ABSTRACT

KUKRETI, SARTHAK. Reducing Hadoop's long tail with Process Cloning. (Under the direction of Dr. Rainer Frank Mueller.)

Recent advances in distributed computing have enabled large-scale data processing on high volumes of data. Programming models like MapReduce and Spark parallelize segments of applications that can be scheduled and executed on clusters of commodity computers. However, the overall runtime of such applications is dependent on the slowest subtask and even a few slow subtasks can significantly slow down an application. Current approaches to mitigate this effect, both proactive and reactive, involve scheduling redundant speculative attempts of slow tasks (also known as straggler tasks). However, at the time of launch, the speculative attempts have to recompute the work already done by the original task. This heavily hinders the capability of late speculations to catch up to (and subsequently run faster than) the straggling task.

In this work, we propose the utilization of process cloning in speculative attempts to avoid redundant computations. Several challenges arise on using cloned speculative attempts, namely, maintaining consistency between task attempts, both during and after cloning, as well as enabling divergent execution paths to differentiate between attempts. Additionally, complex components of subtasks may require recovery on cloning, both in the original and the cloned process.

Our contributions in this work are three-fold: We develop a process cloning tool for enabling cloned speculations. Additionally, we discuss mechanisms to maintain consistency and recover complex components from errors caused due to cloning, in both the original and the cloned process. Finally, we have integrated the cloning and recovery mechanisms into Apache Hadoop and propose optimizations to Apache Hadoop to alleviate resource bottlenecks while cloning tasks. Our simulations show that the option to create clone speculations benefits straggling tasks closer to the median task. We observe performance improvements of up to 25% on jobs with larger tasks. Our prototype scales well with different task and problem sizes and performs well for different straggler task runtime distributions.

Reducing Hadoop's long tail with Process Cloning

by
Sarthak Kukreti

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

Dr. Hung-Wei Tseng	Dr. Ranga Raju Vatsavai

Dr. Rainer Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents and my elder brother.

BIOGRAPHY

The author hails from Delhi, India. He completed his B.E. in Computer Engineering from Netaji Subhas Institute of Technology, Delhi in 2013. He spent the next two years working as a Software Engineer at Qualcomm working on systems. He enrolled for the MS in Computer Science program at North Carolina State University in Fall 2015 and started working with Dr. Frank Mueller as a Research Assistant in August 2016. His hobbies include reading, listening to music and playing the guitar.

ACKNOWLEDGEMENTS

I would like to thank everyone who played a part in helping me complete this work: to paraphrase a famous thought, it took a medium-sized town to raise this work.

I would like to thank my advisor Dr. Frank Mueller for giving me the opportunity to work on such an interesting problem. His insights and guidance have been invaluable to me and are the biggest factors in the completion of this work. Dr. Mueller's words of advice and encouragement have kept me motivated through some of the most challenging yet enriching times of my life. I feel privileged to have worked with Dr. Mueller over the past year.

I would like to thank Dr. Ranga Raju Vatsavai and Dr. Hung-Wei Tseng for taking interest in my research and agreeing to be a part of my advisory committee. I would also like to thank the members of our research group for their constant help.

I would like to thank my room mates Vipin, Neel, Tushar and Soumik for enriching my stay in Raleigh. Time flies when you are having fun and my room mates have kept the clock ticking for the past two years.

I would like to my parents and my brother Akshat for emotionally supporting me throughout the course of this degree. Their trust and belief in my capabilites is what motivates me to try to work beyond my capabilities and challenge myself with bigger goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

The last decade has seen an exponential increase in both the volume of data available and the tools to process data at scale. Many applications, spanning diverse domains, have been adapted to take advantage of the parallel computation capabilities of paradigms like MapReduce [8] and Spark [31] [32]. Additionally, the availability of open source frameworks based on these models [31][5] as well as domain-specific libraries, like Mahout [22], Hive [25] and Pegasus [12], built on top of these frameworks have expanded the application domain for MapReduce and lowered the barrier for user expertise required to parallelize applications.

Programming models like MapReduce and Spark attempt to exploit the inherent parallelism in problems by separating it into parallelizable phases. Each phase can then be further divided into independent subtasks that can be parallely scheduled and executed on a large number of machines. Therefore, any improvement in performance per-task can significantly improve the overall job performance. On the other hand, even slight degradations in performance of a subtask can amplify and slow down an application significantly. Slow tasks — also known as **straggler tasks** — can potentially slow down the start of tasks dependent on them and substantially affect application response times.

Straggler mitigation techniques have primarily been of two types: reactive (speculative task execution [8][1], blacklisting nodes where stragglers occur) and proactive (preemptively schedule

multiple copies of tasks/jobs [2] or preventing stragglers in the first place [30] [27] [28]). However, both categories of techniques suffer from drawbacks: in case of reactive approaches, speculative tasks have to redo computations already done by the original task, which hinders their capability to catch up to the original task. Therefore, only a small fraction of speculated tasks actually end up finishing earlier than the original task. On the other hand, proactive techniques lead to high resource contention for larger jobs/task sizes: each copy of a task effectively doubles the overall resources utilized by a job.

1.1 Motivation: Efficiency of Speculative Executions

Dean et al. [8] originally identified stragglers as tasks with disproportionately slow completion times that negatively impacted the job response time and proposed speculative execution as a method to mitigate the effect of stragglers. Ren et al. [23] analyzed traces from three production clusters and made the following observations about speculative executions:

- Overall, speculative tasks finished faster than the respective original task in only 21% of all speculation events.
- Map tasks speculations did not lead to any significant difference in task completion times with respect to the original tasks.
- A large proportion of Reduce task speculations were too late to be effective (i.e. the speculative task was killed before it completed 10% of the original task's run time)

While early speculations are not very useful, late speculations suffer from the requirement to redo a significant portion of the computation to catch up to the original task. The effect is significantly exacerbated when considering long running jobs: a task has to be more than twice as slow as the median task in order to finish slower than a speculated copy.

Figure 1.1 shows an example of a task execution. At some progress percentage p%, the task is judged to be a straggler and a new speculative task is spawned and re-executes the entire work of the task. Even though the new speculative task is significantly faster than the original task, it still finishes after the original task. The time taken by the speculative task to complete redundant work significantly hampers its ability to catch up to the original task.

Instead, if the task were cloned such that re-execution is eliminated, the original task would be delayed by a certain amount of time (in order to carry out the cloning). However, since the cloned task now resumes from p%, any difference in speed can directly improve the overall execution time.

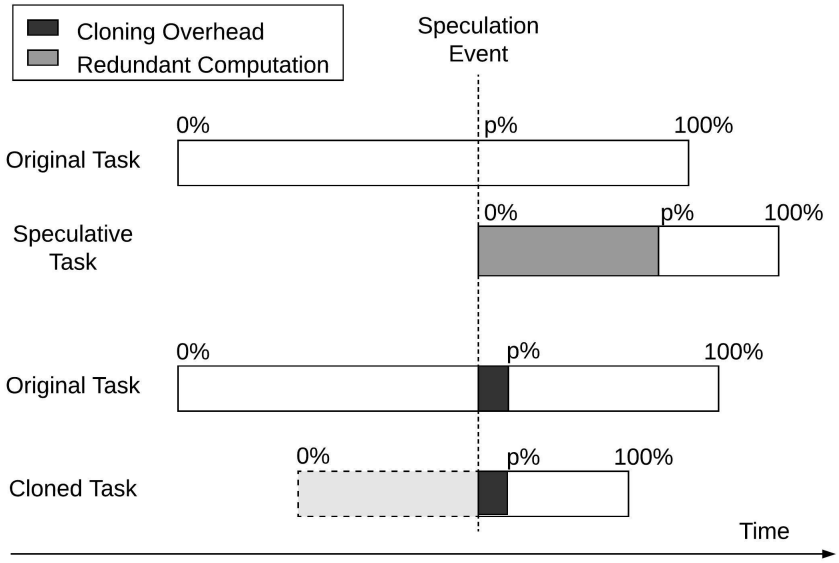


Figure 1.1 Motivation for improving speculative execution: Even when the speculative task is faster than the original task, it may have a later completion time due to the redundant work it has to complete. In contrast, cloning still leads to an overall faster completion time, even though cloning the task delays the original task

1.2 Hypothesis

We put forward the following hypothesis:

Speculative tasks have a higher probability of catching up to stragglers (and subsequently improving completion times) if they resume from current point of execution, instead of recomputing work already done by the straggling task.

This work proposes to improve reactive speculative execution of tasks by utilizing checkpoint-restart to create speculative task clones. This avoids redundant computation by the speculated copy and improves the probability of a speculative task completing before the original straggling task. Additionally, the approach is less resource-intensive compared to proactive approaches, since only straggling tasks are cloned.

Multiple challenges arise in using a checkpoint-restart based approach to cloning tasks, namely:

- maintaining data and resource consistency
- updating changes to the divergent attempt-specific state of a cloned task
- recovering from errors arising due to inconsistent state, in both the original and the cloned

task; and

- policy for decision to create either new task attempts or cloned task attempts.

We have implemented our approach on top of the Apache Hadoop framework: our prototype includes changes to Apache YARN [26] and the Hadoop Filesystem [24] to facilitate task cloning. Additionally, we propose changes to the structure of MapReduce computations as implemented in Apache Hadoop to alleviate resource bottlenecks for process cloning.

Our results show that the cloning speculator performs well for large-sized tasks and for lower-end of stragglers. Additionally, since the default speculator performs well on long-tailed straggler task runtime distributions, the two approaches complement each other. We observe a performance improvement of up to 25% in large jobs, without significantly impacting performance on smaller sized jobs.

The remaining part of this work is structured as follows. Chapter 2 describes the background behind utilizing process cloning to improve speculative execution. Chapter 3 expands upon the different approaches taken towards straggler mitigation. Chapter 4 and 5 describe our approach and the implementation our prototype on top of Apache Hadoop. Chapter 6 and 7 describe the framework we use for testing our hypothesis and the performance of our system under experimental analysis. Chapter 8 describes the limitations and future work areas and chapter 9 concludes our findings.

CHAPTER

2

BACKGROUND

2.1 MapReduce Overview

Dean et al. [8] introduced MapReduce as a programming model to enable large scale processing of data. MapReduce models applications as a combination of two ordered map and reduce phases:

- During the **map phase**, each key-value pair is mapped to an intermediate key-value pair. This stage allows for functions like filtering, sorting and data transformations
- During the **reduce phase**, intermediate key-value pairs are collected by key and a reducing function to the list of values associated with a key.

Figure 2.1 shows the phases of a MapReduce application. Inherently, the execution of the two phases is serialized: there is an implicit barrier at the end of the map phase. In order for the reduce phase to proceed, all of the input data needs to be processed. However, within each phase, the lack of any dependence in the map or reduce function allows for key-partitioned parallelism by partitioning the phase into a large number of subtasks to scale to applications that deal with large volumes of data. This allows the partition of each phase into subtasks that can be parallely scheduled and executed on a large number of machines, thus allowing applications to scale to large problem sizes.

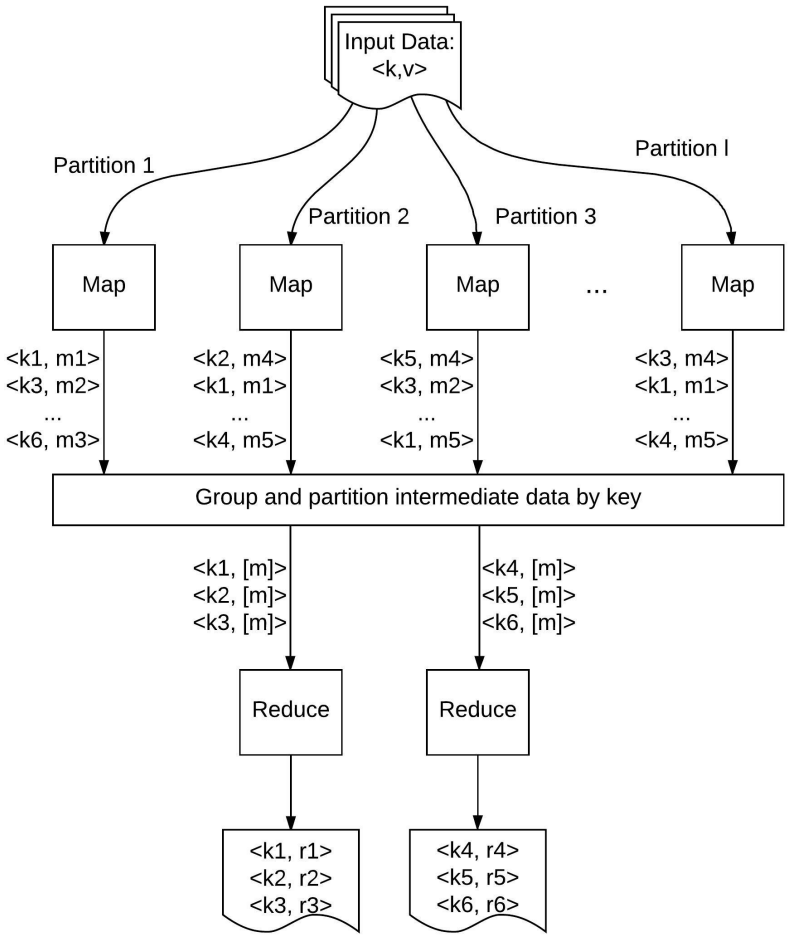


Figure 2.1 Overview of MapReduce paradigm: Each map subtask maps a partition of the input data set into intermediate key-value pairs. The intermediate data is grouped and partitioned by key to be handled by reduce phase. The lack of dependence between computations within a phase allows for high degrees of parallelism within a phase.

2.2 Apache Hadoop: Design and Architecture

Apache Hadoop [5] is an open source implementation of MapReduce programming model and includes supporting libraries that facilitate bookkeeping for MapReduce applications. While the initial implementation was heavily tied to the original MapReduce reference implementation, Hadoop has evolved, in recent years, to a generic, modular and extensible framework that supports other programming models as well. The following subsections describe the main components of the Hadoop framework. In addition to the infrastructure components, Hadoop also comprises of auxiliary libraries, like intermediate file compression and data formats, to support the MapReduce implementation and extend it to more generic interfaces.

2.2.1 Hadoop Filesystem (HDFS)

Hadoop Fielsystem is a distributed file system similar to the Google File System [9]. HDFS facilitates a unified filesystem view of data for Hadoop tasks. Architecturally, it is comprised of a master Namenode and multiple slave Datanodes. The Namenode is responsible for managing the metadata and bookkeeping all file operations on the distributed file system, whereas the Datanodes store the actual file content in blocks. Files on HDFS are stored as a collection of blocks. Similar to the Google File System, HDFS supports file replication to increase fault tolerance in case of node failure.

2.2.2 Yet Another Resource Negotiator (YARN)

Apache YARN is responsible for handling resource management, job scheduling and monitoring on a cluster. All application resource requests are handled by YARN. YARN comprises of:

- **Resource Manager:** The global resource manager comprises of a Scheduler and an Applications Manager. While the Scheduler is responsible for managing resource allocations for applications, the Applications Manager handles application submissions and initializing the application's first containers. The Resource Manger delegates all application-specific schedules to a per-application Application Master. All resource requests for an application are routed to the Resource Manager through the Application Master.
- **Node Manager:** Each node in the cluster runs a Node Manager, which is responsible for managing the lifecycle of a container. The Node Manager can also run pluggable services like the Shuffle Handler, which offloads the process of serving intermediate output to Reduce tasks.

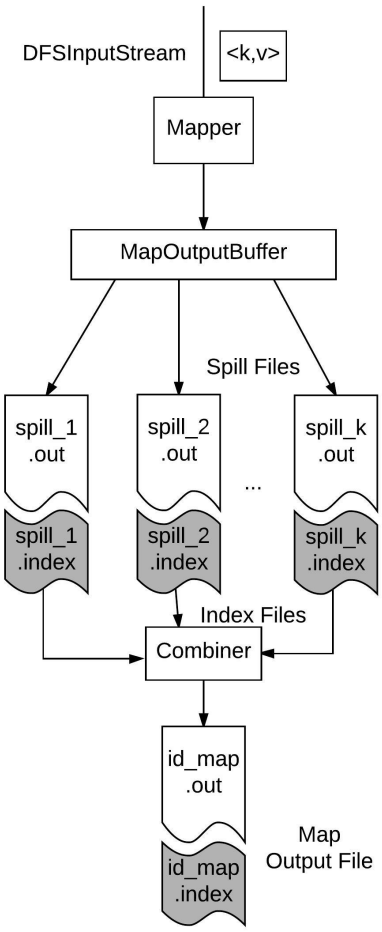


Figure 2.2 Simplified Map Task workflow: the MapOutputBuffer spills key-value pairs to disk when it runs out of memory. Once all key-value pairs have been processed, the buffer spills the remaining data to disk and the combiner collects all spill files into a single map output file.

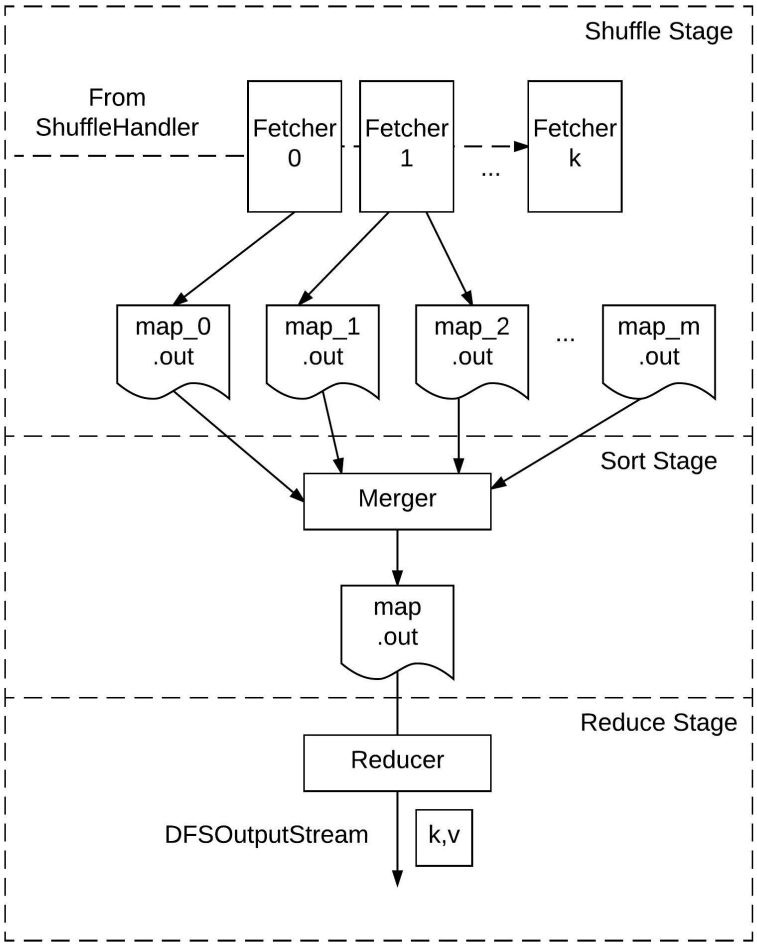


Figure 2.3 Simplified Reduce Task workflow: the Fetchers pull files from ShuffleHandlers on all nodes into memory. If the MapOutputFile does not fit into memory, it is spilled directly to disk. Depending on the size of the output files, the final merged file can contain a combination of disk and in-memory data.

2.3 Lifecycle of a Hadoop job

MapReduce applications are submitted to the Resource Manager as jobs. The following sequence of steps describes the overall execution of a MapReduce application.

- The **Application Master** (AM) is launched on one of the nodes. The Application Master is responsible for coordinating and managing the execution of all the component tasks of an application. This includes monitoring task progress, setting up and cleaning the execution context for tasks, negotiating resources with the resource manager.
- On start up, the Application Master negotiates the number of containers it needs to run the application. Once the containers are assigned, the Application Master can send the container context to each Node Manager via events.
- The Application Master also creates partitions of the input data for the tasks to process.
- Once the initial bookkeeping is complete, the application master schedules tasks that are independent of other tasks. For example, in a MapReduce computation, all map tasks are eligible to run, whereas the reduce tasks are dependent on the completion of all map tasks.
- **Map Task:** Figure 2.2 shows a simplified workflow for a map task. On getting scheduled, the map task starts reading key-value pairs from the input partition. The mapper function is then applied to each key-value pair and the resulting intermediate output is collected in a **map output buffer**. When the buffer is full or filled up to a configurable percentage (80% by default), it purges the key-value pairs into spill files. Once the map task has completed processing all key-value pairs in its allotted partition, it merges all spill files into a single **map output file**. On completion, the map task sends a "DONE" event to the job: this event is later used by reduce tasks to determine which map tasks finished and which node manager would serve the resulting map output file.
- **Shuffle Handler:** Each node manager runs an auxiliary service called shuffle handler, which serves map output files on request. The shuffle handler is responsible for handling requests from reduce tasks and sending the map output segment over a network stream. The shuffle handler does not require any notifications from Map tasks: it is configured to parse the request from reduce tasks to the local folder for the application of the task, where the output files reside.
- **Reduce Task:** Reduce tasks are launched as soon as a configurable percentage of map tasks finish so that the transfer of intermediate data is staggered and the network is not throttled

at the end of the Map phase. A reduce task is slightly more complicated than a map task: it involves fetching and merging data before the reducer can be applied. Typically, a reduce task has multiple threads allotted to fetch map output files from finished map tasks and a thread to fetch map completion events. All the fetched map output files are merged into a single file to be used as input to the reducer. The files can reside either in-memory or completely on disk. The reducer can then write the output directly to to the output filesystem. Figure 2.3 shows a simplified workflow for a reduce task.

2.4 Speculative Execution in Hadoop

Dean et al. [8] originally identified stragglers as machines that took longer times to execute some tasks in a MapReduce computation and added the provision of scheduling new executions of remaining tasks at the end of a phase. Zaharia et al. [29] observed that the progress of a task is not an accurate indicator in heterogeneous clusters and instead introduce the Longest Approximate Time to End (LATE) algorithm to deal with straggling tasks. The default speculator in Hadoop utilizes LATE to estimate the runtime of the task, based on statistical data for tasks in the same phase. Given the estimated end time, the default speculator evaluates a score for each task that estimates the overall average time gain between the estimated finish time of the task and the average task completion time, given historical task completion data. The speculator also accounts for tasks that are too new or are close to completion and avoids speculating on such tasks. Based on scores for all tasks in a phase, the speculator launches speculative tasks for tasks that can potentially be improved by a speculative execution. Additionally, to avoid overwhelming the cluster with speculative attempts, the speculator is limited by a cap on the number of speculative tasks it can launch, which depends on the number of tasks currently running and the overall number of tasks.

Formally, for task a, let t_a^* be the expected task runtime. Let $\bar{t}_a$ be the average time to completion for tasks in the same phase. LATE estimates the score score(a) as:

$$\text{score}(a) = t_a^* - \bar{t}_a$$

The estimated time of completion for the task attempt is derived from progress rates sent by tasks. Formally, each task periodically updates its progress p where, $0 \leq p \leq 1$. with the Application Master. While the progress indicator for map tasks is linearly dependent on the fraction of input data processed, the reduce task progress is segmented into three equal parts for the Shuffle, Merge and Reduce phases. One of the significant implications of this design is that reduce task progress is inherently not linear: the disk-heavy shuffle and merge stages are usually slower than the later

reduce phases.

Algorithm 1 describes the functioning of the speculator: new attempts are created if the task stands to benefit from a speculative execution, up to the speculation cap for tasks.

Hadoop handles contention between two attempts of a task by using a two-phase commit protocol. All task attempts work on temporary output paths. On completion, each task attempt tries to commit to the MapReduce application master, and on receiving a go-ahead, commits its output. The central point-of-decision allows the serialization of requests from task attempts. The MapReduce application master allows the first task attempt to report completion, to commit its output. All other attempts for the task are killed at this point.

An interesting aspect of speculative attempts is their priority with respect to other types of task attempts. In general, failed tasks or tasks yet to be scheduled are prioritized over speculative attempts. This ensures that the completion of tasks is not throttled by speculative attempts.

Algorithm 1 LATE speculator: Hadoop

A : set of all tasks
n : number of currently running speculations
specCap: max number of allowed running speculations
$a^* = \phi$: // best improvable task
$s_{a^*} = 0$: // score for best improvable task
for all $a \in A$ **do**
  if getScore(a) $> s_{a^*}$ **then**
    $a^* = a$
    s_{a^*} = getScore(a)
  end if
end for
if $n <$ specCap **then**
  schedule new speculative attempt a_1^*
  $n+=1$
end if
return

function getScore(TaskAttempt a)
  score $= t_a^* - \bar{t}_a$
  return score

2.5 Checkpoint-Restart

Checkpoint-Restart (C/R) allows users to create a snapshot of an application in progress and recreate the process, in case of errors/failures in the original process. Checkpoint-Restart techniques are of two types:

- **Application-level checkpoint-restart:** In case of application-level techniques, the applications are aware of the checkpoint and restart process. The workflow for this approach normally includes periodic snapshot creation and restoration on failure. Application-level checkpoint-restart techniques suffer from the disadvantage of requiring modifications to the application code. Scalable Checkpoint-Restart(SCR) [20] and Fault-Tolerance Interface [6] are examples of application-level checkpoint-restart libraries.
- **System-level checkpoint-restart:** System-level checkpoint-restart techniques snapshot the state of the process from outside the process' context, in the form of a process image. Typically, the process image contains that memory and execution context of the process, alongwith objects that represent the process' interaction with the system. The advantage of these techniques is that the application does not need to be modified in order to periodically snapshot and restart the process. Berkelely Linux Checkpoint-Restart (BLCR) [10], Linux-CR [16] and Checkpoint Restore In Userspace (CRIU) [7]

2.6 Checkpoint-Restore In Userspace

Checkpoint-Restore in Userspace (CRIU) [7] is a system-level checkpoint-restart utility for Linux. CRIU is widely used in OpenVZ [13], Docker [19] and LXC [18] for applications like container migration, snapshots and remote debugging. In addition to the generic checkpoint-restart semantics, CRIU supports features like live migration and diskless migration, which enable migration without using disk read-writes and hence eliminating an important bottleneck. Additionally, CRIU operates mostly in userspace (apart from a few extensions built into the Linux kernel since r3.11), therefore, improving the overall checkpoint-restart performance. Other system-level checkpoint-restart tools use kernel modules to dump the process state, which inherently creates a bottleneck for multiple checkpoint-restarts.

In the following subsections, we describe the major components of the CRIU toolchain.

2.6.1 CRIU Dump

This command dumps the entire process image as a sequence of files, either to disk or to a remote process. The process image includes information present in the procfs mount like:

- Memory mapped regions in the process memory,
- Open file descriptors, and
- Socket state (send/receive queue, connection status).

The process for dumping a process image is as follows:

- The tool utilizes ptrace $PTRACE\_SEIZE$ to stop all threads/tasks that comprise the target process, so that the threads cannot make any further changes to files or receive data from sockets.
- The tool then injects parasite code to dump the process memory image from within the process context.
- Once the tool has finished dumping the process image, the tool removes the **parasite code** and either resumes the process or leaves it in a stopped state.
- Since the process was seized using ptrace and not put in a stopped state while the checkpoint is taken, the process is unaware of any changes in state. Any side-effects from the checkpoint process like timeouts are considered as normal execution paths. Therefore, it is important for the original process to be resilient to error conditions that may be introduced while the checkpoint is being created.

The dump command supports the capability of dumping the process state to an external process, directly over the network, bypassing the bottleneck of disk writes.

2.6.2 CRIU Restore

Given a process image, the command restores the process and resumes execution from the point that the process was last executing at. The restore tool creates child tasks by forking and recreating the memory image of the original process on to the tasks.

CHAPTER

3

RELATED WORK

Speculative execution has been one of the most well researched approaches to straggler mitigation. Dean et al. [8] first coined the term straggler for slow tasks and introduced backup executions for late in-progress tasks at the end of the phase, the first attempt towards speculative execution. Zaharia et al. [29] formalized the speculative algorithm for heterogeneous clusters by using task completion times instead of progress rates. Ananthanarayanan et al. [1] further extended the scheduler to decide between a speculative task or a task restart by killing the original task and accounting for the network-aware placement of tasks. Zaharia et al. [30] experimented with contrasting fairness with resource locality in scheduling: essentially, delaying the execution of the job to ensure data locality. Ananthanarayanan et al. [2] built Dolly to enable speculative execution of all tasks in smaller sized jobs. Other reactive approaches to straggler mitigation include blacklisting [3] nodes with straggler tasks to prevent future straggler . Our approach complements the above methods by reducing the amount of redundant computation by speculative attempts.

Proactive approaches towards straggler mitigation focus on the causes of straggling. Yadwadkar et al. [28] built Wrangler, a statistical learning based system to detect situations where a task may straggle and utilize delays in task scheduling to prevent straggling. Kwon et al. [14] evaluated data skew as a cause for stragglers and implemented [15] a solution to partition tasks with unbalanced data. Another closely related direction of research is the comparative benefit of straggling tasks

versus lower accuracy for approximate jobs. Ananthanarayanan et al.[4] explored the possibility of trimming the results of straggler tasks in approximation jobs, and developed a model for trading accuracy and speculation overhead.

Li et al. [17] proposed the utilization of checkpoint-restarts to improve preemptive scheduling in shared clusters. While the work is similar in the context of utilization of checkpoint-restarts to improve performance, our work focuses on performance improvements by using cloning semantics for speculative execution. To the best of our knowledge, this is the first attempt to utilize system level checkpoint-restart for improving reactive speculative execution within Hadoop.

CHAPTER

4

DESIGN

4.1 Cloning Semantics

While prior texts have discussed task clones in the context of speculative attempts, we disambiguate the term for discussion for the remainder of our work. We define the following terms:

- **New Speculative Execution:** a speculative task attempt that begins from scratch.
- **Cloned Speculative Execution:** a speculative task attempt that has been cloned from the straggling task and resumes execution from the point the straggling task was snapshotted.

Algorithm 2 Remote Clone

Require: Nodes n_1, n_2, Process p on n_1
n_1 : Stall execution of p
$n_1 \rightarrow n_2$: Transfer the process image of p.
$n_1 \rightarrow n_2$: Synchronize external state of p.
n_1 : Resume p
n_2 : Restart p' from process image of p

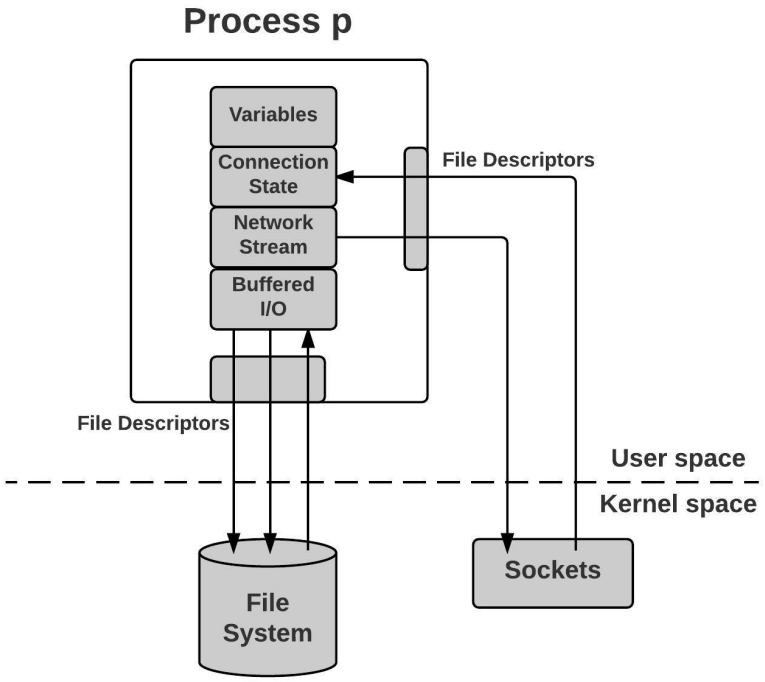


Figure 4.1 Process Resources.

We interchangeably use the terms speculations, speculative executions and task attempts. Algorithm 2 briefly outlines the remote cloning algorithm: the process p is first stalled in order to prevent in-line changes to the process state in the middle of cloning. Then, a snapshot of the state of p is transferred to a remote node in the form of a process image. While the process is still in a stalled state, the external state of the process is synchronized between the two nodes and the process is allowed to proceed. Finally, on the remote node, the process image is recreated by a restart tool and executed.

However, both the processes now contain common state that needs to diverge when execution continues. For example, if the processes were writing to the same networked location, one of the processes needs to recreate the current file state and use the new file for future references. Therefore, in addition to system-level cloning semantics, the application needs to be aware of the cloning process in order to duplicate objects/recover from errors.

The following subsections broadly sketch the design of the cloning semantics and the challenges associated with maintaining consistency across both the parent process and the cloned process.

4.2 Process resources

The process image does not completely encapsulate its execution state. During the course of execution, a process may write to files and communicate with other processes/devices. For example, the representation of file descriptors in the process memory is in the form of indices to files open in the kernel. Similarly, sockets have partial state in the kernel.

We define a **process resource** as any component that contains transient state as a side-effect of its execution. Further, we can classify resources into the following types, depending on whether the resource is accessible outside the process context:

- **Internal resources:** Components that are completely internal to the process, e.g., all variables within the process. Internal resources are completely contained within the process image. Therefore, a snapshot of the process image at any time would be consistent in all of its internal resources.
- **External resources:** Components that are observable outside the process context, e.g., file descriptors, sockets, devices. These components may additionally have transient state outside of the process due to caching or in-progress communications.
- **Compound resources:** Some components require additional internal state, on top of external resources, to provide consistency guarantees across error conditions, e.g., network data streams, remote database connectors (which may utilize sockets).

Figure 4.1 shows the resources of a process and their state. Considering that the process executes on its components, maintaining consistency across all components of a process translates to consistency in the execution of the process. The following subsection describes how we deal with transient state in external and compound resources.

4.3 Transient state of process resources

Resources with partial state within the kernel pose consistency problems during cloning. For example, when a process has written to a file descriptor, it updates the file descriptor offsets. However, the data it has written may not be flushed to disk: the kernel may cache writes and the data might still be in the kernel page cache. Therefore, any intermediate copy of the file would be missing the data that has not been flushed yet. Similarly, if the process is cloned in the middle of communicating with another process, the socket may contain packets sequences that have been put on the outgoing queue and assumed to have been sent. We tackle the problems using the following methods:

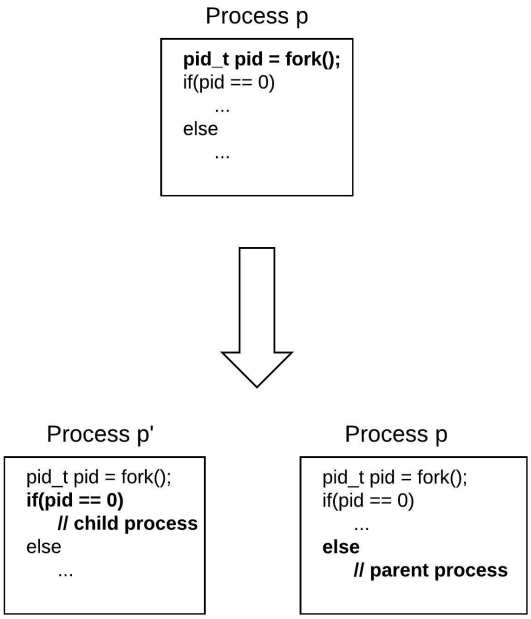


Figure 4.2 fork() system call: The return value of fork() discriminates between the execution paths for the child and parent processes.

- **Synchronize transient state:** In case of open file descriptors, it is easier to flush the transient state to disk using system calls like fsync() or sync(). This synchronizes all writes to disk. Therefore, any copy of a file after synchronization is guaranteed to be consistent.
- **Piggyback on recovery mechanisms:** In case of sockets, it might be easier to simply restore all sockets in a closed state, instead of recreating the send queue. The application then would attempt to recover from a closed socket/connection and, after reopening it, retry sending packets that it has not received an acknowledgement for.

4.4 File clone set

A process may require to read local file data during the course of its execution. When the process is cloned, the set of files needs to be copied over as well to ensure consistency of execution. We define the **file clone set** as the set of local files that the process will require in the course of its execution. The file clone set of a process is one of the major resource bottlenecks associated with cloning: if a

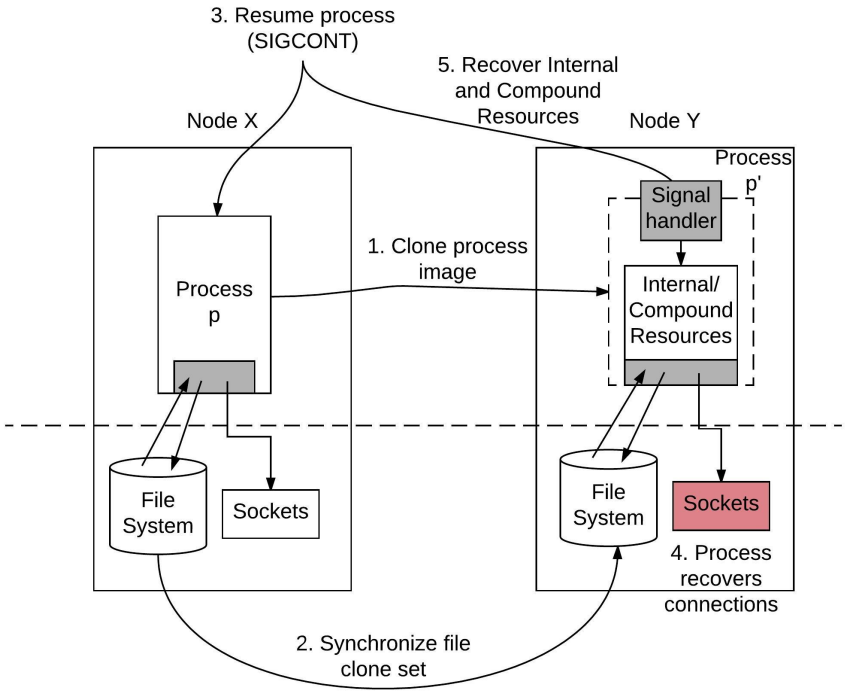


Figure 4.3 Cloning design: Since the cloning algorithm works outside the context of the cloned process, we use a signal handler to modify the cloned process' state.

process requires a large number of local files to execute, the cloning process would depend on how quickly the files can be replicated across the network. We discuss several modifications to alleviate this bottleneck in the implementation section.

4.5 Divergent execution path

Figure 4.2 describes how the fork() system call works: In a synchronous call to fork(), a new duplicate process is created. When the fork() system call returns, its return value can be used as a discriminant between divergent execution paths between the parent and the child processes.

Our design for the remote process clone closely resembles the semantics of the fork() system call. However, instead of a synchronous function call from within the process context, we design the utility to clone a process asynchronously from outside its context. Additionally, since the cloning process is external to the process, we install a signal handler on the process to synchronize the divergent execution path of the cloned process. Figure 4.3 outlines the cloning design for a process.

We use the signal handler to modify the cloned process' execution path to diverge from the original process, if required. For example, the cloned process might be sharing internal resources, like task identifiers or connections to a remote process. On resuming execution, the original process continues to utilize its resources as is. However, the cloned process now holds references to resources that are currently in use by the original process. Therefore, we utilize a UNIX signal for continuing execution, SIGCONT, to synchronize recovery for the resources associated with the cloned process: Resuming the process triggers the signal handler, which in turn triggers the recovery mechanisms.

For all internal resources, all changes can be made atomically. However, compound resources may have more complex recovery mechanisms. Therefore, we utilize repair threads that can be signaled on resuming the cloned process. These threads asynchronously recover compound resources. Once the threads have finished updates, they re-enter the idle state and wait for any further recovery signals.

One of the interesting side-effects of recovering resources asynchronously using signal handlers is that synchronization of recovery with the signal does not enforce ordering or consistency of all resources: the cloned process may create a new file with old identifiers before the signal handler can change the identifier. Therefore, we add consistency checks when the resources are next used and update the resources accordingly.

Another interesting effect is that the duration of cloning may require the recovery of some of the resources of the original process. For example, a process may time-out on communications with another process, if the cloning time exceeds the time limit sending or receiving data or if the connection to a remote process is broken in the middle of a clone. It is, therefore, important to introduce recovery paths in the original task as well, along with indicators that can signal whether an error was a side-effect of cloning or a regular failure. We discuss more details about the semantics of such a recovery mechanism in the implementation section.

CHAPTER

5

IMPLEMENTATION

We have implemented the proposed approach on top of Apache Hadoop v2.7.2 in three layers:

- A generic command-line tool, which wraps around CRIU to encapsulate the complete cloning process (including synchronization of file system and external resources);
- An integration to Apache YARN to detect and synchronize internal resources on cloning; and
- Optimizations to the MapReduce framework that alleviate network transfer bottlenecks associated with cloning files.

5.1 Remote clone

Remote clone is a command-line tool, which uses environment variables to configure the process and associated clone file set to be cloned. Remote clone uses existing UNIX command line tools to facilitate cloning and is essentially a wrapper on top of CRIU. Figure 5.1 shows the control flow of the tool. In addition to the simple cloning semantics, we use the following to maintain consistency and optimize network transfers.

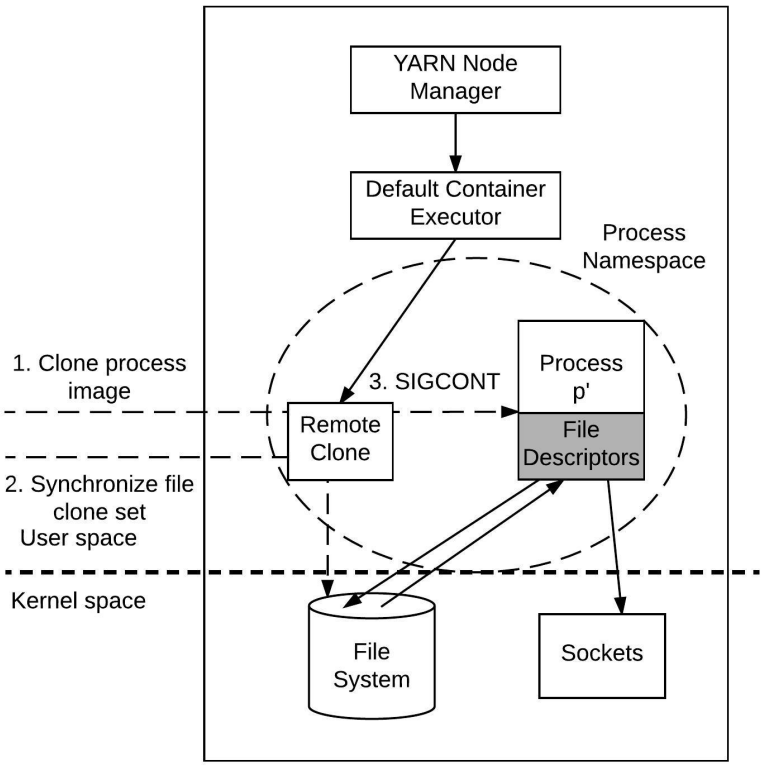


Figure 5.1 Integration of remote clone with Apache YARN Container Executor

- **File clone set synchronization:** While the data written by a process may have been flushed from its internal buffers, the kernel does not synchronously write the data to disk. Instead, the write is scheduled to be written at a later time. While the asynchronous disk write speeds up the write response time for processes, the scheduled writes have consistency implications for our usecase: there might be writes scheduled that are not picked up on cloning the file clone set. Therefore, once the process image has been extracted, we use the 'sync' command to synchronize any scheduled writes to disk. An alternative to using system-level sync is to use fsync calls on every file descriptor open for the process. However, each call to fsync would require a context switch to kernel mode and the latency for synchronizing disk writes would add up if the number of open files is large.

- **Disk I/O bottleneck:** Instead of writing the process image to disk and reading the image back during restoration, we directly write the process image into a RAM disk. This alleviates the disk I/O bottleneck for the process image transfer.

- **PID mismatch:** An existing process on the node may utilize the same task or process ids, which stops the restoration of the process image. We, therefore, restore the process in a new PID namespace.

- **External resources:** To fix external resources, we use sed/awk to fix identifiers in the process image before restoring. For example, consider a process that outputs its results to $id.out$. We modify the process image to change the file descriptor to instead point to $new\_id.out$ before restoring the process image.

5.2 Apache YARN integration

Figure 5.1 describes the integration of remote copy with the Apache YARN container executor. Each task is launched as a YARN child. We, therefore, register a Java signal handler for the YARN child, which handles SIGCONT in a new, high priority thread. By default, on instantiation, the YARN child fetches a task to execute from Task Attempt Listener and registers itself for providing further updates. We add mock registration calls for the newly cloned JVM and re-instantiate task parameters like local/log directories and configuration maps that are used by other classes. Additionally, we atomically modify the TaskAttemptId to ensure that all future references to the attempt ID reflect the new attempt.

While the signal handler can synchronize changes to global task variables synchronously, some higher-level constructs are difficult to access across the class hierarchies in Java. Instead, we use

Algorithm 3 updateYarnChild()

Require: TaskAttemptId new, old
Ensure: new $\neq$ old
  // Atomically update taskAttemptId
  old.update(new.getId())
  // Reinitialize Task setup
  configureLocalDirs()
  // Reinitialize Job specific configurations
  configureTask()
  // Reset connection with TaskAttemptListener
  resetUmbilical()
  self.repairTaskResources()

Algorithm 4 repairTaskResources()

  // Update/Repair Compound Resources
Require: CompoundResources R
  for Resource $r \in R$ **do**
    r.signalRepairThread()
  end for

condition variables to launch repair threads for compound resources. Algorithm 3 shows the implementation of the signal handler function.

5.3 Optimizations in the MapReduce framework

As discussed in the previous section, one of the major resource bottlenecks in cloning is the volume of data required to be transferred. The following section now discusses modifications to the MapReduce framework to facilitate faster cloning and to complement the existing speculation algorithm. We propose CloneHadoop: a modified version of Hadoop v2.7.2 that has been optimized to take advantage of cloned speculations.

5.3.1 Task types and clone file set

The characteristics of the two task types are distinct in terms of the clone file set:

- **Map Task:** Map tasks are essentially generators: they sequentially iterate over a series of key-value pairs and generate subsequent key-value pairs. The output of map tasks is initially stored into spill files. At the end of the map task, these spill files are merged into a single file.

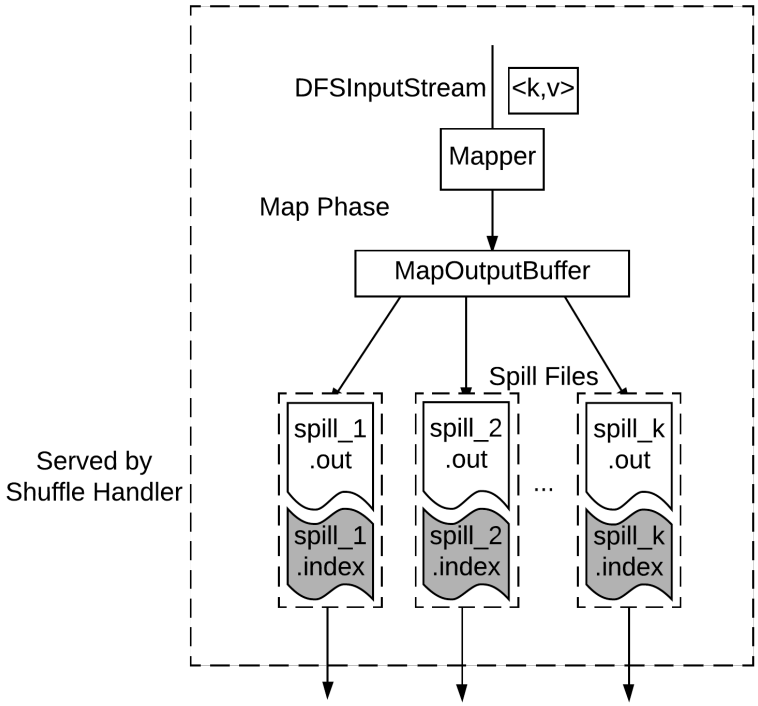


Figure 5.2 Map Task workflow in CloneHadoop: instead of combining all map spill files at the end of the map task, each spill file is served by the Shuffle Handler.

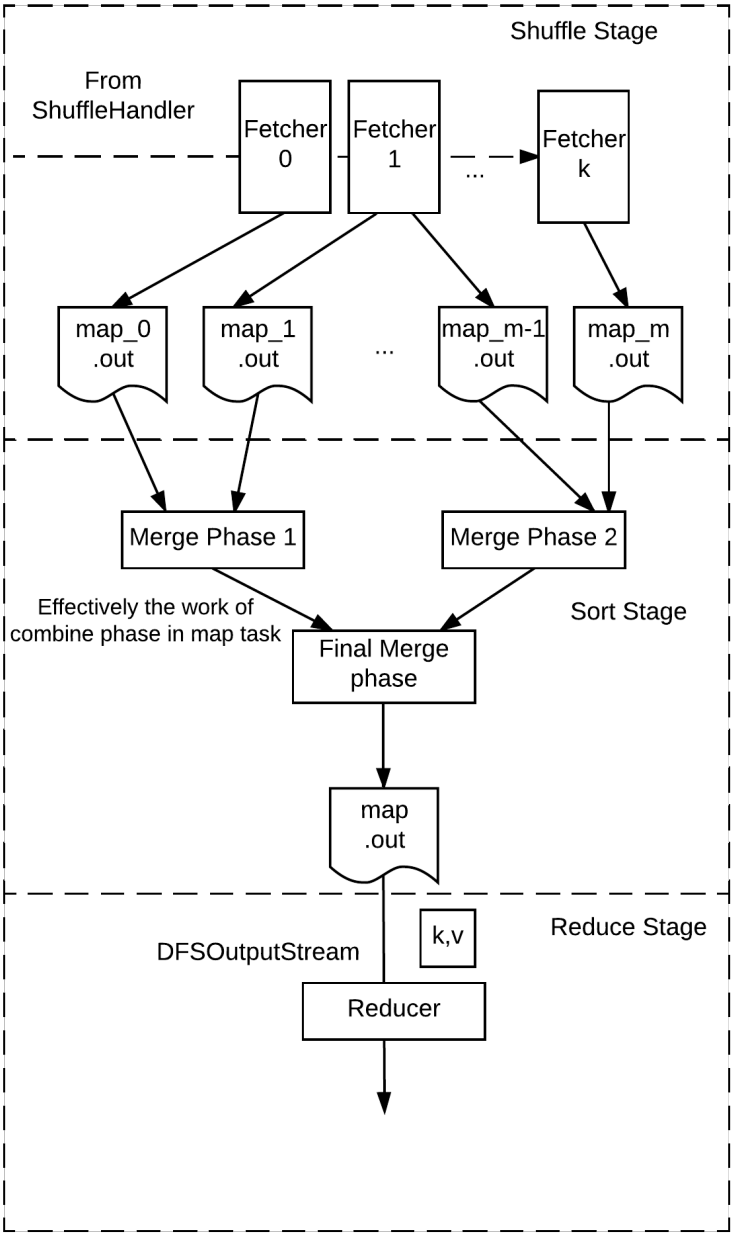


Figure 5.3 Reduce Task workflow in CloneHadoop: since the reduce task now fetches spill files instead of map output files, the reduce task may have to undergo several phase of running the combiner.

Subsequently, the node manager now has the responsibility to transfer the file to any reduce task that asks for the file. Apart from the final merge, the map task, in general, does not need to re-read any spill file it has already generated.

- **Reduce Task:** Reduce tasks are responsible for fetching all files from the node managers (shuffle), merging the files (sort) and then running the reducer on the subsequent input. Apart from the reduce stage, the clone file set of a reduce task is the complete set of files it has generated.

We, therefore, propose the following optimization: instead of combining all spill files at the end of a map task and notifying the job that the task is complete, we now send notifications per spill file. As with task attempts, the spill file done notification is evaluated per task attempt on a first-come, first-serve basis. Additionally, the ShuffleHandler service now serves individual spill files per task attempt, instead of a single output file. This has the added side effect of staggering network transfers over the life of each Map task, instead of higher network utilization, with potential bottlenecks, at the end of each map task. Therefore, the clone file set of a map task is now comprised of the last spill file it has generated. This significantly reduces the overhead of cloning a map task to nearly constant time, since the maximum spill file size is set to 80% of the map output buffer.

While the optimization is helpful in reducing the file clone set for map tasks, the effect on reduce tasks is minimal, since the reduce tasks require all of the intermediate data to proceed. Therefore, the clone time for a reduce task remains dependent on the volume of intermediate data the task has generated. However, the optimization results in a change in the reduce task: instead of fetching one file per task, the reduce task fetches a spill file from whichever task attempt finishes the spill and notifies the job first. Figure 5.2 and 5.3 show the overall changes in the workflow of Map and Reduce tasks.

5.3.2 Compound resource recovery: DFSOutputStream

After the merge stage, reduce tasks iterate over key-value pairs and output the reducer output to HDFS. The HDFS Output stream is constructed on top of DFSClient and communicates with the Namenode and Datanodes for writing files in chunks of dfs.block.size (by default 128 MB). However, on the cloned task, the output stream has to be repaired since:

- the path it points to is still actively appended to by the original task;
- the client name that holds the file lease now has to be modified; and

Algorithm 5 DFSOutputStream Recovery

Require: newTaskAttemptId, src, blockCount
Ensure: taskAttemptId $\neq$ newTaskAttemptId
```
// Stop sending more data packets
dataStreamer.stop()
// Retry all in-progress packets in current (un-synced) block
packetQ.addFront(inProgressQ)
inProgressQ.empty()
// Retry all done packets in current (un-synced) block
packetQ.addFront(doneQ)
doneQ.empty()
// Update file name
src.replace(taskAttemptId, newTaskAttemptId)
// Create new DFS file from existing file (upto last block)
f = dfsClient.newFileView(src, blockCount)
dataStreamer = f.append()
dataStreamer.start()
```

- the stream object is in an inconsistent state: it can never know if any of the pending data packets, that it has not received acknowledgment of, have been processed or not.

Additionally, the partial state of the output stream resides in HDFS: the stream may be in the process of writing data to the path it points to. Therefore, we need mechanisms to duplicate the state of the stream at the HDFS Namenode.

The DFSOutputStream comprises of two daemon threads: the DataStreamer and the ResponseProcessor. The DataStreamer is responsible for getting block locations and streaming data to all Datanodes. The ResponseProcessor parses responses from the Datanodes and maintains the state of the stream.

While cloning, on the cloned process, the stream is in an inconsistent state: while it may have sent packets within a block, the number of packets that have been acknowledged is non-deterministic. Further, the original process will continue to append to the same stream: the cloned process has to take care to not interrupt the file lease or try to append to it.

To recover from this state, we utilize a repair thread that waits on a condition variable. Once the cloned task resumes and handles a SIGCONT, we signal the condition variable and wake up the repair thread that fixes the OutputStream state. Additionally, we add triggers to recovery in the error detection segments of the OutputStream for distinguishing between three cases:

- The stream throws errors before the SIGCONT can trigger recovery;

- If the cloning process takes too long, the stream residing in the original process may time out and raise an exception; or
- The stream actually encounters an error that is unrelated to cloning.

We added filesystem hints in the cloning semantics: on reaching an error condition, the Output stream checks the current task state: if the task is a parent process that has been cloned, we assume that the error was associated with the cloning process and allow the stream to continue by clearing the error state. If the task has been recently cloned, we duplicate the stream state on the name node and start new DataStreamer and ResponseProcessor threads for the new block. In case of a regular error, we do not modify the erroneous state of the resource and start regular recovery mechanisms.

Additionally, in order to enable duplication of the stream, we added functionality in the Namenode to create a new file from the blocks of an existing file, up to the last completed block. Further, we have added reference counts to HDFS blocks to ensure that the blocks are not deleted if the original file is deleted while the other file is preparing to append to the stream. Finally, we open each HDFS output stream with a SYNC_BLOCK flag to ensure that once a block has been written, the data will be synced to disk on each Datanode.

Algorithm 5 shows the functionality of the recovery thread.

5.3.3 Speculation Algorithm

The semantics of cloning introduce subtle differences in the speculation algorithm design. While the default speculator is a best-effort attempt at improving performance, cloning a process at any stage may incur costs to the runtime of the task: since the task is in a frozen state during the checkpointing process, all progress is stalled till resumption. Therefore, the design of a cloning-based speculator has to minimize the chances of a false positive, i.e., a task set to finish soon identified as a straggler. Additionally, not all tasks may benefit from a cloning based approach: smaller sized tasks may even have run-times smaller than the overall clone time. Therefore, the cloning speculator balances the choice of a new speculative attempt versus a cloned speculative attempt.

Formally, for task a, let $\bar{t}_a$ be the expected task completion time and $\bar{t}_{vm}$ be the average JVM cloning time. Let $\bar{B}$ be the average disk-to-disk copy speed and f_{clone} be the file clone set size. Given task progress $p \in (0,1)$ and estimated time of completion t_a^*, we define the cloning time t_{clone}:

$$t_{\text{clone}} = \beta * (\bar{t}_{vm} + \frac{f_{\text{clone}}}{\bar{B}})$$

We add a damping factor β, which acts as a multiplier to the estimated cloning time for cloned

speculations: this allows us to overestimate the cloning time and prevent extraneous cloned speculations. Therefore, the completion time for cloned speculations for task a, t_a^{cloned} is given by:

$$t_a^{\text{cloned}} = t_{\text{clone}} + (1-p)\bar{t}_a$$

In comparison, the time taken to complete a new speculation t_a^{new} is given by:

$$t_a^{\text{new}} = \bar{t}_a$$

The speculation score for the combined speculation algorithm is given by:

$$\text{cloneScore}(a) = t_a^* - \min(t_a^{\text{cloned}}, t_a^{\text{new}})$$

The cloning-enabled speculator compares the overall benefits of a new speculation with respect to a cloned speculation. Since the empirical values for t_{vm} and $\bar{B}$ are independent of the actual application, we determine these values empirically from experiments on the cluster. Algorithm 6 shows the modified cloning speculator.

Algorithm 6 Cloning-enabled LATE speculator: Hadoop

A : set of all task attempts
n : number of running speculations
specCap: max number of running speculations
a^* : best improvable attempt
type_{a^*} : speculation type for best attempt
$s_{a^*} = 0$: score for best attempt
for all $a \in A$ **do**
  $< s_a, \text{type}_a > = \text{getCloneScore(a)}$
  if $s_a > s_{a^*}$ **then**
    $\text{type}_{a^*} = \text{type}_a$
    $s_{a^*} = s_a$
  end if
end for
if $n <$ specCap **then**
  if $\text{type}_{a^*} ==$ CLONE **then**
    schedule clone speculative attempt a'
  else
    schedule new speculative attempt a'
  end if
  $n+ = 1$
end if
return

function getCloneScore(TaskAttempt a)
  $\bar{t}_a$: expected task attempt runtime
  $\bar{t}_{vm}$: average JVM clone time
  f_{clone}: clone file set size
  $\bar{B}$: average disk-to-disk copy speed
  p : current progress of attempt a
  t_a^* : expected completion time of attempt a
$t_{\text{clone}} = \beta * (\bar{t}_{vm} + \frac{f_{\text{clone}}}{\bar{B}})$
$t_a^{\text{cloned}} = t_{\text{clone}} + (1-p)\bar{t}_a$
$t_a^{\text{new}} = \bar{t}_a$
if $t_a^{\text{cloned}} < t_a^{\text{new}}$ **then**
  return $< t_a^* - t_a^{\text{cloned}}, \text{CLONE} >$
else
  return $< t_a^* - t_a^{\text{new}}, \text{NEW} >$
end if

CHAPTER

6

EXPERIMENTAL FRAMEWORK

6.1 Platform

We evaluate the performance and sensitivity of the cloning speculator on a homogeneous cluster of 64 nodes. Each node comprises of an AMD Opteron 6128 processor with 16 cores, 2.0GHz, 4MB L2 cache and 1TB SATA hard-disk. The nodes in the cluster are connected via a gigabit ethernet link. All nodes run CentOS 7. We build our prototype on top of Apache Hadoop v2.7.2 and use the baseline version for comparison. Additionally, we built CRIU from source (v3.0) and added the capability to dump sockets in a closed state. Table 6.1 summarizes the hardware and other software configurations of the cluster. Additionally, we use the baseline Hadoop configuration listed in Table

Table 6.1 Cluster software configuration

OS	Centos 7
Kernel	4.10.13
jre	1.8.0_121-b13
jdk	1.8.0_121
CRIU	v3.0
Hadoop	v2.7.2

Table 6.2 Hadoop configuration

DFS Blocksize	128 MB
Speculation Cap	0.1
Overall Speculation Cap	1
JVM heap size	768 MB
io.sort.factor	100
io.sort.mb	500

6.2 for characterizing the sensitivity of our solution.

6.2 Workload

HiBench [11] is a benchmark workload suite by Intel characteristic of the performance of bigdata engines like Hadoop and Spark. HiBench comprises of workloads spanning a large range of applications, including problems that utilize libraries built on top of Hadoop like Hive, Mahout etc. In addition to characterizing per-job response times, HiBench additionally logs the system load for each node in the cluster with respect to CPU, network and memory utilization. HiBench also generates random data for use by each application. We use the following problems to characterize our cloning speculator:

- **Sort**: Given input text data, the Sort workload sorts the lines in the dataset lexicographically.
- **WordCount**: The workload collates all words in the input data into word-count pairs.
- **TeraSort**: The TeraSort [21] workload is similar to the Sort workload, but adds the capability for scaling Sort to multiple reducers. The workload utilizes a custom partitioner to split the reduce input into non-overlapping, ordered segments for each reducer. Therefore, the combined output of all the reducers in order is sorted globally.
- **PageRank**: The workload benchmarks the PageRank algorithm in Pegasus 2.0, which is built on top of Hadoop.
- **Scan, Join, Aggregate**: The workload tests the performance of Apache Hive queries.
- **Nutch Indexing**: The workload characterizes the search indexing algorithm in Apache Nutch.
- **Bayesian Classification, K-means clustering**: The workload characterizes machine learning workloads present in Apache Mahout.

With applications ranging from SQL, graph algorithms, search indexing and machine learning, HiBench enables us to test our work on a representative set of problems at scales ranging from a few megabytes to 100s of gigabytes.

CHAPTER

7

EXPERIMENTAL ANALYSIS

In this chapter, we characterize the performance of our work on a sample problem and evaluate its sensitivity to experimental parameters. We perform this analysis by slowing down varying numbers of tasks and observing the effects on job runtime. We use a 128 GB Terasort dataset and analyze the resulting difference in performance of both the cloning and the compound speculators, with variance in task sizes and distribution of stragglers. The following subsections explain our experimental methodology:

7.1 Straggler simulation

In order to simulate stragglers, we randomly added delays to the core execution paths of each task, proportional to their runtime. Therefore, if the regular run-time of a task is given by t_{rt}, we add a proportional delay, t_d such that:

$$t_d \propto t_{rt}$$

We use a log-normal distribution to select the proportional constant. In general, as the scale is decreased to zero, the mean and median of the distribution converge. Figure 7.1 shows the probability density function of the lognormal distribution with changes to the scale parameter.

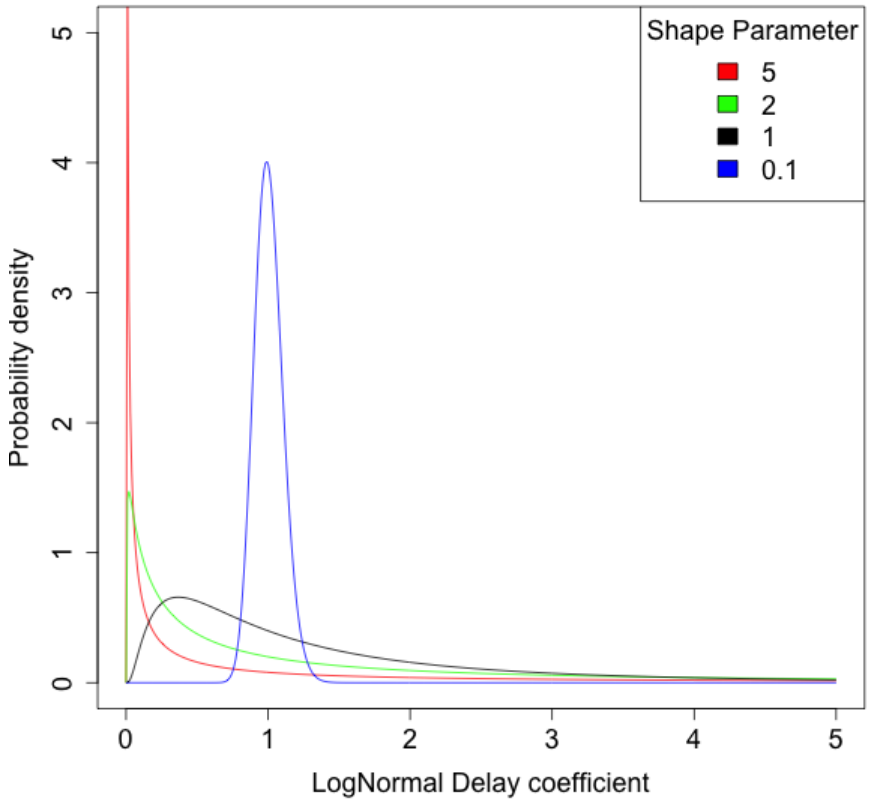


Figure 7.1 Lognormal distribution and its variation with changes to shape: the varying tail characteristics allow us to test the sensitivity of our algorithm in different situations

Using the lognormal distribution allows us to test the sensitivity of our approach by varying the degree and number of stragglers: for higher values of scale (e.g. the red plot in figure 7.1), the stragglers would have limited progress by the time the first tasks finish, whereas smaller values of scale (e.g. blue, black plots in figure 7.1) denote tasks that are slower than the original task but are making some progress as well.

7.2 Cluster parameter tuning

The network-related parameters are important components of the speculation algorithm: underestimating these parameters would lead to missed speculation opportunities, whereas overestimating these parameters can lead to cloning for tasks that are almost complete and indirectly delay the overall job completion time. We run a sample of 20 random sized Terasort jobs, with varying delays and task sizes, and collected cloning statistics from the jobs. While the jobs represent a single application, the cloning process is application agnostic and the data collections simply serves to tune the cloning speed parameters.

The results show a relatively stable JVM clone time $\bar{t}_{vm}$ and average disk-to-disk transfer speeds $\bar{B}$. We observe that the average disk-to-disk clone speed is close to 40 MB per second. This is nearly a third of the overall network bandwidth. Two effects seem to explain this behavior: the presence of slow disks acts as a bottleneck for the overall file transfer process, and the cloning process occurs in the middle of a workload, which explains why the speeds are so far from the optimal values. Our experimental results show the JVM clone for a task takes approximately 15 seconds. Additionally, we use a baseline of $\beta = 1.2$ and log-normal distribution with $shape = 2$ for our experiments. This allows us to characterize the performance of our solution with respect to other parameters.

7.3 Job Analysis

We compared the performance and characteristic runtimes of the speculators on a 128GB Terasort problem with 16 map tasks and 64 reduce tasks, i.e., 4 GB of data per map task and 2 GB per reduce task. Additionally, we analyzed the runtime of the same job over multiple runs and observed minute variations in runtime: For the probem we are analyzing, we observed a standard deviation of no more than 1% in job response times for all experiments.

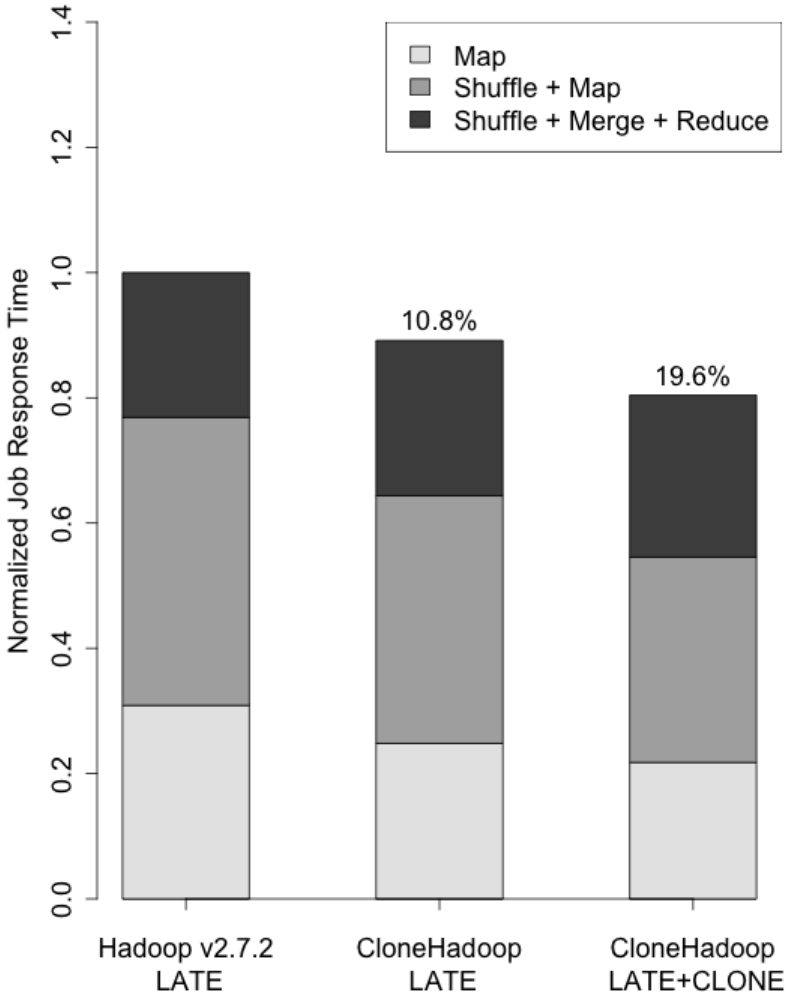
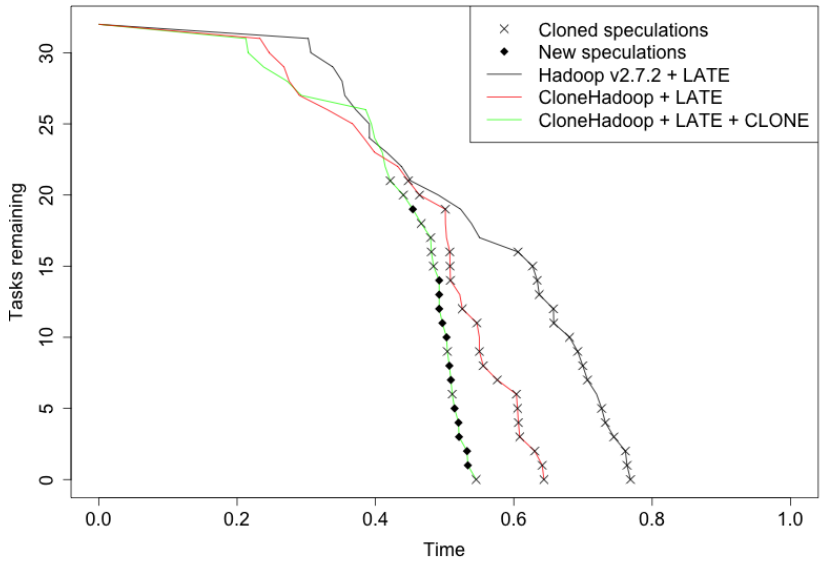
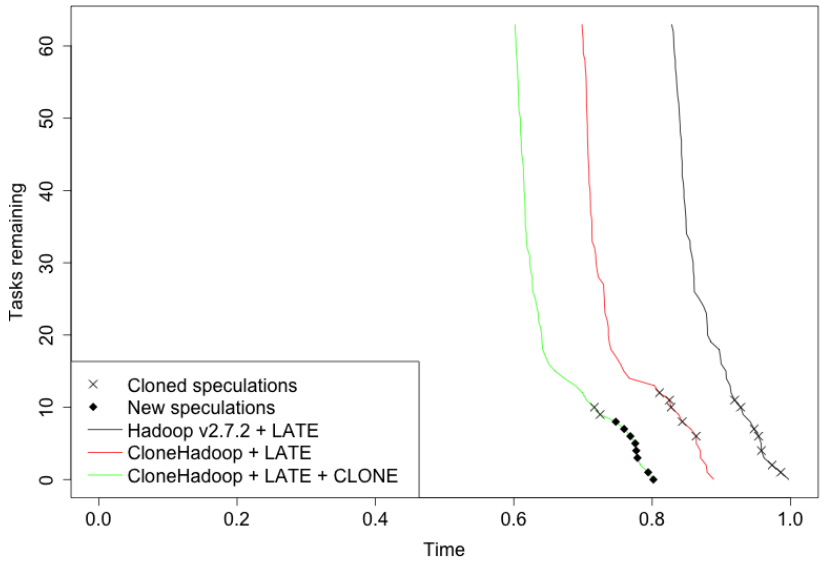


Figure 7.2 Job Runtime Analysis: $\beta = 1.2, shape = 2$: Comparison of the normalized job runtime for a Terasort problem (128GB, 32 map tasks, 64 reduce tasks). We observed a standard deviation of not more than 1% in job response times. Since the task runtime distribution is short-tailed, cloning speculations improve the overall job run-time: nearly 11% for CloneHadoop and 20% for the cloning speculator.



(a) Map Phase Completion: Map phase for the cloning-enabled speculator finishes earlier than LATE speculators



(b) Reduce Phase Completion: Reduce phase completion for the cloning-enabled speculator is staggered over a longer interval. However, since the reduce tasks start completing earlier, the overall effect on job response time is negligible

Figure 7.3 Phase Completion Characteristics: $\beta = 1.2, shape = 2$: Comparison of normalized task completion times over the duration of a job.

7.3.1 Runtime Analysis

We analyzed the beginning and end of MapReduce phases for a Terasort job and characterize the performance improvement with respect to CloneHadoop and cloned speculations. Figure 7.2 shows the runtime for jobs normalized to the baseline Hadoop version (y-axis) under the different implementations (x-axis) over different phases (stacked on y-axis). The baseline implementation of CloneHadoop performs significantly better than the original Hadoop implementation. The performance is significantly improved by using a cloning-based speculator. We observe the following:

Observation 1: The map phase finishes earlier for CloneHadoop.
Once a specified percentage of map tasks complete (default: 5%), the Application Master launches reduce tasks that can prefetch map output files instead of waiting for all map tasks to finish. We observe that the launch of reduce tasks occurs earlier in CloneHadoop. The lack of a combiner phase helps in speeding up the completion of the map tasks. Additionally, the reduce task can buffer spill files till the last map task finishes. This staggers the overall network load on the cluster.

Observation 2: Adding cloning capabilities improves the performance of LATE in CloneHadoop.
Adding cloning capabilities improves map task completion times: Straggling tasks benefit from cloned speculation attempts. Cloning improves the phase completion times for map and reduce phases in CloneHadoop by around 11%. The improvement comes from tasks that are straggling but have enough progress to make cloning worthwhile.

Observation 3: Performance improvements for early phases amplify improvements for later phases.
The additive effect of removing the map task combiner and adding cloned speculations allows map tasks to finish significantly faster in CloneHadoop. This has a cascading effect on:

- Scheduling of reduce tasks plus
- the map phase end (and subsequent merge/reduce phase synchronization).

The cascade effect amplifies performance gains by translating the improvements into an earlier startup of subsequent phases.

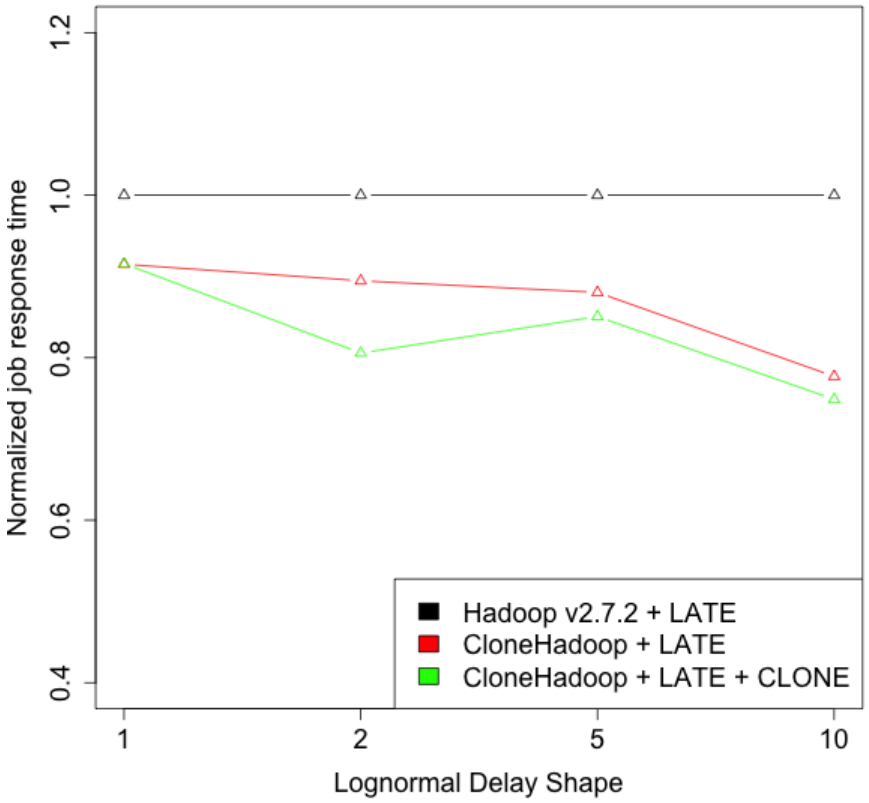


Figure 7.4 Straggler Distribution analysis: $\beta = 1.2$: Comparison of the normalized job response time for varying straggler distribution shapes on a Terasort problem of size 128GB.

7.3.2 Task Completion Statistics

We measured the task completion times in each phase for a Terasort job and compare the characteristics of phase completion for each version of Hadoop. Figures 7.3(a) and 7.3(b) show the number of remaining tasks (y-axis) over time (x-axis) and the distribution of speculations for map and reduce phases (indicated by ♦ and ×) for CloneHadoop with LATE and with cloning-enabled LATE on a Terasort problem of size 128GB. The implicit barrier between the end of the map phase and beginning of the reduce phases ensures that the task completion for all reduce tasks in Figure 7.3(b) occur after the completion of the map phase in Figure 7.3(a).

Observation 4: New speculations cluster at the beginning of the task completion graph whereas cloned speculations cluster near phase completion.
This effect can be explained by the fact that new speculations are more likely in early stages since some tasks may not have made any significant progress. However, later speculations are likely to be cloned speculations if the tasks have made sufficient progress.

Observation 5: The addition of cloning speculations increases the number of successful speculations.
Figures 7.37.3a and 7.37.3b show a relative increase in the number of successful speculations, while the reference implementation has fewer successful map speculations and no successful reduce speculations. This can be explained in terms of the speculation cap: Once a new speculation has started, it will occupy a slot from the speculation cap until it finishes or fails. In comparison, cloned speculations take less time to complete since they continue from an initial state. Therefore, the faster completion times for cloned speculations free up slots in the speculation cap early and may result in faster speculations for subsequent straggling tasks.

7.4 Sensitivity Analysis

In this section, we characterize our speculation algorithm with respect to sensitivity to task size and the straggler distribution:

7.4.1 Sensitivity to straggler distribution

We parameterized the log-normal delay shape for this experiment and use it to benchmark job response times normalized to the baseline Hadoop version (y-axis) for different straggler distributions

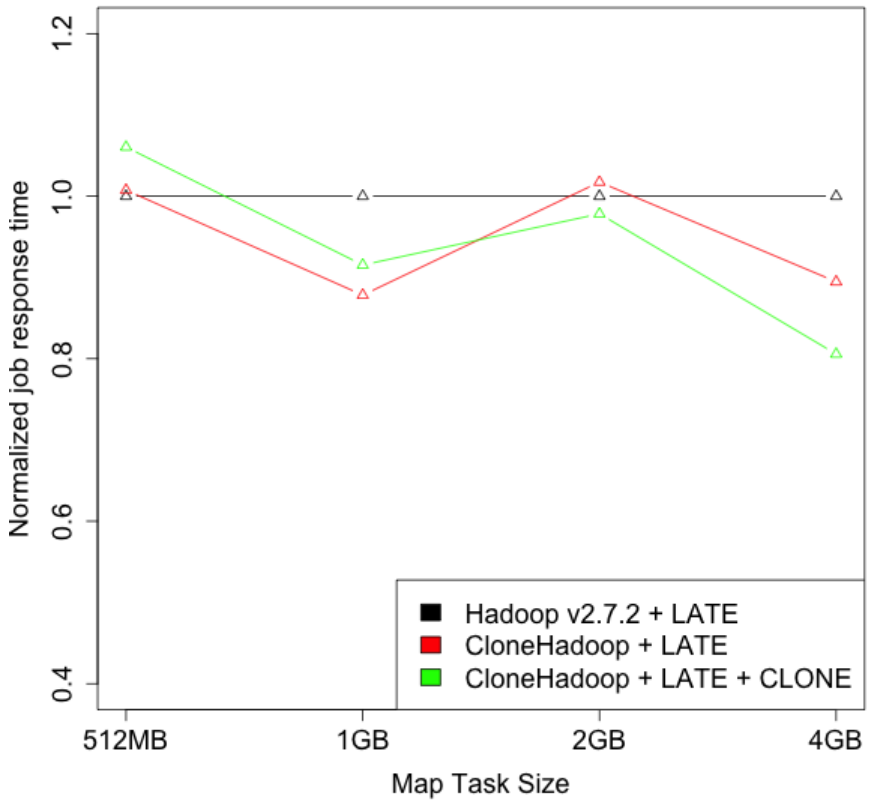


Figure 7.5 Task size scaling: $\beta = 1.2, shape = 2$: Comparison of the normalized job response time for different map task sizes on a Terasort problem of size 128GB and map tasks of size 4GB.

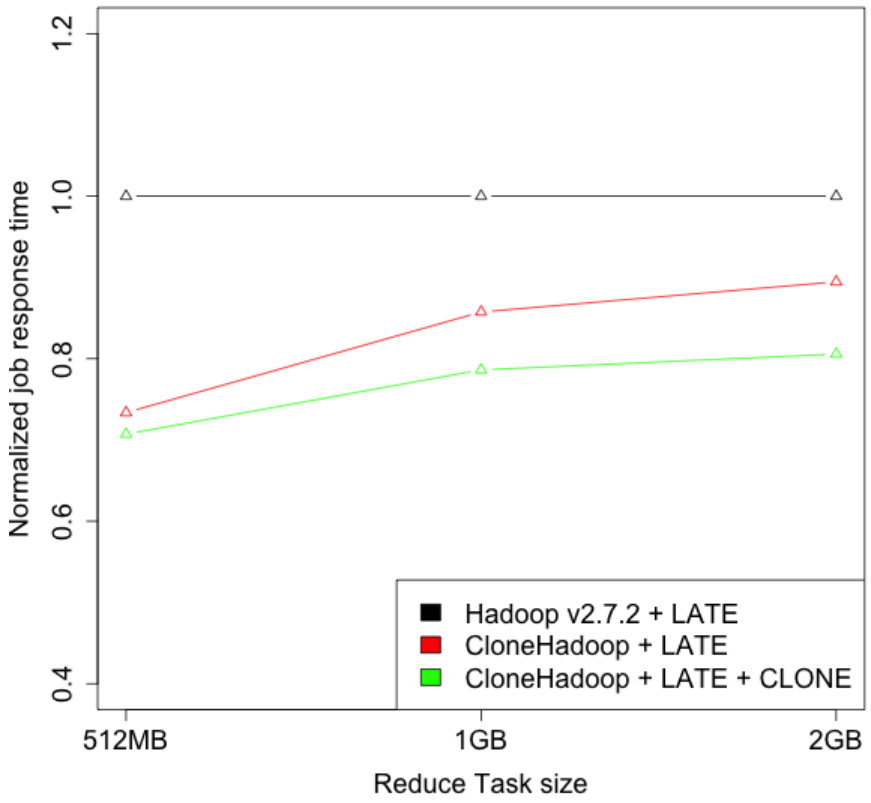


Figure 7.6 Task size scaling: $\beta = 1.2, shape = 2$: Comparison of the normalized job response time for different reduce task sizes on a Terasort problem of size 128GB and reduce tasks of size 2GB.

(x-axis) in Figure 7.4.

Observation 6: Cloned speculations work better for short-tail distributions and complement new speculations in LATE.
We observe nearly no improvement from cloned speculations in regular distributions (shape = 1) and in long tailed distributions (shape = 10), but the highest performance gains are observed in relatively short-tailed distributions (shape = 2). The intuition behind this result is that in long-tailed distributions, most of the tasks have not made progress and, therefore, cloning does not improve performance significantly. Similarly, for regular distributions of task run-times, there are limited opportunities for improvement. However, for short-tailed distributions, most tasks make sufficient progress and, therefore, any speculation opportunity improves the overall performance. Clone-Hadoop with cloned speculations improves performance by upto 25%.

7.4.2 Sensitivity to task sizes:

We vary the number of tasks for the Terasort problem to measure the impact of problem size on cloned speculations and CloneHadoop. We observe the changes in normalized job response times for CloneHadoop and cloning-enhanced LATE on CloneHadoop compared to the reference Hadoop implementation with respect to changes in the number of tasks, keeping the problem size constant.

Observation 7: Smaller map tasks experience performance degradations under cloned speculations.
Figure 7.5 measures the difference in normalized performance (y-axis) with change in map task sizes (x-axis). We observe a performance degradation for jobs with map tasks of size 512MB. In fact, the cloning-enabled speculator performs worse than the reference Hadoop baseline for map tasks of size 512MB. This can be explained by the relative lack of cloning opportunities for smaller tasks. Map tasks of smaller sizes tend to finish fast. Therefore, any attempt to clone a task would delay the overall completion time. However, we observe a direct correlation of performance improvement with map task size: As map task size increases, cloned speculations become more competitive and directly improve performance. The outlier data point for 2GB map tasks is still unexplained and is subject to future work.

Observation 8: Larger reduce tasks improve performance under cloned speculations.
Figure 7.6 shows the variation of normalized job response time with increase in reduce task size.

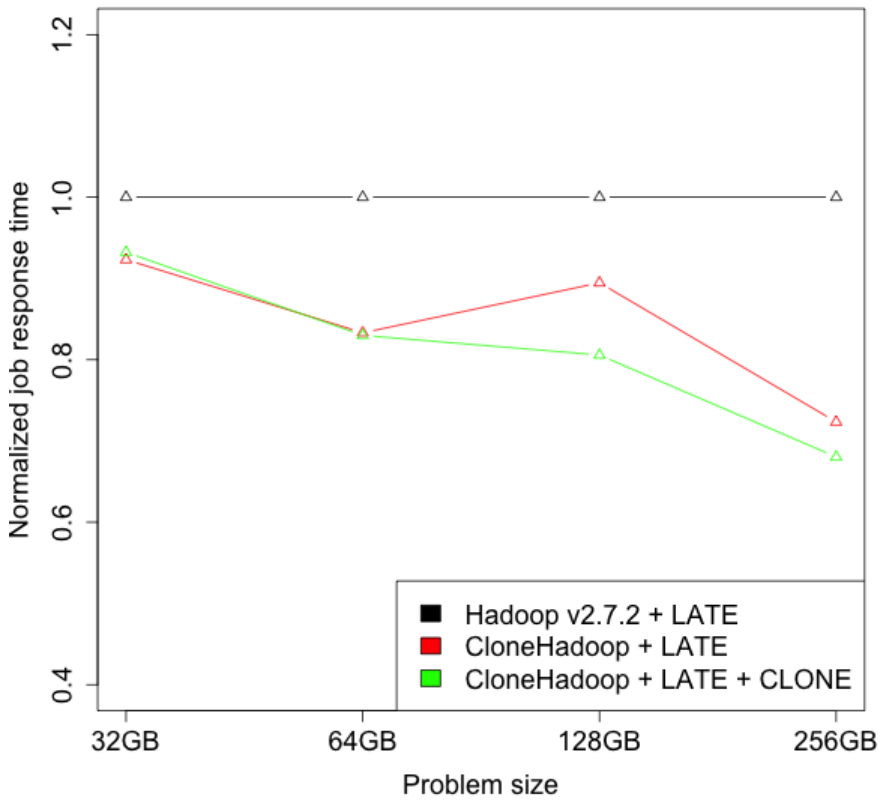


Figure 7.7 Problem Scaling: $\beta = 1.2, shape = 2$: Comparison of the normalized job response time for Terasort jobs of different sizes, with map and reduce tasks configured to work on 4GB and 2GB of data, respectively. Cloned speculations improve performance with increase in problem size.

Table 7.1 Application Parameters

Application	Dataset
Sort	128 GB file data
Terasort	128 GB file data
Wordcount	128 GB file data
PageRank	30 million pages
Kmeans	200 million points, 20 dimensions, 5 clusters

With an increase in reduce task size, the performance of CloneHadoop tends towards the reference Hadoop version: Larger task sizes effectively counteract any gains by the reduce task by effectively adding more merge phases. However, the increase in task size also increases the probability of cloned speculations: we observe this as a direct performance improvement in performance compared to CloneHadoop with LATE speculator.

Observation 9: Larger task sizes scale well.
In general, we observe performance improvements for larger task sizes with cloned speculations, even for reduce tasks whose file clone set increases with increase in task size. While new reduce speculations are launched faster than cloned reduce speculations, a new speculation has to fetch spill files from all node managers. The combined size of all spill files is similar in size to the file clone set for the cloned speculation, but with the added overhead of sending requests to each node manager. Therefore, cloned speculations perform well even on large reduce tasks.

7.5 Problem scaling

We analyzed the scalability of the speculation algorithm with different problem sizes. We measured normalized job response times for Terasort problems, ranging from 32 GB to 256 GB, assigning map tasks and reduce tasks 4 GB and 2 GB of data respectively. Figure 7.7 shows the comparison of normalized job response times with job sizes.

Observation 10: Performance improves with larger job sizes.
We observe that the cloning-enabled speculator performs better for larger problem sizes. For smaller sized problems, the limited number of tasks limit the capability of the cloning speculator to improve performance.

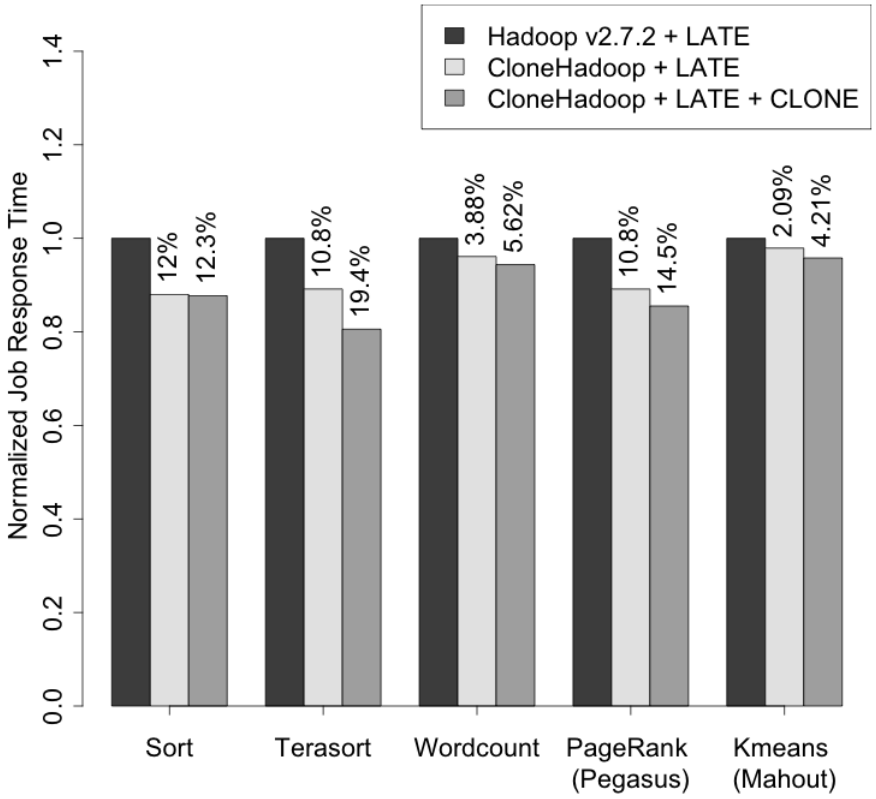


Figure 7.8 Application Sensitivity: $\beta = 1.2, shape = 2$: Comparison of the normalized job response time for different application types.

7.6 Application sensitivity

We analyze the performance of our prototype on several large problems from the HiBench benchmark suite, including applications built on top of Hadoop. Table 7.1 describes the problem sizes for each application. Figure 7.8 shows the performance of our prototype normalized to the baseline Hadoop version (y-axis) for different applications (x-axis).

Observation 11: Applications optimized to minimize intermediate data show lower performance benefits.
In contrast to Terasort and Sort, the volume of intermediate data for WordCount problems is low. Since the distribution of the input data set is derived using a power law equation, the scale of intermediate data is typically low. This results in fast reduce phases, which lowers the benefits of using our prototype. Similarly, the KMeans application is structured to use a single reducer per job. The relatively smaller intermediate and final output sizes lower the overall speculation opportunities within a job, which explains the smaller benefits in job response times for the application.

Observation 12: Multi-phased applications benefit equally from cloned speculations.
PageRank and K-means problem sets typically take multiple job iterations over the course of their execution. Figure 7.8 shows similar performance benefits for multi-job applications over a longer time periods. The normalized job response plot shows that the cumulative effect of improvements on multi-job shows the average performance improvement of the speculative algorithm is similar to single job improvements.

CHAPTER

8

LIMITATIONS AND FUTURE WORK

Cloned speculations improve the overall job response times. However, one of the major limitations of using system-level checkpoints is that the cloning algorithm can only work between machines that share the same architecture. This limits the application of our work into heterogeneous clusters without the use of underlying virtualization techniques.

Cloned speculations open up interesting areas of further research that may be useful for tuning the performance of the speculation algorithm. Specifically:

- **Data locality and rack-aware speculation:** While our work focuses on the semantics of cloning, the performance of the speculator as-is may degrade in large clusters spanning multiple racks. We rely upon the capability of Hadoop to schedule containers to account for data locality. However, for nodes with high latency network paths, it might be more feasible to launch new speculations than to clone speculations across a slow network path. Additionally, there might be a tradeoff between maintaining data locality and speeding of the cloning process.

- **Automated configuration:** While our experiments were based on values tuned on a cluster, the overall characteristic of these values may be different under different load conditions. For example, the bandwidth for cloning might reduce if a large number of network-bound processes are already running on the node. Similarly, while the transfer semantics for the

cloning algorithm are same, the network speed may differ between inter-rack and intra-rack process cloning. By collating throughput data from different Node Managers, we can model the network parameters as a function of the nodes that might participate in cloning and, therefore, reduce the likelihood of false positives.

CHAPTER

9

CONCLUSION

Cloning speculations take advantage of scenarios where task progress is significant enough to continue execution from. Our results show that in most task runtime distributions, opportunities for cloning task exist that can improve job performance. Cloned speculations complement existing speculation strategies and do not adversely impact the job performance. Additionally, cloned speculations improve efficiency and increase the overall number of successful speculative attempts. Therefore, the results prove our hypothesis: *Speculative tasks have a higher probability of catching up to stragglers (and subsequently improving completion times) if they resume from current point of execution instead of recomputing work already done by the straggling task.*

BIBLIOGRAPHY

[1] Ananthanarayanan, G. et al. “Reining in the Outliers in Map-Reduce Clusters using Mantri.” *OSDI*. Vol. 10. 1. 2010, p. 24.

[2] Ananthanarayanan, G. et al. “Why let resources idle? Aggressive cloning of jobs with Dolly”. *Memory* **40**.60 (2012), p. 80.

[3] Ananthanarayanan, G. et al. “Effective Straggler Mitigation: Attack of the Clones.” *NSDI*. Vol. 13. 2013, pp. 185–198.

[4] Ananthanarayanan, G. et al. “Grass: Trimming stragglers in approximation analytics” (2014).

[5] Apache Software Foundation. *Hadoop*. Version 0.20.2. 19, 2010.

[6] Bautista-Gomez, L. et al. “FTI: high performance fault tolerance interface for hybrid systems”. *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. ACM. 2011, p. 32.

[7] *CRIU: Checkpoint Restore in Userspace*. Version 3. 2014.

[8] Dean, J. & Ghemawat, S. “MapReduce: Simplified Data Processing on Large Clusters”. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI’04. San Francisco, CA: USENIX Association, 2004, pp. 10–10.

[9] Ghemawat, S. et al. “The Google file system”. *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 29–43.

[10] Hargrove, P. H. & Duell, J. C. “Berkeley lab checkpoint/restart (blcr) for linux clusters”. *Journal of Physics: Conference Series*. Vol. 46. 1. IOP Publishing. 2006, p. 494.

[11] Huang, S. et al. “Hibench: A representative and comprehensive hadoop benchmark suite”. *Proc. ICDE Workshops*. 2010.

[12] Kang, U et al. “Pegasus: A peta-scale graph mining system implementation and observations”. *Data Mining, 2009. ICDM’09. Ninth IEEE International Conference on*. IEEE. 2009, pp. 229–238.

[13] Kolyshkin, K. “Virtualization in linux”. *White paper, OpenVZ* **3** (2006), p. 39.

[14] Kwon, Y. et al. “A study of skew in mapreduce applications”. *Open Cirrus Summit* **11** (2011).

[15] Kwon, Y. et al. “Skewtune: mitigating skew in mapreduce applications”. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 25–36.

[16] Laadan, O. & Hallyn, S. E. “Linux-CR: Transparent application checkpoint-restart in Linux”. *Linux Symposium*. Vol. 159. 2010.

[17] Li, J. et al. “Improving preemptive scheduling with application-transparent checkpointing in shared clusters”. *Proceedings of the 16th Annual Middleware Conference*. ACM. 2015, pp. 222–234.

[18] *Linux Containers*.

[19] Merkel, D. “Docker: lightweight linux containers for consistent development and deployment”. *Linux Journal* **2014**.239 (2014), p. 2.

[20] Moody, A. et al. *Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system*. Tech. rep. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2010.

[21] O’Malley, O. “Terabyte sort on apache hadoop”. *Yahoo, available online at: http://sortbenchmark.org/Yahoo-Hadoop. pdf,(May)* (2008), pp. 1–3.

[22] Owen, S et al. *Mahout in action. Greenwich, CT*. 2011.

[23] Ren, Z. et al. “Workload analysis, implications, and optimization on a production hadoop cluster: A case study on taobao”. *IEEE Transactions on Services Computing* **7**.2 (2014), pp. 307–321.

[24] Shvachko, K. et al. “The hadoop distributed file system”. *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE. 2010, pp. 1–10.

[25] Thusoo, A. et al. “Hive: a warehousing solution over a map-reduce framework”. *Proceedings of the VLDB Endowment* **2**.2 (2009), pp. 1626–1629.

[26] Vavilapalli, V. K. et al. “Apache hadoop yarn: Yet another resource negotiator”. *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[27] Yadwadkar, N. J. & Choi, W. “Proactive Straggler Avoidance using Machine Learning”. *White paper, University of Berkeley* (2012).

[28] Yadwadkar, N. J. et al. “Wrangler: Predictable and faster jobs using fewer resources”. *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–14.

[29] Zaharia, M. et al. “Improving MapReduce performance in heterogeneous environments.” *Osdi*. Vol. 8. 4. 2008, p. 7.

[30] Zaharia, M. et al. “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling”. *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 265–278.

[31] Zaharia, M. et al. “Spark: Cluster computing with working sets.” *HotCloud* **10**.10-10 (2010), p. 95.

[32] Zaharia, M. et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.