maximizing contention-free executions in multiprocessor scheduling," in *IEEE RTAS*, 2011.

[8] C. Liu and J. H. Anderson, "Suspension-aware analysis for hard real-time multiprocessor scheduling," in *ECRTS*, 2013.

[9] M. A. Haque, H. Aydin, and D. Zhu, "Real-time scheduling under fault bursts with multiple recovery strategy," in *IEEE RTAS*, 2014.

[10] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ mapreduce for log processing," in *USENIX ATC*, 2011.

[11] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Campbell, "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle," in *IEEE NOMS*, 2012.

[12] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *IEEE CLOUDCOM 2010*.

[13] K. Agrawal, C. Gill, J. Li, M. Mahadevan, D. Ferry, and C. Lu, "A real-time scheduling service for parallel tasks," in *IEEE RTAS*, 2014.

[14] R. M. Pathan, P. Stenstróm, L.-G. Green, T. Hult, and P. Sandin, "Overhead-aware temporal partitioning on multicore processors," in *IEEE RTAS*, 2014.

[15] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *IEEE RTAS*, 2010.

[16] C. J. Kenna, J. L. Herman, B. C. Ward, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *ECRTS*, 2013.

[17] "Yahoo! webscope," http://webscope.sandbox.yahoo.com/, 2014.

[18] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, "Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop cluster," in *IEEE ICDCS*, 2014.

[19] "Apache oozie," http://oozie.apache.org/, May 2014.

[20] "Capacity scheduler," http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html, May 2014.

[21] "Fair scheduler," http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html, May 2014.

[22] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *ACM SoCC*, 2013.

[23] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012.

[24] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *ECRTS*, 2014.

[25] J. M. López, M. García, J. L. Díaz, and D. F. García, "Worst-case utilization bound for edf scheduling on real-time multiprocessor systems," in *ECRTS*, 2000.

[26] T. P. Baker, "A comparison of global and partitioned edf schedulability tests for multiprocessors," In International Conf. on Real-Time and Network Systems, Tech. Rep., 2005.

[27] J. Wong, "Which big data company has the worlds biggest hadoop cluster?" http://www.hadoopwizard.com/, September 2008.

[28] G. Lipari and E. Bini, "A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation," in *IEEE RTSS*, 2010.

[29] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur, "Oozie: towards a scalable workflow management system for hadoop," in *ACM SWEET*, 2012.

[30] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *IEEE RTSS*, 2012.

[31] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *IEEE RTSS*, 2011.

[32] C. Maia, L. Nogueira, L. M. Pinho, and M. Bertogna., "Response-time analysis of fork/join tasks in multiprocessor systems," in *ECRTS*, 2013.

[33] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems (RTNS)*, 2014.

[34] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *IEEE RTSS*, 2010.

[35] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global edf for parallel tasks," in *IEEE ECRTS*, 2013.

Appendix

A. Scheduling Segments on Budgets

As Algorithm 1 converts MapReduce workflows into MapReduce pipelines without introducing any utilization penalty, we discuss the scheduling algorithm in the context of a MapReduce pipeline for sake of simplicity. In order to further simplify the notation, we focus on a single MapReduce pipeline invocation from MapReduce task τ_i, saving subscripts for task ID and pipeline ID. The algorithm schedules each phase into its budget portions in a *First-Fit* manner. Starting from the first phase, let $\pi = \{\pi_1, \pi_2, ..., \pi_z\}$ denote the set of budget portions for the first phase, where $z \leq \overline{m_i}$. Please note, some budget may completely fall in time interval $[t_1, +\infty)$, leaving z to be smaller than $\overline{m_i}$. A budget portion π_i is a set of reserved non-overlapping time intervals on some resource slots. Hence, each budget portion associates with two affinities, time and slot. Let $\mathcal{N}(\pi)$ represent the number of budget portions in π (*i.e.*, $\mathcal{N}(\pi) = z$), and $\mathcal{L}(\pi)$ the total size of budget portions in π (*i.e.*, $\mathcal{L}(\pi) = \sum_i \mathcal{L}(\pi_i)$). Without loss of generality, assume the set π is ordered such that $\mathcal{L}(\pi_i) \leq \mathcal{L}(\pi_{i+1})$. As two budgets cannot execute on the same slot at the same time, we have $\mathcal{L}(\pi_u \backslash \pi_v) = \mathcal{L}(\pi_u)$ and $\mathcal{L}(\pi_u \cup \pi_v) = \mathcal{L}(\pi_u) + \mathcal{L}(\pi_v)$, for $u \neq v$. Algorithm 2 schedules x segments of WCET y on the budget portion set π.

Algorithm 2 Schedule a phase on its budget portions

Input: π the set of budget portions, x the number of segments, y the length of each segment
Output: $\mathcal{S}$ the schedule of input segments using in input budget portions

```
1: procedure SCHEDSEG($\pi$, $x$, $y$)
2:     $\mathcal{S} \leftarrow \emptyset$
3:     for $j \leftarrow 1 \sim x$ do
4:         $l \leftarrow y$
5:         for $i \leftarrow 1 \sim \mathcal{N}(\pi)$ do
6:             Schedule the length-$l$ segment on $\pi_i$ following the increasing
                 order of time, skipping all conflicting time instances.
7:             Store the scheduled part in $\pi_i^j$
8:             $l \leftarrow l - \mathcal{L}\left(\pi_i^j\right)$
9:             $\mathcal{S} \leftarrow \mathcal{S} \cup \{\pi_i^j\}$
10:        end for
11:        if $l > 0$ then
12:            return null
13:        end if
14:    end for
15:    return $\mathcal{S}$
16: end procedure
```

The algorithm schedules all segments one by one (Lines 3-14). For each segment, it always tries to use smaller budget portions first (Lines 5-10). A segment fills budget portion π_i^j following the increasing order of time. Remaining parts in π_i^j will be filled by following segments. In order to prevent segment-level parallelism, the algorithm skips all conflicting time intervals when scheduling segments.

As Algorithm 2 only requires a budget portion set and segments' properties, it also applies to the subsequent phase j in the pipeline, by setting x to $\max\{m_i^j, \overline{m_i}\}$, y the WCET c_i^j, and π the budget portions in time interval $[t_{j-1}, t_j)$.

B. Algorithm Correctness

We now prove that Algorithm 2 guarantees to successfully schedule a phase on corresponding budget portions. Again, in order to simplify the notation, we focus on one single MapReduce pipeline from MapReduce task τ_i, and use the first phase as the proof subject, saving the notations for task ID, and phase ID.

The proof can be divided into two cases depending on whether the size of the smallest budget portion $\mathcal{L}(\pi_1)$ is larger than segment WCET y.

Case 1: $\mathcal{L}(\pi_1) > y$.

Algorithm 2 always tries to fill up budget portion π_i before it starts to use π_{i+1}, unless segment-level parallelism conflicts prevent it from achieving that. In the case of $\mathcal{L}(\pi_1) > y$, a segment α can either completely fit into the current budget portion π_i, or exhaust the remaining parts of the π_i and start to use π_{i+1}. As $\pi_{i+1} \geq \pi_1 > y$, the unscheduled parts of α can always fit into π_{i+1} avoiding parallelism conflicts from π_i. Therefore, budget portions are filled up one-by-one following their index order, leaving no gap in the middle. Due to $\mathcal{L}(\pi) \geq xy$, all segments can be scheduled in π.

Case 2:$\mathcal{L}(\pi_1) \leq y$.

We apply induction on z.

Basis: When $z = 1$, all segments are scheduled sequentially into a single budget portion. The induction hypothesis trivially holds.

Inductive Step: Assume the lemma holds for $z \leq k - 1$, we now prove it also holds for $z = k$. There are two cases:

As Algorithm 2 is deterministic, it is easy to figure out which parts of π are assigned to the first segment. Remove those parts from π, and denote the resulting budget portion set as π' . Now, in order to prove the lemma, we only need to show that Algorithm 2 is able to fit the remaining $x - 1$ length-y segments into schedule π'. Given $\mathcal{N}(\pi) \leq x$, $\mathcal{L}(\pi_1) \leq y$, and $\mathcal{L}(\pi) \geq xy$, we have $\mathcal{N}(\pi') \leq x - 1$ and $\mathcal{L}(\pi') = \mathcal{L}(\pi) - y \geq (x - 1)y$. Therefore, according to the induction hypothesis, Algorithm 2 can fit $x - 1$ length-y segments into π', implying that the lemma also holds for $z = k$. ■