Multi Resource Scheduling with Task Cloning in Heterogeneous Clusters

Huanle Xu*
University of Macau
Macau, China

Yang Liu
Shanghai University
Shanghai, China

Wing Cheong Lau
The Chinese University of Hong Kong
Hong Kong, China

ABSTRACT

To mitigate the straggler effect, today's systems and computing frameworks have adopted redundancy to launch extra copies for stragglers. Two limitations of the existing straggler-mitigation techniques, however, are that resource demand of tasks is only considered in the context of slots and, moreover, redundancy is seldom coordinated with job scheduling. To tackle these issues, in this paper, we present DollyMP, a job scheduler that addresses multi-resource scheduling with task cloning in heterogeneous clusters. DollyMP carefully combines SRPT (Shortest Remaining Processing Time) and SVF (Smallest Volume First) via knapsack optimization to schedule tasks with multi-resource demands and, in the meanwhile, dynamically launches task clones to yield a small job completion time. DollyMP is built on a strong mathematical foundation to guarantee near-optimal performance. The deployment of our Hadoop YARN prototype on a 30-node cluster demonstrates that DollyMP can reduce job response time by 50% under different cluster loads.

ACM Reference Format:
Huanle Xu, Yang Liu, and Wing Cheong Lau. 2022. Multi Resource Scheduling with Task Cloning in Heterogeneous Clusters . In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545093

1 INTRODUCTION

Modern data centers typically consist of thousands of heterogeneous servers to handle explosive growth of cloud applications. Within such large-scale infrastructure, parallel and distributed computing have become dominant techniques to achieve efficiency and scalability. Computing frameworks such as MapReduce [14] and Spark [50] split jobs into multiple small tasks which can run in parallel on different servers so as to leverage system resources and significantly accelerate job completion. A key challenge of catalyzing the widespread adoption of these computing frameworks is the disproportionately long-running tasks, or the so-called stragglers, which corresponding to tasks that are unfortunately assigned to servers suffering from a low processing rate. Stragglers can easily lead to unpredictable job completion time and thus degrade the system performance substantially [8]. Measurement traces from the Facebook cluster show that stragglers in Hadoop clusters can run up to 8× slower than normal tasks [5, 38].

The problem of stragglers has received considerable attention from both industry and academic research, with a slew of straggler mitigation techniques being developed by making use of redundancy. These techniques can be broadly divided into two categories, namely, Speculative Execution [4, 7, 8, 14, 23] and Cloning [5, 33]. We next explain more details about these two techniques.

For speculative execution, the progress of each task is monitored by the system and backup copies are launched for those tasks that are running much slower than others. On the one hand, speculative execution techniques suffer from a fundamental limitation, i.e., stragglers can be detected when a job is close to completion and thus relaunching duplicates does not help much. This situation is particularly challenging for small jobs since it is difficult to collect enough statistically significant samples of tasks for small jobs. On the other hand, speculative execution can also incur extra system instrumentation and performance overhead. The case is even worse when the progress of a large number of tasks has to be tracked [46]. Under the cloning approach, extra copies of a task are scheduled in parallel with the initial task, and the one which finishes first is used for the subsequent computation [46]. Though cloning is simple to implement and needs no monitoring, it can incur additional resource contention since it consumes more resources comparing to speculative execution. Nevertheless, today's clusters are underutilized at most times since they are heavily over-provisioned to satisfy their peak demand of over 99% [5]. In particular, traces analysis from Google clusters indicates the average utilization is less than 50% and 95% of jobs are small (duration less than two hours) [36]. As a consequence, there is a large room to make clones for jobs running in production clusters. In this paper, we resort to the use of cloning for improving job performance.

However, one common drawback shared by all existing straggler mitigation approaches is that they consider each server in the cluster is configured into multiple static slots and each slot can hold one task from any job. Obviously, these schemes cannot apply to today's resource scheduling models in modern clusters where each task can have different resource demands with multi-dimensional requirements [20, 25]. In addition, most of these speculative execution/cloning schemes are designed independently with job scheduling via simply allocating slots to speculative/cloned copies in a "best-effort" fashion, e.g., [5, 7, 8, 14]. Speculation/cloning aggressively can improve the performance of the job at hand but may hurt the performance of other jobs, since extra available resources are occupied by redundant copies instead of new tasks.

Contributions of this paper: In this paper, we present DollyMP, a job scheduler that combines multi-resource scheduling with task cloning in heterogeneous clusters. DollyMP is built based

*Huanle Xu is also with Guangdong-Hong Kong-Macao Joint Laboratory of Human-Machine Intelligence-Synergy Systems.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545093

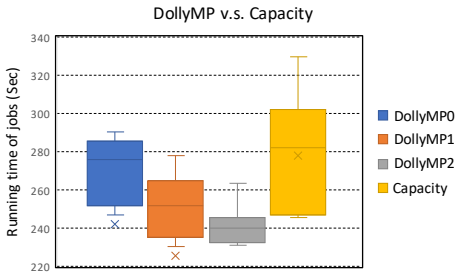


Figure 1: Job running time under different schedulers.

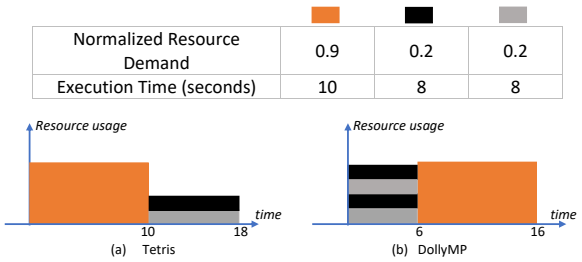


Figure 2: The comparison between different scheduling algorithm. The Tetris Scheduler shown in the left panel achieves a total completion time of 46 seconds for the three jobs. However, the total completion time under the DollyMP scheduler is only 28 seconds and DollyMP makes one clone for both Job 2 and Job 3 when they are scheduled (the cloned copy is in the same color as its original copy).

on a general optimization framework aiming at minimizing the overall job completion time by combining job scheduling with task cloning. DollyMP relies on an interesting observation that, when the execution time of stragglers is heavy-tail distributed, cloning small jobs can help to reduce the overall job completion time a lot. To tackle the multi-resource scheduling problem, DollyMP balances the scheduling between jobs of small volume [26] and those which can lead to resource fragmentation, by solving multiple knapsack problems. As a consequence, DollyMP can yield a more compact and efficient resource allocation result comparing to SVF (Smallest Volume First). More importantly, we prove that DollyMP manages to achieve a competitive ratio of $O(1)$ using a $(2+\epsilon)$ capacity when each job only consists of one single task or there is only one server in the cluster. To the best of our knowledge, this is the first competitive result achieved for multi-resource job scheduling with task cloning in the resource-augmentation setting [16]. We also implement DollyMP under the widely deployed resource management system-Hadoop YARN. Our implementation has made substantial modifications to the current version of YARN [44].

We deploy the DollyMP implementation in our private cluster which consists of more than 300 cores. The experimental results demonstrate cloning alone can reduce the job response time by nearly 15% when a few jobs are running in the cluster, i.e., the cluster is lightly loaded. For the heavily loaded case where a large number of jobs arrive over time, DollyMP can significantly reduce the average job flowtime by 50% compared to the Capacity Scheduler. When compared to the well-known multi-resource scheduler DRF [19] and Tetris [20], DollyMP reduces job flowtime by nearly 40%. In contrast to the state-of-the-art scheduler, i.e., Carbyne [21], DollyMP can still cut down the average job flowtime by nearly 25%. Our conducted trace-driven simulations also show when the cluster load is high, by designing efficient scheduling algorithms, cloning can help to reduce the job response time by nearly 10% with only consuming 2% of extra resources.

The rest of this paper is organized as follows. Section 3 describes the analytical model and Section 4 presents the key design ideas behind DollyMP scheduler. In Section 5, we illustrate the implementation details of DollyMP on the top of Hadoop YARN. We conduct experiments in Section 6 to evaluate the performance of DollyMP. Before concluding our work in Section 8, we review works that are related to redundant execution and job scheduling for computing clusters in Section 7.

2 MOTIVATION OF DOLLYMP SCHEDULER

To motivate the cloning mechanism design of DollyMP, we run a WordCount job with an input size of 4GB and repeat the job 8 times in a small private cluster consisting of 30 heterogeneous nodes including both powerful servers and normal computing nodes [1]. To eliminate the effect of the scheduling policy, each job is submitted to the cluster after the previous one finishes. The result is depicted in Fig. 1 and it shows that, the running times of the same job vary a lot under the Capacity Scheduler even though the MapReduce framework itself has adopted some speculative execution scheme to handle stragglers [44]. Similarly, the DollyMP scheduler with the cloning mechanism disabled, i.e., DollyMP^0 also performs quite poor and the average running time is close to the capacity scheduler. By contrast, DollyMP^1 and DollyMP^2 (DollyMP scheduler with one cloned copy and two cloned copies respectively) perform much more stable, especially in the last five runs. We also report the average running time of jobs under these schedulers. The results indicate DollyMP^2 can reduce the average running time by nearly 20%, when comparing to the capacity scheduler. Therefore, the improvement gain from cloning is significant. We observe two major issues behind this result. On the one hand, the server heterogeneity makes the execution times of tasks within the same job phase differ a lot. On the other hand, the background workload on the physical servers where the VM instances are located also changes over time. Due to this, resource contention can occur and thus lead to stragglers. In this example, the reason why speculative execution fails under the Capacity Scheduler is caused by the late launching of extra backup copies when a straggler is detected.

Furthermore, we need to design an efficient multi-resource scheduler to benefit the most from cloning. To demonstrate this argument, we construct a simple example illustrated in Fig. 2. There are three jobs with their normalized resource demand and task execution times shown in the table and one server with a normalized capacity of one. Under the Tetris Scheduler, Job 1 (in orange color) is scheduled first since it has the highest combination value ($a+\epsilon \cdot p$), where a denotes the alignment score (the inner product between the resource demand the server capacity) and p is the resource usage (the product of the processing time and resource demand). After Job 1

[1] Refer to Section 6.1 for a more detailed description.

finishes, Tetris then begins to execute Job 2 (in black color) and Job 3 (in gray color) simultaneously. Note that, one can also leverage cloning to speedup the processing of these two jobs since there are extra idle resources. In this case, we shall make one clone for Job 2 and Job 3 respectively, leading to a speedup from 8 seconds to 6 seconds for both jobs and the total job completion time becomes 42 seconds. However, an alternative scheme of first scheduling Job 2 and Job 3 along with their clones and then Scheduling Job 1 will result in a total completion time of 28 seconds. Even we do not make any clone under the second scheme, the resulting completion time is still 20% less than that under the Tetris scheduler with cloning (34 seconds v.s. 42 seconds). As a result, the scheduling policy has a heavy impact on cloning efficiency. We need to jointly combine cloning and job scheduling to yield better performance.

3 ANALYTICAL MODEL FOR DOLLYMP

In this section, we develop an analytical framework which is general enough to characterize the multi-resource scheduling problem accounting for stragglers in heterogeneous big data processing clusters.

Consider a time-slotted system and a computing cluster which consists of M servers where the servers are indexed from 1 to M. Server i has a capacity of C_i CPU cores and M_i GB memory. Job j arrives at the cluster at time a_j and the job arrival process, $(a_1, a_2, \cdots, a_N)$, is an arbitrary time sequence. Each job j is characterized by a DAG graph G_j which has multiple phases with dependency and the phase set is denoted by $\Phi_j = \{\phi_j^1, \phi_j^2, \cdots, \phi_j^{\pi_j}\}$ [3]. For each phase ϕ_j^k, we denote by $P(\phi_j^k)$ the set of its parent (upstream) phases and $\phi_j^{\pi_j}$ is the phase that shall be completed last. Phase $\phi_j^k \in \Phi_j$ consists of n_j^k tasks for all $k \in \{1, 2, \cdots, \pi_j\}$, and all these tasks can execute in parallel. However, for a given task in phase ϕ_j^k, it can only begin executing after all the tasks in its parent phases finish processing.

Moreover, each task of phase ϕ_j^k has a resource demand of c_j^k CPU cores and m_j^k GB memory. The execution time of tasks from ϕ_j^k is denoted by Θ_j^k, where Θ_j^k is a random variable with its mean $\mathbf{E}[\Theta_j^k]$ characterized by θ_j^k and standard derivation $\mathbf{SD}[\Theta_j^k]$ captured by σ_j^k. Both θ_j^k and σ_j^k are known when job j arrives.

The cluster manager may assign multiple clones on available servers once a task is scheduled. A copy that completes first among all the cloned ones will be used for subsequent computation. When r clones (copies or replicas) are launched for a task in phase ϕ_j^k, the execution time of this task under cloning (i.e., the fastest replica) is still a random variable denoted by $\Theta_j^k(r)$, whose mean is given by:

$$\mathbf{E}\Big[\Theta_j^k(r)\Big] = \theta_j^k / h_j^k(r), \tag{1}$$

where $h_j^k(\cdot)$ denotes a speedup function of phase ϕ_j^k. One important assumption we make on $h_j^k(\cdot)$ is that $h_j^k(x)$ is strictly increasing and concave with respect to $x \in \mathcal{N}^+$ for all j and k. For example, when Θ_j^k is Type-I Pareto distributed with parameter x_m and α_j^k [7], namely, the cumulative density function of Θ_j^k is given by:

$$\Pr\Big\{\Theta_j^k > x\Big\} = \Big(\frac{x_m}{x}\Big)^{\alpha_j^k}. \tag{2}$$

Then, $h_j^k(x)$ can be simply derived as:

$$h_j^k(x) = \frac{\alpha_j^k - \frac{1}{x}}{\alpha_j^k - 1} = 1 + \frac{1 - \frac{1}{x}}{\alpha_j^k - 1}. \tag{3}$$

By adopting a unified speedup function, we do not distinguish between server heterogeneity and other factors that lead to stragglers. This can ease the modeling and algorithm design.

The aim of DollyMP is to mitigate the impact of stragglers by carefully lunching clones. Let $x_i^{j,k,l}(t) \in \{0, 1\}$ be an indicator variable to denote whether the lth task in phase ϕ_j^k of job j is running on server i at time slot t. Following Eq. (1), the expected amount of work completed within time slot t for this task can be expressed as:

$$y^{j,k,l}(t) = h_j^k\Big(\sum_{i=1}^{M} x_i^{j,k,l}(t)\Big), \qquad \forall j, k, l, t. \tag{4}$$

In Eq. (4), we consider all clones of a task are launched at the same time. In this case, the completion time of a task is the minimum among the execution times of its copies. Nevertheless, it is still difficult to characterize the minimum of multiple random variables from a probabilistic perspective without knowing the complete distribution. Therefore, we take an alternative approach of adopting the speedup function to capture the task completion requirement. With the mean and variance, we derive the speedup function via fitting a Pareto distribution, which is widely implemented in existing straggler mitigation schemes, e.g., [7, 38] and also adopted in theoretical models, e.g., [9, 34].

3.1 Problem Formulation

First, the total amount of resource usage on server i should not exceed the capacity of the server:

$$\sum_{j,k,l} x_i^{j,k,l}(t) \cdot c_j^k \le C_i, \quad \sum_{j,k,l} x_i^{j,k,l}(t) \cdot m_j^k \le M_i, \quad \forall i, t. \tag{5}$$

Second, a job phase finishes when all of its tasks complete, let λ_j^k denote the finishing time of phase ϕ_j^k, it satisfies:

$$\sum_{t=a_j+1}^{\lambda_j^k} y^{j,k,l}(t) = \theta_j^k, \qquad \forall j, k, l, t. \tag{6}$$

Third, each task can not begin its execution until all tasks in its parent phases complete, thus, we have:

$$x_i^{j,k,l}(t) = 0, \quad \forall j, k, l \text{ and } t \le \lambda_j^\sigma \text{ if } \phi_j^\sigma \in P(\phi_j^k), \tag{7}$$

Last, the whole job completes only after its last phase finishes. As such, the job finish time f_j is characterized by:

$$f_j = \lambda_j^{\pi_j}, \quad \forall j. \tag{8}$$

The goal of DollyMP is to minimize the overall job completion time (flowtime), which is a common performance metric in distributed computing clusters [20]. This goal yields the following

optimization problem:

$$\min_{\boldsymbol{x}} \sum_{j=1}^{N}(f_j - a_j) \qquad \text{(OPT)}$$

such that Eqs. (4), (5), (6), (7), (8) are satisfied,

where $(f_j - a_j)$ is treated as the flowtime of job j. In the rest of the paper, we will use the two terms, i.e., completion time and flowtime interchangeably.

3.2 Hardness

The optimization program (OPT) above is essentially a difficult problem. Regardless of task cloning, the multi-resource packing problem alone, e.g., Eq.(5) is APX-Hard [20]. As an illustrative example, consider the case where there is only one machine and all jobs (each job only has one task) require the same processing time but different numbers of CPU cores and amount of memory, our problem is then equivalent to the classical 2D bin-packing problem aiming at minimizing the number of bins used, which is strongly NP-complete. Due to this, DollyMP resorts to the use of approximate solutions.

4 KEY DESIGNS BEHIND DOLLYMP

In this section, we design two heuristics to solve two critical problems, i.e., when cloning can help to improve the overall job performance and how to design efficient job scheduling algorithms for a better coordination with task cloning.

4.1 When cloning is helpful?

Cloning delays the scheduling of other jobs, which may lead to a negative impact on the overall job performance. In this part, we first design simple heuristics for task cloning under DollyMP via analyzing several scheduling cases.

Consider a case where N jobs enter the cluster at time zero and each job has one single task to execute and all jobs have the same speedup function. The total capacity of the cluster is normalized to 1. In particular, job j needs $\frac{1}{2^j}$ normalized CPU cores and $\frac{1}{2^j}$ normalized Memory. In addition, the expected execution time of all jobs is one. In this case, the total resource demand is $\sum_{j=1}^{N} \frac{1}{2^j} = 1 - \frac{1}{2^N}$. The first scheduling scheme is to schedule all jobs at time zero and make one cloned copy for Job N, the resultant job completion time in expectation is $\text{flow}_1 = N - 1 + \frac{1}{h(2)}$.

By contrast, the second scheme tries to clone as many copies as possible for each job. Specifically, it launches 2^j copies simultaneously for job j after job $(j+1)$ completes. In this case, the total expected job flowtime is $\text{flow}_2 = \sum_{j=1}^{N} \frac{j}{h(2^j)}$, since there is only one job being executed at any time slot and other jobs need to wait.

The third scheme makes two clones for each job and schedules the job with the smallest resource demand first. As a result, Job 2 to N are scheduled together before Job 1. In this scenario, the total expected job flowtime is at most $\text{flow}_3 = \frac{N+1}{h(2)}$.

The difference between flow_1 and flow_2 (flow_3) is :

$$\text{flow}_1 - \text{flow}_2 = \sum_{j=2}^{N} \frac{h(2^j) - j}{h(2^j)}, \quad \text{flow}_1 - \text{flow}_3 = N - 1 - \frac{N}{h(2)}.$$

When the task execution time follows a Pareto distribution and the speedup function is given by Eq. (3), $h_j(2^j) < j$ if $j \geq \frac{\alpha}{\alpha-1}$ and $h(2) > \frac{N}{N-1}$ when $N > 2\alpha - 1$. This leads to $\text{flow}_3 < \text{flow}_1 < \text{flow}_2$. One implication of this result is that, a small number of clones can help to reduce the overall job completion time even in an overloaded cluster and priority should be given to small jobs. As such, DollyMP chooses to schedule extra cloned copies for small jobs when the total amount of consumed resources under cloning is less than the resource demand of other jobs.

4.2 Maximizing resource packing or minimizing waiting time?

Another key issue towards designing an efficient job scheduler is to balance the trade-off between maximizing the resource packing efficiency and minimizing the job waiting time. Optimizing the former performance objective leads to better resource utilization and requires one to give scheduling priority to jobs that can be tightly packed on available servers. However, it is usually better to schedule these small jobs first if one wants to reduce the waiting time of other jobs. To address this trade-off and better illustrate the major scheduling logic of DollyMP, we consider a transient case where the arrival time of all N jobs are zero and there is only one server in the cluster. In this transient setting, Im *et al.* design several priority-based algorithms when resource is one-dimensional and there is no task cloning [26]. Priority-based algorithms refer to those that prioritize jobs based on specified quantities, typical ones include:

- Shortest Remaining Processing Time (SRPT): jobs with the smallest running time are scheduled first. SRPT is optimal in the offline case where all the machines are identical and resource demands are homogeneous [17].
- Smallest Volume first (SVF): jobs with the smallest volumes are scheduled first where the volume is defined as the product of the job processing time and the job resource demand. SVF is a simple extension from SRPT to incorporate heterogeneous resource demands.

The above schemes are easy to implement, however, they can lead to quite poor performance. SRPT-based algorithm only prioritizes jobs according to the job processing time and can easily cause resource fragmentation, especially when the resource demand across jobs varies a lot [20]. By contrast, SVF aims at making a better balance between job processing time and resource demand. Nevertheless, SVF still does not perform well in a long run. Some jobs may require large resource demands and thus have low scheduling priority, however, cannot be built up in the queue for a long time. Otherwise, the system keeps scheduling low-demand jobs and can easily lead to low resource utilization.

With these limitations in mind, we combine SRPT and SVF together to yield a more efficient scheduling scheme. Specifically, we first modify the SVF scheme to include the multi-dimensional resource packing. To achieve this, we define the dominant resource of each job j as follows:

$$d_j = \max\left\{\frac{c_j}{\sum_{i=1}^{M} C_i}, \frac{m_j}{\sum_{i=1}^{M} M_i}\right\}, \qquad (9)$$

Algorithm 1: The transient scheduling algorithm.

1 **Procedure** Proc(Job set J, $\{v_j\}_j$, $\{\theta_j\}_j$);
2 Let $g = \log\left(\sum_{j\in J} v_j/(1-\max_j d_j)\right)$;
3 Let $p_j^l = \infty, \forall j \in J, 0 \le l \le g$;
4 **for** $l = 1, 2, \cdots, g$ **do**
5 Find $B_l = \left\{j \in J : \theta_j \le 2^l\right\}$;
6 Call the knapsack optimization oracle to solve:
$$\max_{x\in\{0,1\}} \sum_{j\in B_l} x_j, \quad s.t., \sum_{j\in B_l} v_j x_j \le 2^l.$$
7 **if** $x_j = 1$ *and* $p_j^{l-1} = \infty$ **then**
8 Let $p_j^l = l$;
9 **else**
10 $p_j^l = p_j^{l-1}$;
11 Let $\{\boldsymbol{p} : p_j = p_j^g\}$ Sort jobs in J in an increasing order of p_j and let $sch = 1$ Update remaining resources ;
12 **while** *resource is enough to launch job sch* **do**
13 Schedule job sch ;
14 **if** *job* $(sch+1)$ *cannot be scheduled* **then**
15 Make one extra clone to job sch;
16 $sch++$;

which is similar to that in Dominant Resource Fairness (DRF) scheme [19]. We next define the volume of each job as:

$$v_j = d_j \cdot \theta_j. \tag{10}$$

We then divide jobs into different categories based on the job processing time. Within each category, we try to pack as many jobs as possible via solving a knapsack optimization problem. The objective is to maximize the total number of packed jobs subject to, the total volumes of the packed jobs not exceeding a certain constant. In this case, the scheduling priority is still given to jobs with small processing time, however, all jobs within the same category are treated equally once they can be chosen by the optimization oracle. By doing this, we can achieve a high packing efficiency and in the meanwhile reduce the total job waiting time. In addition, cloning is made when there are extra available resources.

The transient scheduling process is shown in Algorithm 1. It first finds a set of jobs that can be completed within 2^l time slots. It then maximizes the number of jobs packed with the total volume not exceed 2^l. It is worth noting that, in Step 6 of Algorithm 1, the knapsack optimization oracle can be solved efficiently by selecting items with the smallest weights since the profits of all items are the same.

4.2.1 Complexity of the transient scheduling process. Since the system capacity is used by at least $(1-\max_j d_j)$ in any time except the final time slot under Algorithm 1, it manages to complete all jobs by time slot $g = \sum_j v_j/(1-\max_j d_j)$. The optimization oracle only sorts items based on weights and therefore has a time complexity of $O(n^2)$ where n is the number of items to be packed. Since there are g steps and there are at most N jobs to be selected in each step, the time complexity of the transient scheduling process is $O\left(N^2 \cdot \log\left(\sum_j v_j/(1-\max_j d_j)\right)\right)$.

4.2.2 Optimality. We show that the transient scheduling algorithm can achieve a bounded competitive performance.

Theorem 1. *When $h_j(x)$ is upper bounded by R for all j and $x \in \mathcal{N}$, then Algorithm 1 without cloning can achieve a competitive ratio of $6R$ with respect to the total job flowtime.*

Proof. Let $\mathcal{OPT}$ denote the optimal schedule and $N_{-1} = 0$. When $\mathcal{OPT}$ finishes N_l jobs by time slot 2^l, Algorithm 1 have completed at least N_l jobs by time slot $3R \cdot 2^l$, following the result of 2D-strip packing [40]. In this case, there are at most $(N - N_l)$ jobs that are still active in the cluster during the time interval $[3R\cdot 2^l, 3R\cdot 2^{l+1})$. Let W_l denote the amount of flowtime accumulated during this interval, W_l is upper bounded by:

$$W_l \le 3R \cdot 2^l \cdot (N - N_l). \tag{11}$$

As such, one upper bound for the total flowtime under Algorithm 1 is given by:

$$F^A = \sum_{l=0}^{g} W_l \le 3R \cdot \sum_{l=0}^{g} 2^l (N - N_l). \tag{12}$$

By contrast, these $(N - N_l)$ jobs need to wait for at least 2^{l-1} time slots under $\mathcal{OPT}$. And no jobs complete in the first time slot, thus, the total job flowtime under $\mathcal{OPT}$ is lower bounded by:

$$F^* \ge \sum_{l=0}^{g} 2^{l-1}(N - N_l) + N, \tag{13}$$

Combining Eq. (11) and Eq. (12), we have that $F^A \le 6R \cdot F^*$, this completes the proof of Theorem 1. □

The following corollary states that, Algorithm 1 can achieve a much better competitive performance if we schedule clones in a more careful manner. More specifically, let $r_j = \min\left\{r \in \mathcal{N} : 2^l h_j(r) \ge \theta_j\right\}$, then $(r_j - 1)$ clones will be made if the corresponding resources can be packed in Step 6.

Corollary 4.1. *There exists an offline scheduling algorithm that can achieve a competitive ratio of 6.*

5 DOLLYMP SCHEDULER FOR GENERAL DAG JOBS

In this section, we apply the key designs above to handle the scheduling of general DAG jobs. In these jobs, tasks that have a large variation in execution times in a phase can easily prolong other tasks in dependent phases and hence should be given lower scheduling priority. When taking this into account, DollyMP scheduler incorporates the standard deviation of task execution time by a factor of r to define the effective processing time of phase ϕ_j^k as $e_j^k = \theta_j^k + \sigma_j^k$.

In a DAG graph, each path contains a chain of sequential phases and the length of the path is the sum of the effective processing time of phases on the path. In particular, let L_j be the critical path,

i.e., the longest path in job j's DAG, we then define the effective volume and effective processing time of a DAG job j as follows:

$$v_j = \sum_{k=1}^{\pi_j} n_j^k \cdot e_j^k \cdot d_j^k, \quad e_j = \sum_{k:\phi_j^k \in L_j} e_j^k. \tag{14}$$

where d_j^k is the dominant resource of the tasks in phase ϕ_j^k of job j and is given by the following formula:

$$d_j^k = \max\left\{\frac{c_j^k}{\sum_{i=1}^M C_i}, \frac{m_j^k}{\sum_{i=1}^M M_i}\right\}. \tag{15}$$

With this newly defined effective job volume and job processing time for DAG jobs, we adopt the same scheduling logic as that in Algorithm 1 to determine the scheduling order for all jobs that have not finished in the cluster. Based on this order, DollyMP scheduler tries to schedule as many tasks as possible for jobs with a small order (high priority). For jobs with multiple phases, the tasks in the subsequent phase can be scheduled only after all the tasks in their parent phases have been completed. When the cluster can not schedule any new task (either due to all the tasks of a job have been scheduled or due to some of the dependent tasks have not finished yet), the scheduler begins to launch clones for the scheduled tasks if there still exist extra idle resources. When launching cloned copies, the scheduler also follows the same order as that for scheduling the normal tasks. In addition, the maximum number of clones for each running task is two under DollyMP, namely, there are at most three concurrent copies running for each task. There are two reasons leading to this design. On the one hand, the speedup function $h_j^k(x)$ is concave in x and thus a large number of clones does not improve much comparing to a small number. On the other hand, in the distributed file systems, each data block usually keeps two replicas for fault tolerance. As such, two clones can maintain a good data locality for reducing the time of fetching the input data.

Each job reports its effective volume and processing time to the job scheduler when it is newly submitted. The job scheduler inputs this information to the transient scheduling Algorithm, which in turn returns the scheduling orders of all input jobs. To reduce the overhead, the scheduling order of all jobs in the cluster won't be updated until the next job arrival. Once resources on a server become available, the scheduler then assigns tasks with the highest priority to this server if its resource demand can be satisfied. If the resource demand of the most preferable task exceeds the available capacity, the scheduler then turns to the next preferable task. When multiple jobs own the highest priority returned from the optimization oracle, DollyMP will choose a task from these jobs with the best resource fit. Similar to Tetris Scheduler, the resource fit is computed as the inner product between the resource demand vector of each task and the remaining resource capacity of the available server.

The corresponding pseudo-code is shown in Algorithm 2. It is worth noting that, in Step 3, the job scheduler needs to update the computation of job volume and processing time when a new job arrives to the cluster. Let $\Phi_j(t)$ denote the remaining phases of job j that have not been finished, the updated job volume is given by:

$$v_j(t) = \sum_{k:\phi_j^k \in \Phi_j(t)} n_j^k(t) \cdot e_j^k \cdot d_j^k, \tag{16}$$

Algorithm 2: The online scheduling process of the DollyMP Scheduler

1 **if** *A new job enters the Cluster in time t* **then**
2 Let $A_t = \{j : a_j \le t < c_j\}$;
3 For each $j \in A_t$, compute $v_j(t)$ and $e_j(t)$ following Eq. (16) and Eq. (17);
4 Call Proc$(A_t, \boldsymbol{v}_j(t), \boldsymbol{e}_j(t))$ in Algorithm 1 to obtain the job priority $\{p_j(t)\}$;
5 Let $A_t^s = sort(\{p_j(t)\})$ based on the increasing order;
6 **while** *There are available resources on a server i* **do**
7 Let R_i^c be the amount of available CPU Resource;
8 Let R_i^m be the amount of available Memory Resource;
9 **for** $l \in A_t^s$ **do**
10 Let $\Omega_t^l = \{j \in A_t : p_j(t) = l\}$;
11 **while** $R_i^c < \min_{j\in\Omega_t^l} c_j^k$ *or* $R_i^m < \min_{j\in\Omega_t^l} m_j^k$ **do**
12 Let $j^* = \arg\max_{j\in\Omega_t^l} R_i^c \cdot c_j^k + R_i^m \cdot m_j^k$;
13 **if** $c_j^k < R_i^c$ & $m_j^k < R_i^m$ **then**
14 Assign a task or its clone from phase ϕ_j^k of job j^* to server i;
15 $R_i^c = R_i^c - c_j^k$, $R_i^m = R_i^m - m_j^k$;
16 Repeat Step 9 twice if there are available resources.

where $n_j^k(t)$ is the number of not-finished tasks in phase ϕ_j^k of job j by time t. Let $L_j(t)$ be the critical path in the remaining phases of job j, correspondingly, the updated processing time of job j is defined as:

$$e_j(t) = \sum_{k:\phi_j^k \in L_j(t)} e_j^k. \tag{17}$$

Each task can maintain two extra clones at most, Step 9 to 15 of Algorithm 2 is repeated three times before the resources are used up in the cluster. Moreover, the scheduler needs to wait for available resources before it can make actual scheduling decisions (i.e., Step 11). For each job j, ϕ_j^k is the first available phase that can be scheduled at present. In addition, DollyMP follows the delay assignment policy in [5] to determine whether and when to kill other clones once the first copy of a task completes.

5.1 Optimality of DollyMP Scheduler

DollyMP Scheduler can manage to achieve a bounded competitive ratio in some special settings under resource augmentation [16]. In a capacity augmentation analysis, an algorithm is given a capacity greater than one while being compared to the optimal schedule on a unit-capacity server.

Theorem 2. *Algorithm 2 is $(2+\epsilon)$-capacity, $O(\frac{1}{\epsilon})$ competitive with respect to the total job flowtime, when there is only one server in the cluster or all the jobs have only one single task.*

Proof sketch. The key step is to formulate a simplified optimization problem whose cost is within a constant factor of the offline optimal cost. We adopt the online primal-dual fitting approach

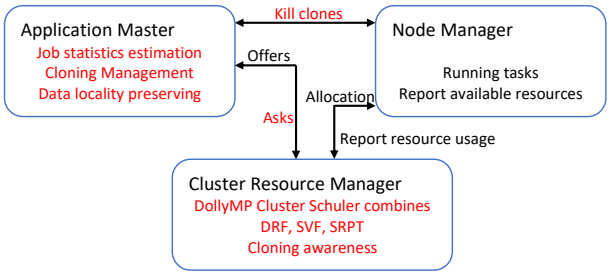


Figure 3: The system architecture of DollyMP built on the top of Hadoop YARN (shown in red).

via investigating this approximate problem, followed by constructing a pair of dual variables following the policy of Algorithm 2. This construction combines SVF and knapsack packing, which is completely different from traditional dual designs in the setting of homogeneous demands.

Discussion: when there is no straggling task in the cluster, i.e., the execution time of each task is a fixed constant, Algorithm 2 achieves a competitive ratio of $\left(\frac{3+3\epsilon}{\epsilon}\right)$ for minimizing the overall job flowtime when using a $(2+\epsilon)$-capacity with 1-speed in an online setting. By contrast, the HRDF policy presented in [16] achieves a competitive ratio of $\left(\frac{5+3\epsilon}{\epsilon}\right)$ using the same amount of capacity, in the single-server setting. In this sense, our designed online scheduler manages to achieve a much better competitive performance than existing schemes, especially when ϵ is small.

5.2 DollyMP Implementation under YARN

We implement DollyMP on top of Hadoop YARN. The system architecture is illustrated in Figure 3 where we highlight the modifications to Hadoop YARN in red colors.

We implement the scheduling algorithm in Section 5 under the Resource Manager of YARN (RM). The new scheduling logic combines DRF, SVF, and SRPT to recompute the priority of each job whenever a new Application Master is created. In addition, the Resource Manager knows the ID of each task so as to make cloned containers for each task.

Application Master (AM) estimates task demands as well as the statistics of task execution times from historical jobs and from earlier tasks in the same phase. First, recurring jobs are fairly common in big data processing clusters, we observe in the cluster that the jobs submitted from the same usually repeat the same computation. For such jobs, AM directly applies task statistics measured in prior runs of the job [20, 21, 28]. Second, the tasks from the same phase within a job have similar resource requirements and execution properties [6, 15, 43]. Hence, AM estimates the resource demands and execution times of a phase (mean and stand derivation) using the measured statistics from the first few tasks, and update it timely when more tasks finish. Third, when none of the above properties are satisfied. AM just uses the resource demand from the container request to perform the computation. For the estimation of task execution times, AM simply uses all the prior jobs from the same application framework to calculate both the mean and the stand derivation.

AM relies on the Resource Manager to report such information. Based on this information, Application Master computes the job volume along with the processing time, and sends them to the Resource Manager. Moreover, each Application is responsible for launching cloned copies for a task. When RM allocates more containers than the number of pending tasks, an AM will make a second-level scheduling decision to determine where to launch each task and its clones, based on the data locality constraint. Whenever a task or its cloned copy finishes, the corresponding AM keeps another running copy with the best data locality level and kills the remaining running copies on their corresponding Node Managers.

We also add the information of each task ID to a container request. With this ID, RM knows the data locality preferences of each task and thus launches cloned copies for the task to satisfy such preferences. Moreover, the container request also encodes information of the maximum number of clones to launch for each task and the default value is two.

For intermediate data transfer between two successive phases within a job, e.g., Map Phase and Reduce phase, DollyMP adopts the mechanism of delay assignment only when tasks from the downstream phase have also been scheduled clones [5]. Under this scenario, AM first waits to assign the outputs of two early upstream copies to each of the downstream clones evenly, and thereafter proceeds without waiting for the last clone if there are copies running for tasks in the upstream phase. In the case where the number of copies in the upstream phase is less than that in the subsequent phase, AM assigns the output from the copy that finishes first to all the copies of each downstream task.

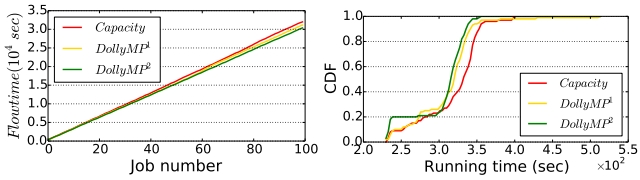
6 PERFORMANCE EVALUATION

We evaluate DollyMP using our prototype implementation on a private cluster with 30 heterogeneous nodes with a total of 328 cores. We also supplement the evaluation with trace-driven simulations.

6.1 Experiment Setup

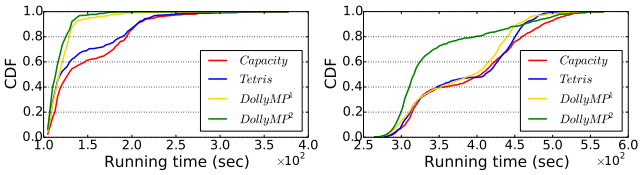
Cluster: The cluster consists of three different types of servers. There are two powerful servers each one with 24 CPU cores and 48GB memory, and seven normal servers each one with 16 CPU cores and 32-64GB memory. The remaining nodes are similar where each one has 8 CPU cores and 16GB of memory. All servers are placed within two racks and connected in a folded CLOS.

Baselines: We compare DollyMP with $\delta = 0.3, r = 1.5$ to the default scheduler of YARN, i.e., the Capacity Scheduler [2]. The Capacity Scheduler is also deployed in production clusters of Yahoo. In addition, we also compare DollyMP to the well-known scheduler, i.e., Tetris [20] and DRF [19]. Tetris combines the SRPT scheduler and heuristic algorithms for the multi-dimensional resource packing problem to compute a weighted score for each of the mapping pairs between the available server and unscheduled tasks. Then, Tetris assigns a task with the highest score to the available servers. DRF is a widely-adopted fair algorithm under which it offers resources to the job whose dominant resource's allocation is furthest from its fair share. Moreover, we also deploy three different versions of DollyMP according to the limit on the maximum number of cloned copies that can be launched for a task: no clone, one clone, and two clones. For ease of presentation, we let DollyMP^0, DollyMP^1, DollyMP^2 denote these versions respectively in the rest of this paper.



(a) Total job flowtime (b) Job execution times

Figure 4: Job flowtime and execution times under different schedulers in the lightly-loaded regime.



(a) Pagerank Application (b) WordCount Application

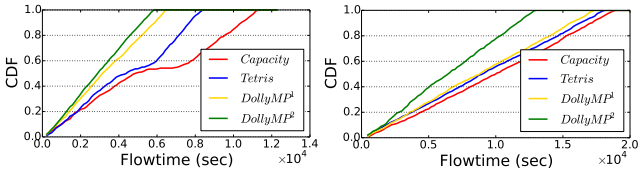
Figure 5: The running time CDF of different applications in the heavily-loaded regime.

Performance metrics: To compare the performance across different cluster schedulers, we study both the job flowtime and the actual job execution time (running time). We characterize the overall distribution as well as the sum of the job flowtime and execution times. We also investigate the resource consumption under cloning.

6.2 Deployment Results

Workload: To evaluate DollyMP, we constructed a workload suite of over 1000 jobs by picking uniformly at random from the Google traces [37]. The Google traces contain job size (the total number of tasks) and the resource demand for CPU cores and memory of each task. Based on the task number, we generate a fixed portion of map tasks and reduce tasks respectively. Our constructed workload includes two types of applications, i.e., PageRank and Wordcount. For PageRank jobs, half of them have an input data size of 10GB and another half has an input size of around 1GB. For Wordcount, all jobs have an input data size of 10GB.

6.2.1 Evaluation of cloning efficiency. To study the performance of DollyMP on different job types and job sizes in a lightly-loaded case, we select 100 jobs from our constructed workload suite where a half of the jobs run the Pagerank application and the other half run the Wordcount example. The inter-arrival time between these jobs is around 200 seconds. In this case, the job flowtime is very close to the job running time since only a few jobs need to wait for available resources when they enter the cluster. And Tetris performs quite similarly to Capacity scheduler. Fig. (4b) depicts the cumulative density function (CDF) of the job execution times under different schedulers. Observe from Fig. (4b) that, nearly 95% of jobs can complete within 350 seconds under DollyMP^2 while only 80% of jobs can achieve this under the Capacity scheduler. We also illustrate the overall job flowtime achieved under different schedulers in Fig. (4a). Still, DollyMP performs better than Capacity scheduler and it can reduce the average job flowtime by nearly 10% comparing to the latter. Interestingly, we also note that, DollyMP^2 outperforms DollyMP^1. This is because DollyMP^2 tends to launch more clones for a running task and therefore can better reduce the execution time of a job. In particular, when a job is small, there



(a) Pagerank Application (b) WordCount Application

Figure 6: The flowtime CDF of different applications in the heavily-loaded regime.

are enough resources to make clones in the cluster and therefore the performance improvement is even larger by launching more clones.

6.2.2 Evaluation of the scheduling policy. In this subsection, we aim to study the impact of the job scheduling policy from different schedulers on the overall system performance. As is known, Tetris combines SRPT and multi-resource packing together by simply computing the weighted sum of these two terms. By contrast, DollyMP makes a better trade-off between SVF and SRPT by solving a knapsack problem when a new job comes.

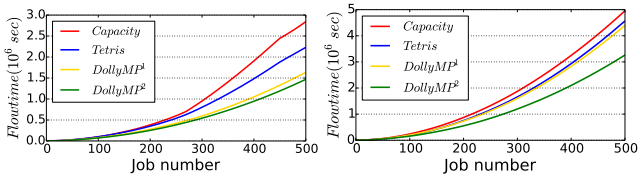
In the first experiment, we run 500 Pagerank jobs and the inter-arrival time between jobs is around 20 seconds. In this case, the jobs enter the cluster with a very high frequency, the system load is thus very heavy and there is only room to make clones for small jobs. We then run another 500 Wordcount jobs in the second experiment with similar inter-arrival times as the Pagerank experiment.

The job flowtime is usually much larger than the job running time in a heavily-loaded cluster as many jobs need to wait for a long time before being processed. As such, we evaluate both the job flowtime and the job running time in these two experiments. As illustrated in Fig. (5a), in the Pagerank experiment, the running times of jobs under DollyMP are much smaller than that under Tetris and Capacity scheduler. In particular, all the jobs can complete within 200 seconds after they are scheduled under DollyMP. However, only 80% of jobs can finish within 200 seconds under Tetris. Similar results could be observed from the Wordcount experiment in Fig. (5b). Under DollyMP, when a job is scheduled, most of the tasks within the same job phase can be executed simultaneously in the cluster, and therefore, the running time performs similar to that in the lightly-loaded regime.

When referring to the performance metric of job flowtime, DollyMP performs even better. As depicted in Fig. (6a) and Fig. (6b), mots jobs finish within 6000 seconds since their arrival under DollyMP. By contrast, only 60% (45%) of jobs can complete within 6000 seconds under Tetris (Capacity scheduler). We also illustrate in Fig. (7a) and Fig. (7b) the total job flowtime accumulated as jobs enter the cluster over time. The result indicates that, DollyMP can reduce the overall job flowtime by nearly 50% (30%) when comparing to the Capacity scheduler (Tetris).

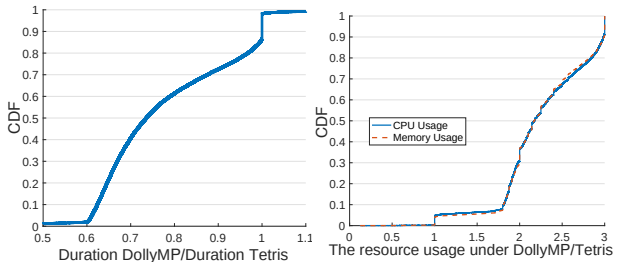
6.3 Trace-driven Simulations

We also build a trace-driven simulator to evaluate the performance of DollyMP in a large-scale computing cluster that consists of more than 30K heterogeneous servers. The simulator replays job traces



(a) Pagerank Application **(b) WordCount Application**

Figure 7: The overall job flowtime of different applications in the heavily-loaded regime.



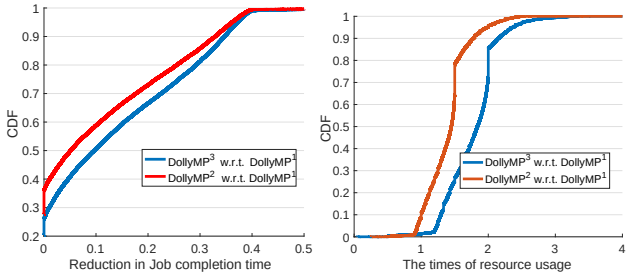
(a) Job speedup **(b) Resource usage**

Figure 8: CDF for the ratios of job duration and resource usage under DollyMP^2 to that under Tetris.

from Google Clusters [37]. The statistics show that the task execution times within the same job phase cancan vary substantially (the stragglers could be 20× slow as the normal tasks) and moreover, the percentage of job phases that contain stragglers is also very high (i.e., 70% of job phases contain a fraction of more than 15% task stragglers). When mimicking the execution of task clones, we set the running time of each clone to be the same as that of a task randomly chosen from the same job phase. Unless otherwise specified, the evaluation results use the default parameter value $r = 1.5$. In the simulations, we choose the scheduling interval (i.e., the length of the slot) to be 5 seconds, which is comparable to the duration of small tasks in traces. At the beginning of each interval, DollyMP shall check the amount of available resources in the cluster to make scheduling decisions.

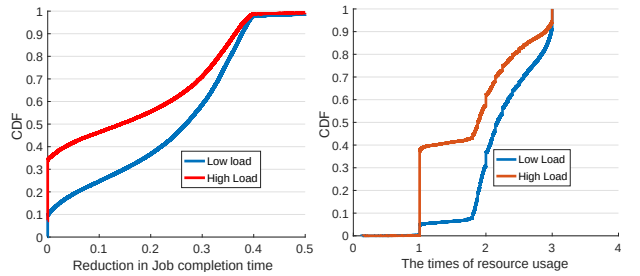
6.3.1 Job speedup v.s. extra resource usage. Cloning helps to speed up the task execution process and however, can incur extra resource consumption. To make a fair comparison, we quantify both the job speedup and the resource usage of cloning under DollyMP^2, Tetris, DRF respectively. Here, the resource usage is the sum across the (normalized) CPU and Memory resource multiplied by the task duration within a job. As shown in Fig. (8a), at least 40% of jobs obtain a reduction by 30% in job flowtime under DollyMP^2 compared to Tetris and the average speedup is 22%. Fig. (8b) shows that around 70% of jobs consume double amount of resources in both CPU and Memory dimensions under DollyMP^2 w.r.t. DRF. Since DollyMP prefers to make clones for small jobs, the overall resource consumption of DollyMP^2 is only 60% higher than that of DRF. In addition, DollyMP^2 also reduces the makespan (the longest job completion time) by 18%. As the cluster load is not high, the DRF scheduler performs similar to the Tetris scheduler and therefore we do not show the results of DRF in the figures.

What is the optimal number of clones for each task? To answer this question, we tune the number of clones from one to three and



(a) The CDF of job speedup **(b) Resource usage**

Figure 9: The job speedup and resource usage under different number of clones.



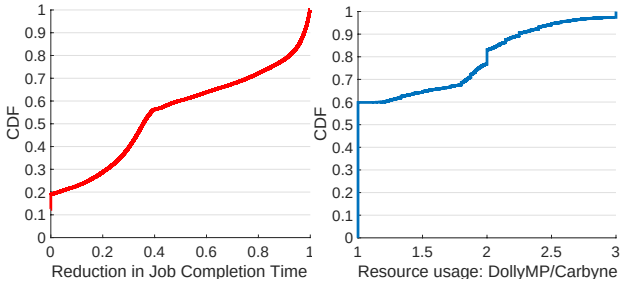
(a) Job speedup **(b) Resource usage**

Figure 10: DollyMP^2 v.s. DollyMP^0: the effect of cloning under different cluster loads.

investigate both the job speedup and the amount of extra resource usage. The results shown in Fig. 9 indicate that, increasing the number of clones from two to three does not help much. Comparing to DollyMP^1, DollyMP^2 helps more than 30% of jobs to reduce the job flow time by 20%. However, DollyMP^3 only leads to another 5% of jobs achieving the same level of reduction in job flowtime. Moreover, DollyMP^3 results in an amount of total resource usage which is 15% higher than that achieved by DollyMP^2.

We also evaluate the efficiency of cloning under different cluster loads. To achieve this, we fix the job workload while varying the number of CPU cores in the cluster. Intuitively, cloning should be helpful only when the cluster load is not high. However, Figure. (10a) demonstrates that cloning can still effectively reduce the job flowtime under high cluster loading (10× the low load). The overall job flowtime is reduced by 10% under cloning with 2% of extra resource consumption. In addition, more than 40% of jobs show a reduction in job flowtime by at least 20% under DollyMP^2 when comparing to the scheme with cloning disabled. The major reason is due to that the job scheduling policy can make a heavy impact on the cloning effect. Since the number of jobs staying in the cluster is not large under DollyMP even when the cluster load is high, there often exists room to make clones for small tasks running in the cluster. Figure. (10b) further implies that nearly 40% of tasks have cloned copies running in the cluster when the cluster load is high. In a conclusion, one can make the most use of cloning by carefully designing job scheduling policies.

6.3.2 Comparison with the state of the art. Finally, we compare our built scheduler to the state of the art, i.e., Carbyne [21] and Graphene [22]. The Carbyne Scheduler adopts ideas from DRF and



(a) Flowtime reduction **(b) Resource usage**

Figure 11: Comparison between DollyMP^2 and Carbyne when the cluster is heavily loaded.

Tetris, and applies altruistic scheduling to collect leftover resources. The leftover resources are then be redistributed to other tasks for achieving better job performance and cluster efficiency. By contrast, the strength of Graphene is to deal with jobs consisting of heterogeneous DAGs, and it performs similarly to Tetris for jobs with sequential dependencies. As such, we only depict the comparison results between DollyMP^2 and Carbyne.

We compare the performance of DollyMP^2 to Carbyne under in a heavily-load cluster. As shown in Fig. (11a), nearly 30% of jobs achieve a reduction in job completion time by more than 80%. In the meanwhile, around 60% of jobs consume the same amount of resources under these two schedulers (Fig. (11b)). Moreover, DollyMP^2 reduces the average job completion time by 25% comparing to Carbyne. Even though Carbyne makes use of the redistribution of leftover resources to optimize the performance of jobs with DAG graphs, DollyMP^2 can still achieve a significant improvement by cloning small jobs.

6.3.3 Scheduling overhead. The overall scheduling overhead of DollyMP is very low. Specifically, the scheduler takes less than 20ms to make scheduling decisions for all jobs in our private cluster. When referring to scheduling costs in a large-scale cluster, the simulation result shows that scheduling 1K jobs to 30K machines costs less than 50ms on a 3.3 GHz 6-Core Intel Core i5 processor.

7 RELATED WORK

In the literature, there have been several research efforts to design speculative execution and cloning schemes for MapReduce Systems [1, 5, 7, 8, 13, 14, 27, 41, 46, 51]. While some of these approaches are implemented in practical systems and demonstrated to be efficient for specific job types, they mainly deal with traditional MapReduce jobs which only require slots to be statically configured on each machine. Another major limitation is that, these schemes do not jointly design job scheduling with redundant execution, making redundancy not as efficient as expected. To overcome these drawbacks, Ren *et al.* propose Hopper, a speculation-aware scheduler, which coordinates job scheduling with speculative execution [38]. However, Hopper still has several downsides that can degrade the cluster performance. Hopper is non-work-conserving: it is possible to keep a computing slot idle as a reservation for a future straggler while other jobs/ tasks already queue up for computation resources.

Recently, Xu *et al.* present Chronos to bring several different speculative scheduling strategies together under a unifying optimization framework [48]. Chronos defines a novel metric, i.e., Probability of Completion before Deadlines (PoCD), to compute the probability that a job meets its desired deadline, under cloning and speculative execution respectively. Based on this metric, Chronos proposes to solve an optimization problem for jointly optimizing PoCD and the execution cost in different strategies. While such optimizations can lead to utility increase and cost improvements, they fail to deal with multiple jobs that have multi-resource requirements and complicated task dependencies. In addition, Zhou *et al.* build an energy consumption model to characterize the energy efficiency for different speculative execution solutions under MapReduce systems [53]. Relying on the theoretical analysis, the authors then propose a window-based dynamic resource reservation and a heterogeneity-aware copy allocation technique to further optimize the job performance and energy consumption.

To efficiently allocate resources in production clusters, several schedulers have been proposed from both production clusters [10, 24, 29, 35, 39, 44, 45] and the academia, see [11, 12, 18, 20–22, 30–32, 42, 43, 47, 49, 52] for example. In particular, schedulers in [11, 12, 30, 32, 42, 49, 52] are designed under the MapReduce system and only consider one-dimensional resource, i.e., CPU. In contrast, all the other schedulers consider multi-dimensional resources for the task requirement. Moreover, schedulers in [11, 12, 20, 30, 32, 42, 44, 49, 52] are centralized where the resource manager is responsible for scheduling all applications/ jobs in a cluster. Mesos adopts the two-level scheduling paradigm under which resources are first allocated to each framework by the master and then the framework will make second-level scheduling decisions [24]. By contrast, schedulers in [10, 29, 35, 39, 45] are fully distributed, each scheduler makes the scheduling decision independently and only one commitment will succeed if there are any conflicts.

8 CONCLUSIONS AND FUTURE WORKS

This paper makes an attempt to mitigate stragglers by making task clones for big data processing systems with multi-resource requirements. Our primary goal and contribution are to build the first analytical model accounting for multi-resource scheduling under task cloning for jobs with DAG dependencies. We conducted a simple analysis to study when cloning is helpful. Such analysis could enable us to design efficient online scheduling algorithms. Another contribution of this paper is to bound the competitive performance of a multi-resource scheduling algorithm with cloning. As future works, we plan to apply online learning methods to quickly identify those servers that can easily lead to stragglers.

ACKNOWLEDGMENTS

This work is supported in part by the Start-up Research Grant of University of Macau (SRG2021-00004-FST), CUHK Direct Grant #4055108, and in part by the National Natural Science Foundation of China under Grant Number 61802060.

REFERENCES

[1] 2013. Apache. http://hadoop.apache.org.
[2] 2015. Hadoop: Capacity Scheduler. https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[3] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. 2016. Scheduling Parallel DAG Jobs Online to Minimize Average Flow. In *Proceedings of SODA*.

[4] Sultan Alamro, Maotong Xu, Tian Lan, and Suresh Subramaniam. 2020. Shed+: Optimal Dynamic Speculation to Meet Application Deadlines in Cloudy. *IEEE Transactions on Network and Service Management* 17, 3 (2020).

[5] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of NSDI*.

[6] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. 2012. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of NSDI*.

[7] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. 2014. GRASS: Trimming Stragglers in Approximation Analytics. In *Proceeding of NSDI*.

[8] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in MapReduce Clusters using Mantri. In *Proceedings of USENIX OSDI*.

[9] ELENE ANTON, URTZI AYESTA, MATTHIEU JONCKHEERE, and INA MARIA VERLOOP. 2020. Improving the Performance of Heterogeneous Data Centers through Redundancy. In *Proceedings of ACM Meas. Anal. Comput. Syst.*

[10] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceeding of OSDI*.

[11] H. Chang, M. Kodialam, R. R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee. 2011. Scheduling in MapReduce-like systems for fast completion time. In *Proceedings of IEEE Infocom*.

[12] Fangfei Chen, Murali Kodialam, and T. V. Lakshman. 2012. Joint scheduling of processing and shuffle phases in MapReduce systems. In *Proceedings of IEEE Infocom*.

[13] Qi Chen, Cheng Liu, and Zhen Xiao. 2013. Improving MapReduce Performance Using Smart Speculative Execution Strategy. *IEEE Trans. Comput.* PP, 99 (January 2013).

[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI*.

[15] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of Eurosys*.

[16] Kyle Fox and Madhukar Korupolu. 2013. Weighted Flowtime on Capacitated Machines. In *Proceedings of SODA*.

[17] Kyle Fox and Benjiamin Moseley. 2011. Online Scheduling on Identical Machines using SRPT. In *Proceeding of SODA*.

[18] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of EuroSys*.

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of NSDI*.

[20] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of Sigcomm*.

[21] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *Proceedings of OSDI*.

[22] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proceedings of OSDI*.

[23] Robert Grandl, Arjun Singhvi, Raajay Viswanathan, and Aditya Akella. 2021. Whiz: Data-Driven Analytics Execution. In *Proceedings of NSDI*.

[24] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI*.

[25] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. 2018. Wide-Area Analytics with Multiple Resources. In *Proceedings of EuroSys*.

[26] Sungjin Im, Mina Naghshnejad, and Mukesh Singhal. 2016. Scheduling Jobs with Non-uniform Demands on Multiple Servers without Interruption. In *Proceedings of Infocom*.

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceeding of Eurosys*.

[28] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of OSDI*.

[29] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of ATC*.

[30] Minghong Lin, Li Zhang, Adam Wiermana, and Jian Tan. 2013. Joint optimization of overlapping phases in MapReduce. In *Proceedings of IFIP Performance*.

[31] Yang Liu, Huanle Xu, and Wing Cheong Lau. 2019. Online Job Scheduling with Resource Packing on a Cluster of Heterogeneous Servers. In *Proceedings of Infocom*.

[32] Benjamin Moseley, Anirban Dasgupta, Santa Clara, Ravi Kumar, Santa Clara, and Tamás Sarlós. 2011. On scheduling in Map-Reduce and flow-shops. In *Proceedings of SPAA*.

[33] Tommi Nylander, Johan Ruuskanen, Karl-Erik Årzén, and Martina Maggio. 2020. Modeling of Request Cloning in Cloud Server Systems using Processor Sharing. In *Proceedings of ICPE*.

[34] Tommi Nylander, Johan Ruuskanen, Karl-Erik Årzén, and Martina Maggio. 2020. Towards Performance Modeling of Speculative Execution for Cloud Applications. In *Proceedings of ICPE*.

[35] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of SOSP*.

[36] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SoCC*. 1–13.

[37] C. Reiss, J. Wilkes, and J. L. Hellerstein. 2011. Google Cluster-Usage Traces. https://github.com/google/cluster-data.

[38] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. 2015. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In *Proceedings of Sigcomm*.

[39] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys*.

[40] A. Steinberg. 1997. A strip-packing algorithm with absolute performance bound 2. *SIAM J. Comput.* (1997).

[41] Xiaoyu Sun, Chen He, and Ying Lu. 2012. ESAMR: An Enhanced Self-Adaptive MapReduce Scheduling Algorithm. In *the 18th International Conference on Parallel and Distributed Systems (ICPADS)*.

[42] Jian Tan, Xiaoqiao Meng, and Li Zhang. 2012. Delay Tails in MapReduce Scheduling. In *Proceedings of SIGMETRICS*.

[43] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of EuroSys*.

[44] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, and M Sharad Agarwal. 2013. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of SoCC*.

[45] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of EuroSys*.

[46] Huanle Xu and Wing Cheong Lau. 2015. Optimization for Speculative Execution in a MapReduce-like Cluster. In *Proceedings of Infocom*.

[47] Huanle Xu, Yang Liu, and Wing Cheong Lau. 2021. Optimal Job Scheduling with Resource Packing for Heterogeneous Servers. *IEEE/ACM Transactions on Networking* 29, 4 (2021).

[48] Maotong Xu, Sultan Alamro, Tian Lan, and Suresh Subramaniam. 2018. Chronos: A Unifying Optimization Framework for Speculative Execution of Deadline-Critical MapReduce Job. In *Proceedings of ICDCS*.

[49] Yi Yuan, Dan Wang, and Jiangchuan Liu. 2014. Joint Scheduling of MapReduce Jobs with Servers: Performance Bounds and Experiments. In *Proceedings of IEEE Infocom*.

[50] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of HotCloud*.

[51] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving Mapreduce performance in heterogeneous environments. In *Proceedings of USENIX OSDI*.

[52] Yousi Zheng, Ness Shroff, and Prasun Sinha. 2013. A New Analytical Technique for Designing Provably Efficient MapReduce Schedulers. In *Proceedings of IEEE Infocom*.

[53] Amelie Chi Zhou, Tien-Dat Phan Inria, Shadi Ibrahim Inria, and Bingsheng He. 2018. Energy-Efficient Speculative Execution using Advanced Reservation for Heterogeneous Clusters. In *Proceedings of ICPP*.