Forseti: Dynamic Chunk-level Reshaping for Data Processing on Heterogeneous Clusters

Sultan Alamro[a,b], Tian Lan[b], Suresh Subramaniam[b]

[a] *Department of Electrical Engineering, College of Engineering, Qassim University, Buraidah 51452, Saudi Arabia*
[b] *Department of Electrical and Computer Engineering, the George Washington University, Washington, DC, 20052 USA*

Abstract

Data-intensive computing frameworks typically split job workload into fixed-size chunks, allowing them to be processed as parallel tasks on distributed machines. Ideally, when the machines are homogeneous and have identical speed, chunks of equal size would finish processing at the same time. However, such determinism in processing time cannot be guaranteed in practice. Diverging processing times can result from various sources such as system dynamics, machine heterogeneity, and variable network conditions. Such variation, together with dynamics and uncertainty during task processing, can lead to significant performance degradation at job level, due to long tails in job completion time resulted from residual chunk workload and stragglers.

In this paper, we propose *Forseti*, a novel processing scheme that is able to reshape data chunk size on the fly with respect to heterogeneous machines and a dynamic execution environment. *Forseti* mitigates residual workload and stragglers to achieve significant improvement in performance. We note that *Forseti* is a fully online scheme and does not require any *a priori* knowledge of the machine configuration nor job statistics. Instead, it infers such information and adjusts data chunk sizes at runtime, making the solution robust even in environments with high volatility. In its implementation, Forseti also exploits

Email addresses: alamro@qec.edu.sa (Sultan Alamro), tlan@gwu.edu (Tian Lan), suresh@gwu.edu (Suresh Subramaniam)

a virtual machine reuse feature to avoid task start-up and initialization cost associated with launching new tasks. We prototype Forseti on a real-world cluster and evaluate its performance using several realistic benchmarks. The results show that Forseti outperforms a number of baselines, including default Hadoop by up to 68% and SkewTune by up to 50% in terms of average job completion time.

1. INTRODUCTION

Data-intensive computing frameworks (DISCs) have become the *de facto* standard for large-scale computing applications like web indexing and data mining, which often need to process up to petabytes of data on a daily basis. To enable distributed computing, these frameworks typically split job data into fixed size chunks and process them by parallel tasks on distributed machines that involve commodity hardware/software. Ideally, in a homogeneous environment with identical-speed machines and equal-size chunks, the chunk processing intervals would be perfectly aligned with each other, eliminating any possibility of residual workload and stragglers[1] during job executions. However, such an ideal homogeneous environment is not feasible in practice. It has been shown that the divergence and uncertainty in task processing times resulting from machine heterogeneity and execution dynamics could lead to significant performance degradation of up to 75% [1] due to residual workload and stragglers.

This paper proposes a novel processing scheme called *Forseti*, which has the ability to reshape data chunk size processed by heterogeneous machines on the fly and to dynamically balance the workload assigned to parallel processing tasks. It effectively mitigates residual workload and stragglers during job execution, and as a result, leads to substantial job-level performance improvement, e.g., in terms of job average completion times and completion time tails. We note that the performance loss stemming from machine heterogeneity

[1]Stragglers refer to tasks that are running slow and behind the progress of average task executions.

and execution dynamics has been identified by many reserchers [45, 1]. While existing work mainly focus on either optimizing DISC cluster configurations based on the specific applications and infrastructure available [20, 21, 34, 6] or mitigating the negative effect of stragglers through task scheduling and placement [4, 3, 43, 45, 9], *Forseti* advocates an alternative approach to reshape data chunk size and re-balance task workload in a dynamic, online fashion throughout job processing. This equips the system with the ability to automatically adapt its execution and workload partitioning in *any* heterogeneous, uncertain execution environment.

Taming residual workload and stragglers is a crucial task for any computing performance optimization. In practical DISC clusters, heterogeneity can be caused by a number of reasons. First, the links within data centers suffer from congestion that could last up to several hundreds of seconds [24, 38]. This congestion makes tasks run slow (i.e., straggle) as their execution time and progress fall behind the average execution time of other tasks. Second, cloud providers use virtualization process to provide isolation among jobs and tasks running simultaneously on the same machines. However, practical task scheduling and isolation mechanisms either require precise job processing models or are too coarse-grained. Third, computing nodes are typically composed of commodity parts, thereby becoming dissimilar in processing speed. Last, heterogeneity in execution time can occur due to load imbalance assigned or created by different tasks [25, 26]. Thus, the divergence in task execution speed on heterogeneous machines is considered as the main issue that leads to excessive worker idleness (and thus resource under-utilization) in the cluster along with the creation of stragglers. To better understand the issue of heterogeneity and its impact on detecting stragglers on real systems, consider for instance the performance of a map-reduce job in Figure 1(a). The figure shows the execution time of map and reduce tasks of Hadoop for a WordCount benchmark running on a heterogeneous cluster. In this experiment, we set the level of heterogeneity in the cluster to 1-2-3 ratio (i.e., CPU speeds are 1x, 2x and 3x of a base speed). Initially, each map task is assigned 128MB of data to process. It can be clearly seen that

Hadoop creates discrepancy in performance among running tasks. Hadoop fails to adapt to the heterogeneity, even if speculation mechanism is enabled.

To reshape data chunk size and re-balance task workload, Forseti adapts to the divergence in task execution times and dynamically redistributes workload through an efficient pointer-rebalancing mechanism according to the underlying nodes' processing speeds. Forseti estimates the progress rate of different tasks/chunks, obtain a prediction of task completion times, and redistributes workload accordingly to minimize any potential residual work or stragglers. Forseti aims to reassign unprocessed data to machines so that the completion time of a job is minimized. As a result, this mechanism also ensures that the overall energy consumption/cost is minimized. Figure 1(b) shows how Forseti is able to reduce the overall completion time by 56% compared with Hadoop. We emphasize that Forseti does not require any *a priori* knowledge of the machine configuration nor job statistics. Instead, it infers such information on the fly and adjusts data chunk sizes at runtime, making the solution robust even in environments with high volatility.

While Forseti works with any distributed data processing framework, for the purpose of evaluating its performance, we implement a prototype of Forseti on Hadoop map-reduce framework. The reshaping algorithm is implemented in the *master*, which monitors the progress rate of all tasks of a job, estimates the completion time, and redistributes the remaining workload accordingly. In order to minimize overall network overhead, Forseti checks for data locally before fetching data from remote nodes. We note that even when data is not local, we found in our experiments that the benefit of redistributing data outweighs any overhead caused by fetching non-local data [32]. Forseti exploits "JVM reuse" to recycle virtual machine container of completed tasks without termination and re-launching [30, 17, 33, 19]. It effectively eliminates the JVM launching time overhead for new tasks. Moreover, Forseti does not need to know about the cause of divergence and uncertainty in execution time of tasks nor the exact job processing model. In addition, Forseti is designed to be transparent to the task function and requires no modification to the function design. Evaluating

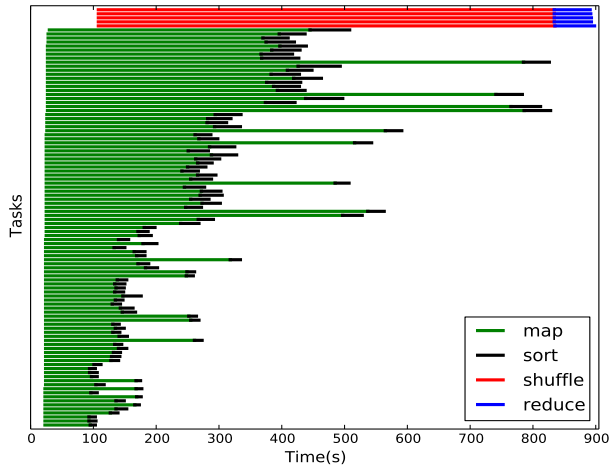
Forseti on real-world benchmarks, our results show that Forseti can significantly reduce job execution time by up to 68% on average compared to default Hadoop and 50% to SkewTune [27], a popular data rebalancing scheme. Moreover, the results show that Forseti can exploit the divergence and dynamics in progress rate among tasks and redistribute unprocessed workload efficiently. The findings substantiate our assertion that dynamic task/chunk reshaping mitigates the discrepancy in progress rate and minimizes the completion time of a job.

The rest of this paper is organized as follows. Section 2 presents related work, and Section 3 presents background and our motivation. The design of Forseti is presented in Section 4, and the algorithm's implementation is described in Section 5. Experimental results are presented in Section 6, and finally the paper is concluded in Section 7.

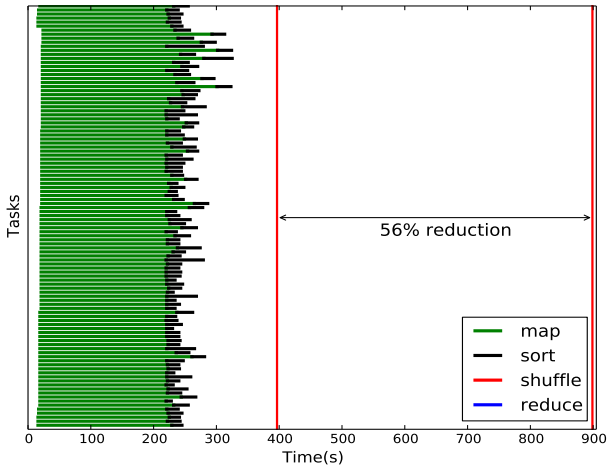
2. Background and Related Work

DISC frameworks, such as Hadoop (open source of MapReduce) and Spark, have been widely employed in production systems. Such frameworks process large datasets (e.g., terabytes or petabytes of data) across huge clusters (e.g., hundreds or thousands of nodes). The massive data are divided into and configured as fixed size chunks/blocks and then stored within an underlying distributed file system so as to support simultaneous processing of computation tasks across heterogeneous machines and clusters on the cloud. Generally, the execution flow is processed in a multi-stage/phase fashion by using the output from one phase as the input to another phase. A phase is considered completed when tasks of the phase finish processing. Therefore, a slowdown in one phase due to some tasks running slow can lead to a late start of next phase.

The issue of heterogeneity and the way it creates stragglers have been extensively studied in the context of DISC frameworks. LATE [45] suggested that MapReduce has many limitations under heterogeneous environments, mainly because straggler identification mechanisms that are in-built struggle to function properly within an environment that is heterogeneous. To address this challenge, better priority, scheduling, and identification techniques have been



(a)



(b)

Figure 1: Timing flow of a WordCount job running *map* and *reduce* tasks: (a) Hadoop (b) Forseti.

proposed. For instance, [5, 37, 9, 8, 41, 22, 3, 4, 43, 42, 44] attempt to mitigate stragglers and enhance the speculation mechanism of default Hadoop. They have proposed novel strategies to track stragglers, launch speculative tasks reactively and proactively. Another study [1] found that the use of remote map tasks increases network traffic when applied on fast machines significantly. However, one challenge to this technique is that the increased network traffic could end up competing with shuffle between phases, a factor that causes deterioration in performance. They overcame this issue through undertaking communication-aware load balancing as it helps keep away from busty network traffic. This process is enhanced further by [18] through undertaking fresh key partitioning schemes that have been established to improve Hadoop's performance with heterogeneous clusters. A node-capability-aware data placement model was developed that distributes data among nodes according to their processing capabilities [36]. The issue of data skewness among tasks due to data placement in a heterogeneous cluster has been addressed in [27, 25, 28, 14, 35, 18, 11].

Going by most research findings, data skewness can be mitigated. Here, skewness refers to the imbalance of computational nodes and datasets among tasks. Various researchers [9, 26, 2] studied and analyzed different skews that appear in different types of applications. SkewTune [27] proposed a strategy which balances data distribution across different nodes. SkewTune repartitions stragglers' data to capitalize on the idle task which just finished processing. In contrast to SkewTune, Forseti redistributes data assigned to *all* tasks on previous rounds upon a new task completion. FlexMap [14] tackled the heterogeneity issue and proposed a scheme to create map tasks with small block size and increase the sizes according to node's capabilities. The system initially launches a large number of maps with a small block size. However, this creates significant scheduling and starting overhead on the scheduler as well as resource contention. Moreover, it does not assume a shared cluster and fails to consider the JVM launching time overhead. Forseti aims to bypass the JVM launching overhead and follows the policies imposed by the master.

There are other papers too which focus on skewness. [7] proposed a frame-

work that reproduces blocks according to their popularity. It aims to minimize interference on any running jobs that have been co-hosted under a similar cluster with an accurate prediction of file popularity. Another work [16] proposed a task progress indicator in order to deal with data skewness. [46, 40, 15] proposed techniques to improve the performance of DISC frameworks and jointly optimize performance and cost within heterogeneous cloud environments. [39, 29] propose a dynamic data placement scheme for a heterogeneous cluster. However, such scheme requires *a priori* knowledge about the capability of the cluster. [35] proposed the concept of a virtual split, wherein its size changes (by adding more splits) as the mapper runs. Nonetheless, unlike Forseti, the assigned splits are never reassigned to other maps.

In multi-tenant data centers, resource sharing has become vital. Various studies have addressed the issue of unpredictable application performance in shared clusters [10, 23, 13]. The lack of performance isolation among users and applications leads to volatile application performance. The absence of proper isolation causes the task executions of DISC jobs to be stochastic. The uncertainty in their execution times affects the ability of straggler identification mechanisms, and makes their decision to speculate (or not to speculate) a straggling task very challenging. Thus, discrepancy in performance is the norm of shared resources rather than the exception.

3. Motivations and Problem Statement

In this section, we start by briefly introducing the fundamentals of DISC frameworks. We also discuss the ways through which the performance in heterogeneous settings gets severely affected by having routine parallelization in homogeneous clusters. Further, we show that the heavy-tailed behavior in the runtime distribution and large variation of execution times among tasks can be solved through an efficient dynamic load balancing.

DISC is the default for many data processing systems. Its implementation can be better understood from two specific phases, namely *map* and *reduce*. In this context, input data and a record of transitional key or value pairs are

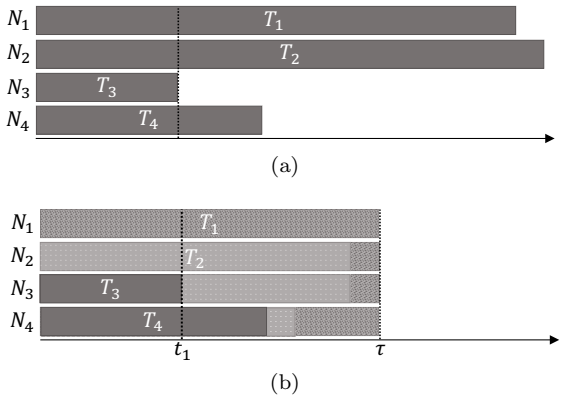


Figure 2: An illustrative example of the impact of a heterogeneous cluster on job completion time: (a) Default Hadoop. (b) Forseti.

formed via the map task. Every map task accesses and processes one split/chunk from a Distributed File System (DFS). On the other hand, these transitional key/value pairs are accumulated collectively and thereby passed to the reduce task through a communication stage named *shuffle*. A *master* monitors the progress of every task, and reports to the user about the job completion.

Case study. To demonstrate the problem considered in this paper, we perform a case study on Hadoop map-reduce framework. Consider for instance a DISC job in a heterogeneous environment with four unrelated tasks, i.e., $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$, which are running in parallel on four different nodes (N_1, N_2, N_3, N_4). It can be seen (see Figure 2(a)) that T_1 and T_2 took a long time compared with T_3 and T_4. The job cannot be considered complete until the processing of T_2 finishes. Moreover, N_3 finished first and stayed idle for the rest of the time. This indicates the inefficiency of the entire process. Figure 2(b) illustrates how Forseti re-balances the remaining workload among running nodes. Upon the completion of T_3, i.e., $\mathcal{T}_f = \{T_3\}$, at t_1, Forseti checks the progress rate of all running tasks (i.e., T_1, T_2, T_4) and redistributes the remaining data, i.e., D_1^r, D_2^r, and D_4^r, among the four tasks so they all finish at the same time (τ). The figure shows that T_4 can finish its remaining workload and process more data before τ. Thus, only data belonging to T_1 and T_2 are redistributed. This process is repeated until the completion of all tasks. This

case can be extended to consider a multi-phase framework. However, Figure 1 shows that the map phase can take up to 75% of the whole job completion time. Further, the figure shows that the shuffling starts right after a few map tasks complete. Nonetheless, the actual processing of reduce tasks is delayed until the last map task commits its output.[2] Thus, since the execution of reduce tasks takes only about 5% of job execution time, Forseti is designed to optimize and re-balance workload map tasks only, which leads to overall reduction in job execution time. Moreover, Forseti assumes all tasks are independent and have no precedence among them, which is typical for map tasks.

In DISC frameworks, the presence of homogeneous task model cannot fulfill the load balancing obligations and thereby maintain an effective heterogeneous setting. Furthermore, the model is also incapable of adapting to the fluctuating performance due to shared resources. The divergence and uncertainty is a problem for performance optimization and scheduling. This is because they make it almost impossible to obtain a precise model of task processing times. Additionally, tasks are generally regarded as the procedure of collecting records through serial key-value pairs. Nevertheless, as per the availability of any sort of application, such records might necessitate CPU as well as memory for processing valuable data based on the runtime of the DISC cluster. The key is to quickly and accurately estimate the completion time of running tasks based on their progress rate, and redistribute load swiftly. *Forseti* is built to develop straggler and skew mitigation through an efficient load balancing scheme that *dynamically* rebalances workload among running tasks. The objective is to reassign unprocessed data to machines so that makespan of a job (i.e., the time to complete all the tasks of a job) is minimized. The new load assignment aims to reallocate workload to machines according to their capabilities. Unlike [14, 27], Forseti reallocates and rearranges data assigned to all tasks on previous rounds upon a new task completion. Failure to do so can lead to a significant degradation and violate the service level agreement (SLA) between users and cloud

[2]Similar results are reported in [28, 27].

operators.

We now formally state the problem of minimizing the completion time of a job: Given a job (or a set of jobs) with a set of tasks $\mathcal{T}$, our goal is to design a processing scheme that is capable of reshaping data chunk sizes assigned to each task on the fly with respect to cluster heterogeneity in order to minimize the job completion time. When a task finishes processing, the scheme takes all tasks' associated unprocessed workload D_n^r, $n \in \mathcal{T}$, and redistributes it proportionally based on the measured process rate R_n among all tasks. The goal is to balance the residual workload among the processing nodes and let all tasks finish at the same time with the new assignment. We present the details of the design of Forseti in the next section, and then show the significant improvement in computing performance that it achieves.

4. Forseti Design

In this section, we present the design of Forseti and its applicability to any DISC framework. In addition, we explain how Forseti estimates the execution time of tasks and redistributes workload accordingly. Further, relying on Forseti, we propose a greedy algorithm which aims to minimize the job execution time.

4.1. Overview

We design Forseti to be applicable to any multi-phase DISC framework. Forseti assumes a job consists of tasks that run on parallel unrelated machines. Each task uses data within boundaries, reads it as records, generates key-value pairs and passes the pairs to the next phase. In addition, Forseti makes no assumption about *a priori* knowledge of the cluster state nor does it require to know about job requirements and configuration upon job submission (or past runs). Moreover, Forseti exploits "JVM reuse", and seamlessly redistributes data among running tasks without interruption.

Every task in a DISC framework is given a boundary which defines the start and end of a segment (or split) to be processed. A task completes when the segment end is reached. Tasks use a pointer to specify the start byte offset of a

Symbol	Description
$\mathcal{T}$	The set of all tasks of a job
$\mathcal{T}_r$	The set of all tasks currently processing
$\mathcal{T}_f$	The set of all idle (finished) tasks
$\mathcal{T}_w$	The set of all tasks waiting for resources (JVM)
$\mathcal{T}_o$	The set of all tasks being optimized
$\mathcal{S}$	The set of all segments being distributed
$\mathcal{W}_n$	The set of segments assigned to task n
τ	The current estimated finishing time of all tasks with the new rebalancing
D_n^r	The unprocessed workload for task n
D_s^e	The processed workload for segment s
D_s^r	The unprocessed workload for segment s
R_n	The real time process rate of task n
B_n	The estimated logical buffer size
p_s	Progress of segment s
ε	Threshold in seconds to terminate Forseti
σ	Threshold of progress score at which a task is included for optimization
ω	Threshold of progress score at which a task is excluded from optimization

Table 1: List of symbols

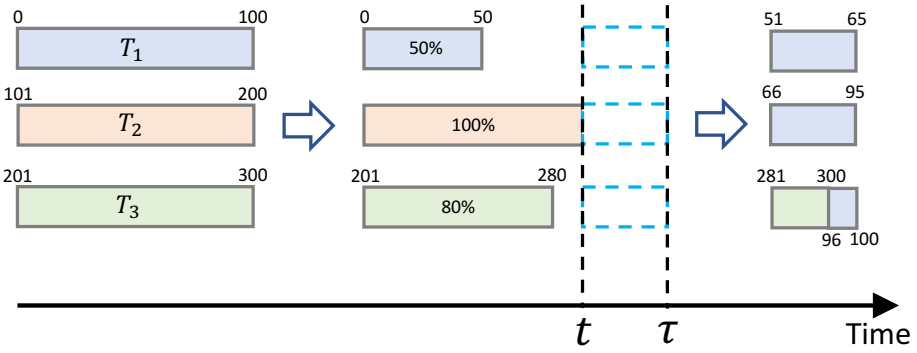


Figure 3: Segments Creation and Allocation

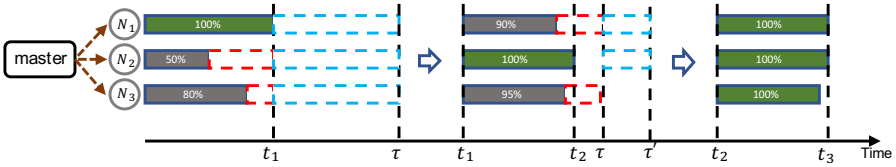


Figure 4: System architecture and steps taken upon new task completion.

record. The key of a record is the byte offset at which it is located, and the value is the data present in this record. The pointer is incremented by the record size in order to point to the next record. For instance, the start byte offset of the first record is 0, and if we assume the first record size is 100KB, the start byte offset of the second record is (100KB + 1B).

Forseti uses the idea of boundaries limits and pointers to specify the data to be processed by every task. Forseti defines distributed data as a set of segments, which have start and end byte offsets. Upon a job submission, every task is assigned one segment, and it is roughly the same size for every task. Any part of a segment can be specified by a start and end offset. As data is redistributed among running tasks periodically upon a new task completion, the number of segments assigned and their sizes are changing based on the current state of tasks. Forseti creates a logical *buffer* when optimizing distributed data. The buffer size of every task is calculated based on the estimated finishing time. That is, the master finds the amount of data that should be assigned to every task so that all tasks finish roughly at the same time. This data defines the

buffer size in bytes.

Figure 3 explains the concept of segments and how they are created and distributed periodically among tasks. Let us suppose we have one file of size 300B to be processed. If the configured initial segment size is 100B, the master launches three tasks, each of which processes 100B. The numbers shown on top and bottom are the start and end byte offset. At time t, T_2 finishes processing the assigned data, while T_1 and T_3 are still processing data. Based on the process rate, the master finds that the remaining data from T_1 can be split into three segments with different sizes. The blue dashed rectangle represents the buffer (the estimated data to be processed by τ). The segments are redistributed among the running tasks, where T_1 processes 15B extra, T_2 processes 30B and T_3 finishes processing its data and processes one segment from T_1 (5B). Note that the master only sends the start and end byte offset to every designated task. Then, the tasks use their pointers to point to the start byte offset of a segment achieving online rebalancing.

4.2. Estimating New Workload Assignment of Tasks

Forseti is designed based on the assumption that *a priori* knowledge of the cluster capability and the submitted jobs configuration are unknown. Thus, Forseti has to deal with and adapt to the divergence in the cluster performance. Forseti relies on the real time tasks' progress rate and data remaining to be processed in order to estimate the tasks' finishing time. Moreover, Forseti takes into account the remaining data of all tasks when rebalancing workload. That is, the remaining workload is redistributed among all tasks so that the completion time is minimized.

To estimate the new amount of data to be assigned to a task, we first need to calculate the estimated finishing time (τ) considering the total remaining data and the progress rate of all tasks. τ is calculated as follows:

$$\tau = \frac{\sum_{n=1}^{|\mathcal{T}_r|} D_n^r}{\sum_{n=1}^{|\mathcal{T}_o|} R_n} \tag{1}$$

where D_n^r and R_n are the unprocessed workload and progress rate of task n, respectively. We use the term *buffer* to represent the logical available space in every task which can be filled with data. Once τ is found, the estimated buffer size B_n available for task n is calculated as follows:

$$B_n = R_n \cdot \tau. \tag{2}$$

The buffer size plays an important role in defining the segments limits.

4.3. Proposed Dynamic Load Balancing Algorithm

In Forseti, we use a greedy algorithm to fill buffers with data. After a job submission, the master launches tasks with a pre-configured segment (or split) size. Then, the master monitors every task and waits for tasks to finish. Every task notifies the master upon completion. The master waits for at least one task to finish before optimizing workload among tasks. Figure 4 shows the manner in which the master reacts to a new finished task. Suppose that a job is submitted to a cluster and is running. For the sake of simplicity, suppose that the job has only three tasks and they start running simultaneously. T_1, T_2 and T_3 run on node N_1, N_2 and N_3, respectively. The percentage shown represents the fraction of data processed. The master polls the status of every task periodically and records the progress rate R based on the number of bytes processed and elapsed time. Based on the first workload assignment, T_1 finishes first at t_1, while T_2 and T_3 are still running. The dashed red rectangle represents the unprocessed data D_2^r and D_3^r in T_2 and T_3, respectively. Now, the master needs to redistribute the workload in T_2 and T_3 among all three tasks so that all tasks finish roughly at the same time with the new assignment. The master first estimates the finishing time τ of all tasks based on the remaining workload and the real time progress rate. That is, the master tries to find the amount of data that should be given to every task based on its progress rate so that they all finish roughly at the same time. The blue dashed rectangle represents the estimated buffer size after workload taken from T_2 and T_3 is redistributed. Once τ is found, the master notifies every task about the new start and end offsets of

segments assigned to be processed. After the first load rebalancing, T_2 finishes first at t_2. This indicates that T_2, which suffers temporal slowdown at t_1, is able to finish before τ. Now, the master estimates a new finishing time τ', and the unprocessed data of T_1 is divided among T_1 and T_2 based on the current progress rate, while T_3 is left untouched as it is about to finish processing. If all tasks are about to finish, and no more data can be redistributed, the master checks if there are tasks waiting to be launched. If found, their data gets redistributed in the same manner among running tasks, and they get removed from the system. This process eliminates the JVM launching time overhead and exploits "JVM reuse" of running tasks. This optimization is repeated periodically upon a new task completion until all tasks finish processing or τ becomes very small. Forseti ensures seamless execution throughout the lifetime of a task. Tasks only need to point to the right start offset of a segment and continue processing from there. If a task's progress rate indicates that it can process more data after completion and before τ, the master assigns new segments to be processed right after completing the current workload.

In order to redistribute data among tasks, we first need to find the set of segments $\mathcal{S}$, which includes all non-processed segments previously assigned to tasks, as well as the remaining data from the current segment being processed that will not be processed after τ. We also need to find the set of tasks $\mathcal{T}_o$ which make enough progress, and an accurate process rate is recorded. Algorithm 1 explains how segments are collected from running tasks (already started processing). Steps (2-10) check the progress of the current segment being processed, s, and add all remaining unprocessed segments to $\mathcal{S}$. In our optimization, we only consider tasks that make enough progress (more than ω) and there is more than σ of data remaining in the current segment. The remaining segments of the task which is about to finish processing the current segment are excluded. Steps (11-20) calculate the buffer size and checks whether a task can process more data after finishing the current segment.

Algorithm 2 works in a greedy manner to fill the logical tasks' buffer with segments. Forseti always prioritizes re-balancing the workload of running tasks.

Algorithm 1: unprocessedSegs()

$\mathcal{S} = \{\emptyset\}$
for $n \in \mathcal{T}_r$ **do**
  $s = \mathcal{W}_n[c]$ \\c: Current segment being processed
  \\Check progress of current segment
  if $\sigma > p_s \geq \omega$ **then**
    $\mathcal{T}_o.append(n)$
    $\mathcal{S}.append(\mathcal{W}_n[c+1:])$
  else if $p_s < \omega$ **then**
    $\mathcal{S}.append(\mathcal{W}_n[c+1:])$ \\Add all unprocessed segments
  end if
end for
for $n \in \mathcal{T}_o$ **do**
  $s = \mathcal{W}_n[c]$
  $B_n = R_n \cdot \tau$\\Calculate estimated buffer size for task n
  \\Add remaining unprocessed workload for segment c
  if $D_s^r - B_n > 0$ **then**
    $\kappa = s.start + D_s^e + B_n$
    $\mathcal{S}.append([\kappa, s.end])$
  else
    $B_n = B_n - D_s^r$
  end if
end for

Algorithm 2: segmentsAssign()

```
1: if S = {∅} then
2:   if T_w ≠ {∅} then
3:     \\Assign unscheduled tasks' workload to idle tasks
4:     for v ∈ min(|T_w|, |T_f|) do
5:       x = T_w[v]
6:       y = T_f[v]
7:       W_y.append(W_x)
8:     end for
9:   end if
10: else
11:   while T_o ≠ {∅} do
12:     T_0 = T_o[0]
13:     W_0 = []
14:     while B_0 > 0 do
15:       s = S[0]
16:       y = s.end − s.start\\Data size of segment s
17:       while y > 0 & B_0 > 0 do
18:         \\Check if whole segment fits in buffer
19:         if B_0 ≥ y then
20:           W_0.append(s)
21:           B_0 = B_0 − y\\Decrease buffer size
22:           y = 0
23:           S = S − {s}\\Remove segment s from S
24:           \\Remove task T_0 from T_o if buffer is filled
25:           if B_0 == 0 then
26:             T_o = T_o − {T_0}
27:           end if
28:         else
29:           \\Fill the remaining buffer space
30:           z = s.start
31:           W_0.append([z, z + B_0])
32:           s.start = z + B_0
33:           y = y − B_0
34:           T_o = T_o − {T_0}
35:         end if
36:       end while
37:     end while
38:   end while
39: end if
```

The unscheduled tasks, $\mathcal{T}_w$, are only considered if there are no segments to be redistributed from the running tasks. Steps (1-8) assign the initial segment assigned to an unscheduled task to an idle task (a finished task waiting for more data to be processed). Once the segment is assigned, the unscheduled task gets removed from the cluster. Steps (9-35) assign segments from $\mathcal{S}$ to all tasks in $\mathcal{T}_o$, where $\mathcal{T}_f \cup \mathcal{T}_r = \mathcal{T}_o$. The algorithm assigns segments to a task until no more space is available for data to be processed before τ. If a whole segment cannot be assigned to a task, it gets split between two or more tasks. Algorithm 3 shows when Forseti allows tasks to commit their results. If τ is less than or equal to a threshold, all tasks commit their results and Forseti terminates.

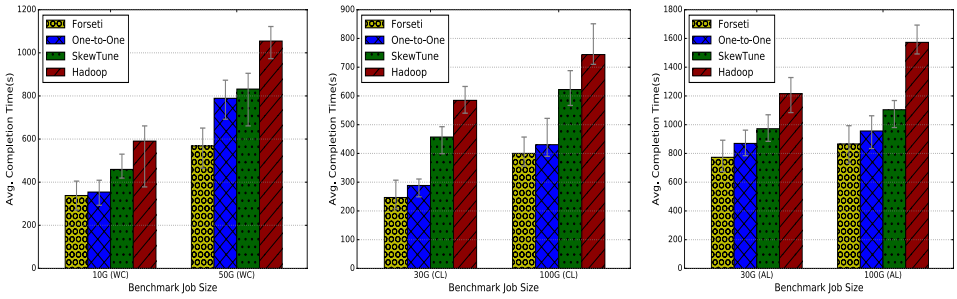
Algorithm 3: commForseti()

```
1: \\Commit if no more segments in S and no more tasks waiting
2: if S = {∅} & T_w = {∅} & τ > ε then
3:    \\Ask every finished task to commit
4:    for n ∈ T_f do
5:       n.commit = true
6:    end for
7: else if τ ≤ ε then
8:    n.commit = true, ∀n ∈ T
9: end if
```

5. Forseti Implementation

While Forseti works with any distributed computing framework, to evaluate its performance in this paper, we implement Forseti using Hadoop YARN, which includes an RM (Resource Manager), an AM (Application Master) for each application (job) as well as an NM (Node Manager within each node). The AM negotiates resources from the RM and works with the NMs to execute and monitor an application's tasks. Forseti is job-independent, so the algorithm is implemented in the AM to estimate τ and redistribute workload among all tasks of a job accordingly.

The Forseti load balancing algorithm is centralized and implemented in the AM. The AM keeps tracking the segments assigned to every task. The AM runs

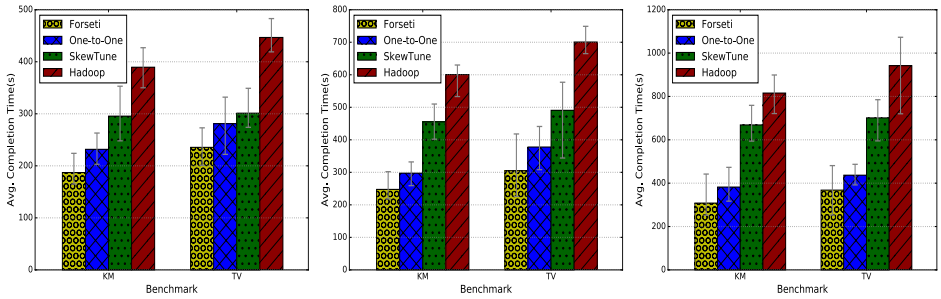


(a) (b) (c)

Figure 5: Comparisons of Forseti, One-to-One, SkewTune and Hadoop in terms of average completion time using three benchmarks: (a) WordCount (b) Classification (c) Adjacency List. The heterogeneity level is 1-2-3.

the algorithm upon a new task finishing. Instead of relying on progress score sent from tasks to AM, when load re-balancing is needed, we let the AM poll for the real-time progress rate from tasks and the current segment being processed. We create a new thread for every task which is in charge of communicating with the AM and updating it with current status upon request. The outputs of all segments are concatenated before a task commits the results and are fetched by reduce tasks.

One challenge confronting us is that when calculating the start byte offset of a segment, the AM does not know about the start offset of records within a segment. These offset values are only known to tasks when processing segments. Because of the resulting fraction from R_n and τ, there is no guarantee that the start byte offset of a segment always leads to the start byte offset of a record. Recall that the start byte offset of a record is the key, and the value is the data present in this record. Thus, the start byte offset usually points to the value of a record, not the key. Therefore, we let tasks always skip the first record of new segments and point to the start byte offset of the second record upon processing. The skipped part will be the last record to be processed by whoever gets assigned the corresponding segment.



(a) (b) (c)

Figure 6: Comparisons of Forseti, One-to-One, SkewTune and Hadoop in terms of average completion time using Kmeans and TermVector benchmarks with different level of heterogeneity: (a) 1-2 (b) 1-2-3 (c) 1-2-3-4.

6. Evaluation

The performance of Forseti is evaluated on a local cluster as well as Amazon EC2 cloud. In this section, we present the evaluation results. We first give a description of the experimental setup, and then we show our results comparing Forseti with SkewTune and Hadoop. We also compare Forseti with a simple heuristic which works as *one-to-one* mapping. That is, upon completion of a new task f, we find the slowest running task l that will finish the latest, based on its progress. Then, the remaining data is divided among these two tasks such that they both finish at the same time. The slow task is notified to process B_l more bytes, while the idle task processes the remaining. B_l is calculated as follows:

$$B_l = \frac{R_l \cdot D_l^r}{R_l + R_f} \tag{3}$$

where D_l^r, R_l, and R_f are the remaining unprocessed data from the slow task, the process rate of the slow task and the process rate of the idle (finished) task, respectively.

6.1. Experimental Setup

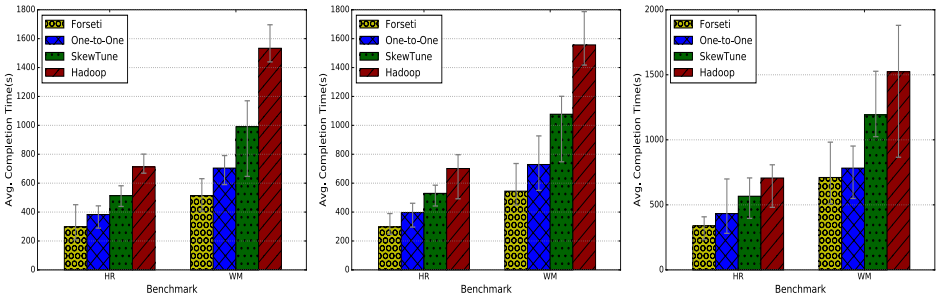
Forseti is deployed on a local cluster and Amazon EC2 consisting of 101 nodes - one master and 100 slaves, and 145 nodes - one master and 144 slaves,

respectively. All local servers are connected with a Gigabit Ethernet switch, each of which is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz and run on Ubuntu 16.04.6 LTS operating system. We set ω, σ and ε equal to 5%, 90% and 15 seconds, respectively. These variables help determine whether the segment currently processed should be included in the optimization. If not chosen properly, some segments might get processed more than once. Note that the variables can be environment- and applications-specific values. We use Docker [31] platform to create a cluster with different levels of heterogeneity. In our cluster, each node is capable of running one task at a time. Forseti is evaluated by using popular benchmarks, TermVector (TV), WordCount (WC) and WordMean (WM) as well as Machine Learning benchmarks such as Histogram Ratings (HR), Classification (CL) and KMeans (KM) clustering benchmarks, and Graph processing benchmark such as Adjacency List (AL) [12]. The figures show average completion times of 20 jobs as well as the maximum and minimum of these completion times.

6.2. Results

Figure 5 compares the average completion time of Forseti with One-to-One, SkewTune and Hadoop for various job sizes and benchmarks. In this figure, we fix the level of heterogeneity to 1-2-3 ratio (i.e., CPU speeds are 1x, 2x and 3x of a base speed) and run jobs with different workload size (i.e., 10G, 30G, 50G and 100G bytes of data) one by one and measure the completion time for each job. The figures show that Foresti outperforms all strategies and reduces the average completion time. The figures also show that even with large data size, Forseti is able to exploit the dissimilarity in progress rate among tasks and redistributes workload efficiently. We notice in this figure that the gap between Forseti and SkewTune stays roughly the same with different data size, but Forseti still provides better results. However, the difference in completion time between Forseti and Hadoop increases as we increase the job size. This demonstrates Forseti's superiority in dealing with large jobs.

In Figure 6, we compare the average completion time of Forseti with One-



(a) (b) (c)

Figure 7: Comparisons of Forseti, One-to-One, SkewTune and Hadoop in terms of average completion time using Histogram Ratings and WordMean benchmarks with different job contention levels: (a) Low (b) Medium (c) High.

to-One, SkewTune and Hadoop for different heterogeneity levels. In this figure, we fix the job size and run the benchmarks with various cluster settings, i.e., 1-2, 1-2-3, 1-2-3-4 ratio of CPU speed. As in the previous experiment, jobs are presented one by one to the system, and the completion time is measured and the average taken over all jobs. The two benchmarks, KM and TV, process 30G and 10G bytes of data, respectively. Clearly, we can see that as we increase the heterogeneity level, Forseti is able to maintain a superior performance difference compared with other strategies. In this figure, we can see that the gaps with SkewTune and Hadoop keep increasing as we increase the heterogeneity level. This is because Hadoop and SkewTune make no assumptions about the fluctuation in processing speed in the cluster. The results also show that, even with a highly heterogeneous system, Forseti shows to be more appealing compared with SkewTune and Hadoop. Forseti can adapt and adjust to the current state of a system regardless of benchmarks and the discrepancy in progress rate.

In the previous two experiments, jobs were presented one by one to the system and did not compete with each other for system resources; the only contention for resources is among the tasks of the same job. In the next experiment, we allow multiple jobs to compete with each other. Here, we present j jobs simultaneously to the system and the tasks of these j jobs compete with each other. Note that if there are not enough VMs to launch all the tasks of

these jobs, some will have to be scheduled after other tasks finish. We measure the completion time of each job from the time the job is presented to the system (i.e., including any waiting time for launching the job's tasks). Figure 7 presents a comparison of the average completion time of Forseti with One-to-One, SkewTune and Hadoop for 3 different contention levels: Low ($j = 2$), medium ($j = 3$), and high ($j = 5$) contention levels. In this experiment, we run two benchmarks, HR and WM, which process 30G and 10G bytes of data, respectively. The figures show that Forseti notably outperforms all baselines and is able to reduce the average completion time. The figures also show that SkewTune fails to perform well at high contention level especially for WM benchmark. That is, with high contention level, SkewTune is not able to reduce the completion time of the jobs running on the cluster and free resources for the jobs that need them. On the other hand, Forseti's outstanding performance is due to the fact that the processing time of jobs is minimized, which makes resources available for yet-to-be-scheduled jobs.

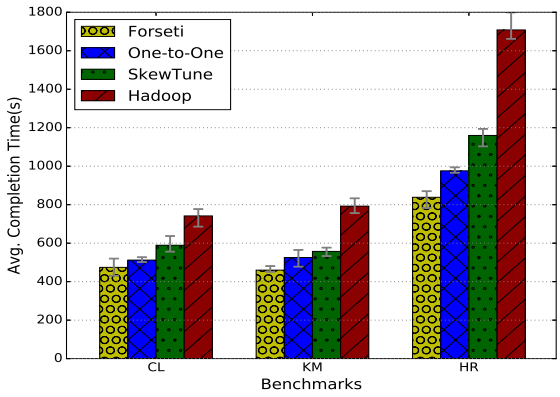


Figure 8: Comparisons of Forseti, One-to-One, SkewTune and Hadoop in terms of average completion time with 300GB of workload using Classification, KMeans and Histogram Ratings benchmarks on Amazon EC2.

Figure 8 depicts results from experiments on EC2. The figure shows the average completion time of Forseti compared with One-to-One, SkewTune and Hadoop for different benchmarks with 300GB of workload. The results show that, even with large jobs, Forseti significantly outperforms all baselines. This

improvement over other strategies is due to the fact that the assigned segments are always reassigned if the AM finds that a new data distribution is needed to minimize the completion time.

	ARF	ACT(s)
10G (WC)	9	338
50G (WC)	14	568
30G (CL)	6	247
100G (CL)	10	400
1-2 (KM)	5	188
1-2-3 (KM)	6	248
1-2-3-4 (KM)	7	309
1-2 (TV)	5	236
1-2-3 (TV)	7	306
1-2-3-4 (TV)	8	368

Table 2: Average number of re-optimizations of Forseti for different workloads, benchmarks and heterogeneity levels.

Table 2 shows the average re-optimization frequency (ARF) and the average completion time (ACT) in seconds of Forseti for different workloads, benchmarks and heterogeneity levels. The table shows that the re-optimization frequency is directly proportional to the total completion time of a job. The table also shows jobs with different characteristics and workload sizes require different number of re-optimizations, and large job sizes do not always provide enough information about jobs. Thus, Forseti makes no assumption about the jobs and adapts to the current state of a system accordingly.

7. Conclusion

In this paper, we present Forseti, a dynamic load balancing framework that aims to adjust the workload assigned to each task periodically according to their processing capability. Forseti makes no assumption about the cluster state and optimizes workload based on the current tasks' state. It exploits "JVM reuse" by assigning more data to running tasks and avoiding launching new tasks. In addition, Forseti does not need to investigate the cause of discrepancy in execution time of tasks. We design Forseti to be transparent to the map function

and require no modification to the function design. Our results show that Forseti can significantly reduce job completion time by up to 68% on average compared to default Hadoop and 50% to SkewTune.

In the future work, we plan to investigate the network usage and conduct deep analysis and its impact on the performance. Furthermore, we aim to evaluate how our greedy algorithm scales as the complexity increases.

References

[1] Ahmad, F., Chakradhar, S.T., Raghunathan, A., Vijaykumar, T., 2012. Tarazu: optimizing mapreduce on heterogeneous clusters. ACM SIGARCH Computer Architecture News 40, 61–74.

[2] Ahmad, Z., Duppala, S., Chowdhury, R., Skiena, S., 2020. Improved mapreduce load balancing through distribution-dependent hash function optimization, in: 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS), IEEE. pp. 9–18.

[3] Aktas, M.F., Peng, P., Soljanin, E., 2018. Straggler mitigation by delayed relaunch of tasks. SIGMETRICS Perform. Eval. Rev. .

[4] Alamro, S., Xu, M., Lan, T., Subramaniam, S., 2018. Shed: Optimal dynamic cloning to meet application deadlines in cloud, in: 2018 IEEE International Conference on Communications (ICC).

[5] Alamro, S., Xu, M., Lan, T., Subramaniam, S., 2020. Shed+: Optimal dynamic speculation to meet application deadlines in cloud. IEEE Transactions on Network and Service Management 17, 1515–1526.

[6] Alipourfard, O., Liu, H.H., Chen, J., Venkataraman, S., Yu, M., Zhang, M., 2017. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics, in: 14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17), pp. 469–482.

[7] Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., Harris, E., 2011. Scarlett: coping with skewed content popularity in mapreduce clusters, in: Proceedings of the sixth conference on Computer systems, pp. 287–300.

[8] Ananthanarayanan, G., Ghodsi, A., Shenker, S., Stoica, I., 2013. Effective straggler mitigation: Attack of the clones, in: NSDI'13.

[9] Ananthanarayanan, G., Kandula, S., Greenberg, A.G., Stoica, I., Lu, Y., Saha, B., Harris, E., 2010. Reining in the outliers in map-reduce clusters using mantri., in: OSDI'10.

[10] Angel, S., Ballani, H., Karagiannis, T., O?Shea, G., Thereska, E., 2014. End-to-end performance isolation through virtual datacenters, in: 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), pp. 233–248.

[11] Anjos, J.C., Carrera, I., Kolberg, W., Tibola, A.L., Arantes, L.B., Geyer, C.R., 2015. Mra++: Scheduling and data placement on mapreduce for heterogeneous environments. Future Generation Computer Systems 42, 22–35.

[12] Apache Software Foundation, . Puma: Purdue maprduce benchmark suite. URL: `https://engineering.purdue.edu/~puma/pumabenchmarks.htm`.

[13] Ballani, H., Jang, K., Karagiannis, T., Kim, C., Gunawardena, D., O?Shea, G., 2013. Chatty tenants and the cloud network sharing problem, in: Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13), pp. 171–184.

[14] Chen, W., Rao, J., Zhou, X., 2017. Addressing performance heterogeneity in mapreduce clusters with elastic tasks, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE. pp. 1078–1087.

[15] Cheng, D., Rao, J., Guo, Y., Zhou, X., 2014. Improving mapreduce performance in heterogeneous environments with adaptive task tuning, in: Proceedings of the 15th International Middleware Conference, pp. 97–108.

[16] Coppa, E., Finocchi, I., 2015. On data skewness, stragglers, and mapreduce progress indicators, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, pp. 139–152.

[17] Fu, S., Mittal, R., Zhang, L., Ratnasamy, S., 2020. Fast and efficient container startup at the edge via dependency scheduling, in: 3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20).

[18] Gandhi, R., Xie, D., Hu, Y.C., 2013. {PIKACHU}: How to rebalance load in optimizing mapreduce on heterogeneous clusters, in: Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13), pp. 61–66.

[19] Gunasekaran, J.R., Thinakaran, P., Nachiappan, N.C., Kandemir, M.T., Das, C.R., 2020. Fifer: Tackling resource underutilization in the serverless era, in: Proceedings of the 21st International Middleware Conference, pp. 280–295.

[20] Herodotou, H., Dong, F., Babu, S., 2011a. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, pp. 1–14.

[21] Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S., 2011b. Starfish: A self-tuning system for big data analytics., in: Cidr, pp. 261–272.

[22] Ibrahim, S., Jin, H., Lu, L., He, B., Antoniu, G., Wu, S., 2012. Maestro: Replica-aware map scheduling for mapreduce, in: Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on, IEEE. pp. 435–442.

[23] Jang, K., Sherry, J., Ballani, H., Moncaster, T., 2015. Silo: Predictable message latency in the cloud, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, pp. 435–448.

[24] Kandula, S., Sengupta, S., Greenberg, A., Patel, P., Chaiken, R., 2009. The nature of data center traffic: measurements & analysis, in: SIGCOMM'09.

[25] Kwon, Y., Balazinska, M., Howe, B., Rolia, J., 2010. Skew-resistant parallel processing of feature-extracting scientific user-defined functions, in: Proceedings of the 1st ACM symposium on Cloud computing, pp. 75–86.

[26] Kwon, Y., Balazinska, M., Howe, B., Rolia, J., 2011. A study of skew in mapreduce applications. Open Cirrus Summit 11.

[27] Kwon, Y., Balazinska, M., Howe, B., Rolia, J., 2012. Skewtune: mitigating skew in mapreduce applications, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 25–36.

[28] Le, Y., Liu, J., Ergün, F., Wang, D., 2014. Online load balancing for mapreduce with skewed data input, in: IEEE INFOCOM 2014-IEEE Conference on Computer Communications, IEEE. pp. 2004–2012.

[29] Lee, C.W., Hsieh, K.Y., Hsieh, S.Y., Hsiao, H.C., 2014. A dynamic data placement strategy for hadoop in heterogeneous environments. Big Data Research 1, 14–22.

[30] Lion, D., Chiu, A., Sun, H., Zhuang, X., Grcevski, N., Yuan, D., 2016. Don't get caught in the cold, warm-up your {JVM}: Understand and eliminate {JVM} warm-up overhead in data-parallel systems, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 383–400.

[31] Merkel, D., 2014. Docker: lightweight linux containers for consistent development and deployment. Linux journal 2014, 2.

[32] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.G., 2015. Making sense of performance in data analytics frameworks, in: 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), pp. 293–307.

[33] Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I., 2013. Sparrow: distributed, low latency scheduling, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 69–84.

[34] Pettijohn, E., Guo, Y., Lama, P., Zhou, X., 2014. User-centric heterogeneity-aware mapreduce job provisioning in the public cloud, in: 11th International Conference on Autonomic Computing ({ICAC} 14), pp. 137–143.

[35] Vernica, R., Balmin, A., Beyer, K.S., Ercegovac, V., 2012. Adaptive mapreduce using situation-aware mappers, in: Proceedings of the 15th International Conference on Extending Database Technology, pp. 420–431.

[36] Wang, B., Jiang, J., Yang, G., 2015. Actcap: Accelerating mapreduce on heterogeneous clusters with capability-aware data placement, in: 2015 IEEE Conference on Computer Communications (INFOCOM), IEEE. pp. 1328–1336.

[37] Wang, D., Joshi, G., Wornell, G.W., 2019. Efficient straggler replication in large-scale parallel computing. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 4, 1–23.

[38] Wang, W., Ying, L., 2016. Data locality in mapreduce: A network perspective. Performance Evaluation 96, 1–11.

[39] Xie, J., Yin, S., Ruan, X., Ding, Z., Tian, Y., Majors, J., Manzanares, A., Qin, X., 2010. Improving mapreduce performance through data placement in heterogeneous hadoop clusters, in: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), IEEE. pp. 1–9.

[40] Xu, F., Liu, F., Jin, H., 2015. Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud. IEEE Transactions on Computers 65, 2470–2483.

[41] Xu, H., Lau, W.C., 2017. Optimization for speculative execution in big data processing clusters. IEEE Transactions on Parallel and Distributed Systems 28, 530–545.

[42] Xu, M., Alamro, S., Lan, T., Subramaniam, S., 2017. Laser: A deep learning approach for speculative execution and replication of deadline-critical jobs in cloud, in: Computer Communication and Networks (ICCCN), 2017 26th International Conference on, IEEE. pp. 1–8.

[43] Xu, M., Alamro, S., Lan, T., Subramaniam, S., 2018. Chronos: A unifying optimization framework for speculative execution of deadline-critical mapreduce jobs, in: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), IEEE.

[44] Yang, D., Rang, W., Cheng, D., 2020. Mitigating stragglers in the decentralized training on heterogeneous clusters, in: Proceedings of the 21st International Middleware Conference, pp. 386–399.

[45] Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R.H., Stoica, I., 2008. Improving mapreduce performance in heterogeneous environments., in: OSDI'08.

[46] Zhang, Z., Cherkasova, L., Loo, B.T., 2015. Exploiting cloud heterogeneity to optimize performance and cost of mapreduce processing. ACM SIGMETRICS Performance Evaluation Review 42, 38–50.