CloneHadoop: Process Cloning to Reduce Hadoop's Long Tail

Sarthak Kukreti, Frank Mueller
North Carolina State University, USA, mueller@cs.ncsu.edu

***Abstract*—Recent advances in distributed computing have enabled large-scale data processing on high volumes of data with MapReduce. However, the overall runtime of such applications is dependent on the slowest subtask at the tail end of the work queue. Current approaches mitigate this effect by scheduling redundant speculative attempts of straggler tasks. However, speculative attempts have to recompute work already done by the original task. This often prevents late speculations from completing before the straggling task.**

This work promotes a novel speculation approach via process cloning to avoid redundant computations transparent to users. But speculation requires consistency between task attempts and results in divergent execution of subtasks. The work contributes (1) a process cloning approach for speculative execution, (2) mechanisms to maintain consistency and recover complex components, and (3) an integration of cloning and recovery into Apache Hadoop with optimizations to alleviate resource bottlenecks. In experiments, cloning benefits straggling tasks with runtime reductions of up to 25% on jobs with larger tasks. Our solution scales with different task and problem sizes and is robust across different runtime distributions of straggler tasks.

***Keywords*-Mapreduce; Stragglers; Process Cloning**

I. 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 application of diverse domains have been adapted to exploit parallel computation capabilities of paradigms like MapReduce [1] and Spark [2], [3]. Additionally, the availability of open source frameworks based on these models [2], [4] and domain-specific libraries (Mahout [5], Hive [6], Pegasus [7]), 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 phases and independent subtasks scheduled in parallel. However, degradations in performance of a subtask can amplify and slow down the entire 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 [1], [8], blacklisting nodes where stragglers occur) and proactive (preemptively schedule multiple copies of tasks/jobs [9] or preventing stragglers in the first place [10], [11], [12]). However, reactive approaches require speculative tasks to repeat computations of the original task, and proactive techniques lead to high resource contention for larger jobs/task sizes: each task copy increases the overall job resources.

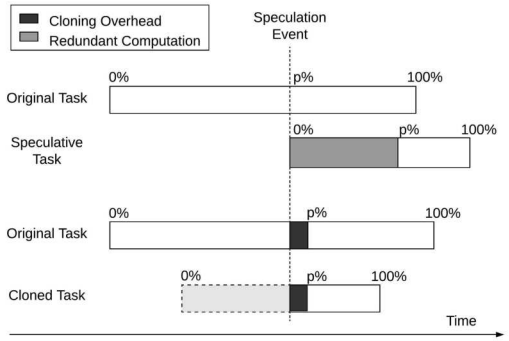


Fig. 1: A speculative task requires re-execution while cloning continues execution mid-task with small overheads.

Dean et al. [1] originally identified stragglers as tasks with disproportionately slow completion times that negatively impact the job response time and proposed speculative execution to mitigate the effect of stragglers. Ren et al. [13] 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 task speculations did not lead to significantly earlier task completion times compared 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.

Fig. 1 shows a sample task execution. At some progress percentage $p\%$, the task is judged to be a straggler and a new speculative task is spawned that re-executes the task's entire work. Even though the new speculative task is significantly faster than the original one, it still finishes after the original one. The time for 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.

This work shows that 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 improves reactive speculative execution of tasks by creating 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 state of a cloned task,
- recovering from errors arising due to inconsistent state, in both the original and the cloned task, and
- deriving a policy for when to create new or cloned tasks.

We have implemented our approach, CloneHadoop, on top of Apache as changes to Apache YARN [14] and the Hadoop Filesystem [15] to facilitate task cloning. We also propose changes to the structure of MapReduce computations implemented in Apache Hadoop to alleviate resource bottlenecks for process cloning. MapReduce applications transparently integrate into CloneHadoop without any changes.

Our results show that CloneHadoop with cloning speculation performs well for large-sized tasks and for the lower end of stragglers. Since the default speculator performs well on long-tailed straggler task runtime distributions, the two approaches complement each other. We observe performance benefits of up to 25% in large jobs without significant impact on performance for smaller jobs. Contrary to prior work, cloned speculation shows significant benefits for map tasks.

II. Related Work

Checkpoint-Restore in Userspace (CRIU) [16] is a system-level checkpoint-restart utility for Linux. CRIU supports live and diskless migrations, the latter of which, used in our work, eliminates the disk as a bottleneck. CRIU operates mostly in userspace to improve the overall checkpoint-restart performance. Other system-level checkpoint-restart tools [17], [18] use kernel modules to dump the process state, which inherently creates a bottleneck for multiple checkpoint-restarts.

Speculative execution has been one of the most well researched approaches to straggler mitigation. Dean et al. [1] 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. [19] formalized the speculative algorithm for heterogeneous clusters by using task completion times instead of progress rates in their Longest Approximate Time to End (LATE) algorithm. 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 $score(a) = t_a^* - \bar{t}_a$. 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. Speculative attempts are created if the task stands to benefit from a speculative execution, up to the speculation cap for tasks. Our work further enhances this predictor and integrates it with cloning.

Ananthanarayanan et al. [8] further extend 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. [10] experiment with contrasting fairness with resource locality in scheduling by delaying the execution of the job to ensure data locality. Ananthanarayanan et al. [9] build Dolly to enable speculative execution of all tasks in smaller sized jobs. Other reactive approaches to straggler mitigation include blacklisting [20] nodes with straggler tasks to prevent future stragglers. 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. [12] build 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. [21] evaluate data skew as a cause for stragglers and implement [22] a solution to partition tasks with unbalanced data. Also related is the comparative benefit of straggling tasks versus lower accuracy for approximate jobs. Ananthanarayanan et al. [23] explore the idea of trimming results of straggler tasks in approximation jobs, and developed a model for trading accuracy and speculation overhead.

Kawachiya et al. [24] create a vanilla JVM and then clone an existing JVM execution into it. Orthogonally, we clone a JVM at the process level in its live execution stage and run it on another node. Wang et al. [25] eliminates intermediate files to speed up MapReduce but thereby breaking fault tolerance of even conventional Hadoop, unlike our approach. Li et al. [26] utilizes checkpoint-restarts to improve preemptive scheduling in shared clusters. In contrast, our work focuses on performance improvements by using cloning semantics for speculative execution. Xu and Lau [27] initially study two policies, the Smart Cloning Algorithm (SCA) and the Straggler Detection Algorithm (SDA), for MapReduce workloads simulated in Matlab and find opportunities to reduce the job runtime by close to 60% for both depending on cluster load scenarios. They then contribute an Enhanced Speculative Execution (ESE) algorithm, which further improves runtime by 18%. However, their SCA does not clone tasks, it speculatively creates "extra copies of a task" at the time of original task creation. Our work differs in that we provide a Hadoop extension for true cloning with its full system complexity of socket and I/O redirection, we develop CloneScore as a realistic metric to choose between speculation and cloning instead of the model-based ESE approach, and we evaluate our system on actual hardware in a cluster instead of simulation.

III. Design

Cloning: Let us outline Clone Hadoop's remote cloning algorithm: A 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.

Semantically, the application needs to be aware of the cloning process in order to duplicate objects and recover from error codes due to interrupted system calls with challenges discussed in the following.

Process resources: The process image does not completely encapsulate its execution state. A process may write to files and communicate with other processes/devices. E.g., file descriptors are represented in the process memory as 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 accessible of the process context:

Internal resources: Components that are completely internal to the process, e.g., all variables within the process, 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, may additionally have transient state outside of the process due to caching or in-progress communications.

Compound resources: Some components require additional internal state to provide consistency guarantees across error conditions, e.g., network data streams, remote database connectors (which may utilize sockets).

Transient state of process resources: Resources with partial state within the kernel pose consistency problems during cloning. The kernel may cache file writes before flushing them to disk. A socket may contain packet sequences that have been put on the outgoing queue but have not yet been sent. CloneHadoop tackles these problems as follows:

Synchronize transient state: For open file descriptors, it is easier to flush the transient state (from writes) to disk using system calls (fsync()/sync()). Thus, a file copy is consistent after synchronization.

Piggyback on recovery mechanisms: Sockets can be restore all sockets in a closed state instead of recreating the send queue. The application then recovers from a closed socket/connection by reopening it and retrying to send packets that were not acknowledged.

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. CloneHadoop defines the *file clone set* as the set of local files that the process requires during execution. This set is a major resource bottleneck associated with cloning: if a process requires a large number of local files to execute, cloning depends on how quickly the files can be replicated across the network. We discuss several techniques to alleviate this bottleneck in Sec. IV. Notice that Hadoop utilizes

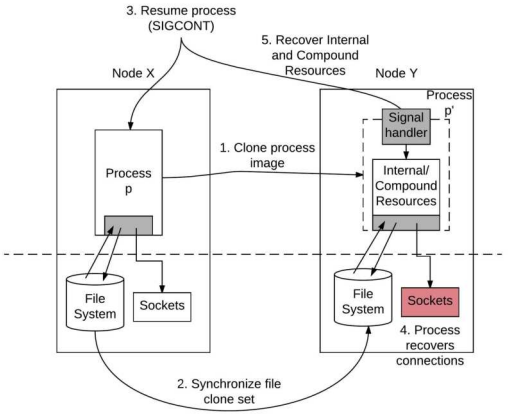


Fig. 2: Cloning: Signals cause process state modifications.

local filesystems due to their bandwidth and performance benefits instead of networked ones (NFS/UnionFS).

Divergent execution path: 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 clone a process asynchronously from outside its context and install a signal handler to synchronize the divergent execution path of the cloned process. Fig. 2 outlines our cloning design.

We use the signal handler to modify the cloned process' execution path to diverge from the original process, e.g., if the cloned process shares internal resources (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. We utilize a continue to resume execution and thereby 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, changes can be made atomically. However, compound resources require more complex recovery mechanisms. Thus, we utilize repair threads that are signaled on resuming the cloned process and then asynchronously recover compound resources. Once finished with updates, these threads become idle and wait for further recovery signals.

One of the interesting side-effects of recovering resources asynchronously 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.

Also, the duration of cloning may require the recovery of some resources of the original process. E.g., a process may time out on communications if the cloning time exceeds the time limit for sending/receiving data or if the connection to a remote process is broken during cloning. It is, thus, 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 as discussed next.

IV. Implementation

We have implemented our CloneHadoop on top of Apache Hadoop v2.7.2 in three layers: (1) A generic command-line tool, which wraps around CRIU to encapsulate the complete cloning process (including synchronization of file system and external resources); (2) An integration to Apache Yarn to detect and synchronize internal resources on cloning; and (3) Optimizations to MapReduce that alleviate network transfer bottlenecks associated with cloning files.

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. Fig. 3 shows the control flow of the tool. In addition to the simple cloning semantics, our CloneHadoop maintains consistency and optimizes network transfers as follows:

(a) *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 but may delay it. While the asynchronous disk write speeds up the write response time for processes, they inflicts consistency problems as writes may be 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 (as per-file fsyncs would result in high overhead due to many kernel calls).

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

(c) *PID mismatch:* An existing process on the target node of cloning may already utilize the same task or process ids. We, therefore, restore the process in a new PID namespace.

(d) *External resources:* To fix external resources, we use update identifier names in the process image before restoring. E.g., 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 it.

Yarn integration: Fig. 3 depicts the integration of remote copy within Yarn's container executor. Each task is launched as a Yarn child. By extending Hadoop's current signal capabilities, we register a Java handler for the Yarn child to respond to "continue" signals in a new, high priority thread. By default, on instantiation, the Yarn child fetches a task to execute from Task Attempt Listener (Hadoop's legacy handling of stragglers) and registers itself for providing further updates. Our CloneHadoop adds mock registration calls for the newly cloned JVM and re-instantiates task parameters like local/log directories and configuration maps that are used by other classes. Such a call is triggered when we clone the entire process with its JVM that represents a task.[1] Furthermore, we provide a redirection layer for sockets used by the Hadoop task manager, which is needed to switch from the old to the new task (inside the old and new processes, respectively, which includes a change on nodes). Inside the cloned task, we also 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 cannot be directly accessed without breaking software abstractions across Java class hierarchies. Hence, we bridge class hierarchies by using synchronization via condition variables to launch repair threads for compound resources.

Optimizations in the MapReduce framework: One of the major resource bottlenecks in cloning is the volume of data required to be transferred. Our CloneHadoop modifies the MapReduce framework to facilitate faster cloning and to complement the existing speculation algorithm in "CloneHadoop" as an enhancement to Hadoop v2.7.2.

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. The node manager then transfers this file to any requesting reduce task. Apart from the final merge, the map task 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.

Our CloneHadoop makes 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 (Fig. 4, red box skipped for CloneHadoop). 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 staggers network transfers over the life of each Map task and thus reduces network utilization as opposed to the traditional bottleneck at the end of each map task. The clone file set of a map task is now comprised of just the last spill file 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.

This reduces the file clone set significantly for map tasks but not much for reduce tasks, since the latter require all intermediate data to proceed. Therefore, the clone time for a reduce task remains dependent on the volume of intermediate data 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. This larger number

[1] One could clone just the JVM, but critical resources (.e.g, open files) need to be handled at the process level in any case. Furthermore, our approach generalizes beyond JVM (i.e., could clone native binaries inside a process). Linux containers only add additional overheads as they may carry extra libraries, yet do not solve the socket problems that we address.

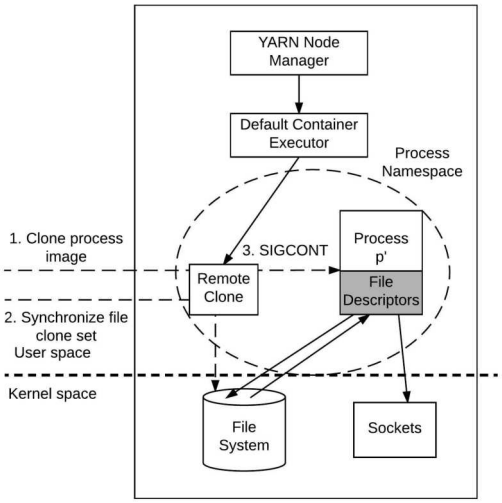


Fig. 3: Integration of remote clone into Yarn Container Executor

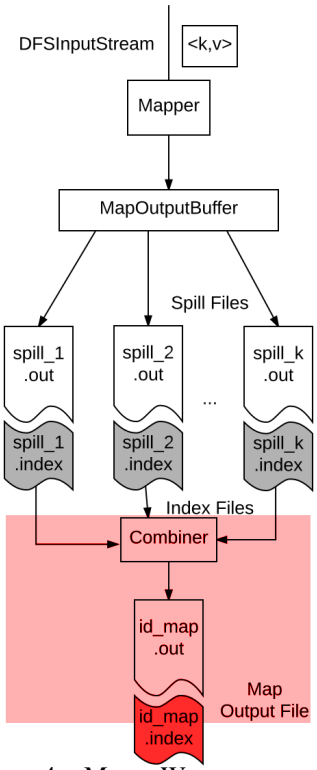


Fig. 4: Map: We remove spill-file combiners

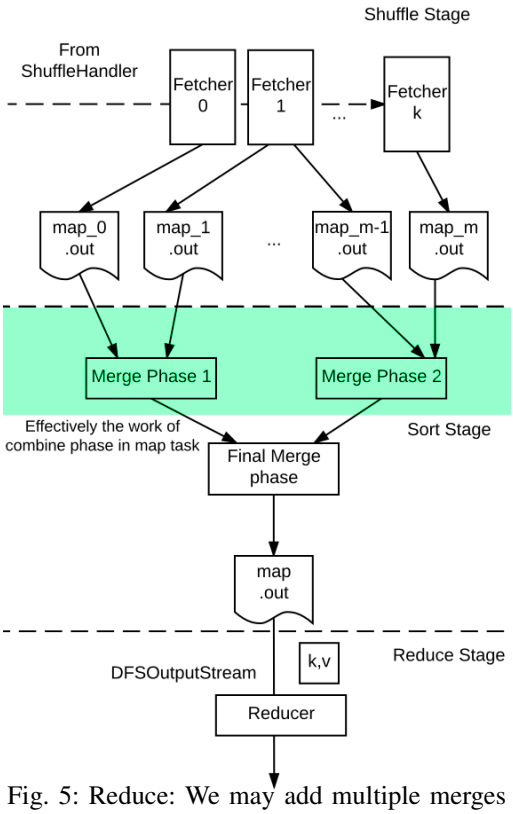


Fig. 5: Reduce: We may add multiple merges

of output files sometimes results in an additional merge stage (see Fig. 5, green box added for CloneHadoop) for configurations near the multi-stage threshold (see Sec. V).

Compound resource recovery: 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 128BM by fault (dfs.block.size). However, on the cloned task, the output stream has to be repaired since (1) the path it points to is still actively appended to by the original task; (2) the client name that holds the file lease now has to be modified; and (3) the stream object is in an inconsistent state: It is unknown if pending data packets without acknowledgment have been processed yet. Also, the partial state of the output stream resides in HDFS, and the stream may be in the process of writing data to the path it points to. Thus, we need mechanisms to duplicate the state of the stream at the Namenode.

The DFSOutputStream comprises of two daemon threads, the DataStreamer and the ResponseProcessor. The DataStreamer obtains block locations and streams 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 "continue" signal, we signal the condition variable and wake up the repair thread that fixes the OutputStream state. Additionally, we add triggers to the recovery in the error detection segments of the OutputStream for distinguishing between three cases: (1) The stream throws errors before the "continue" signal can trigger recovery; (2) if the cloning process takes too long, the stream of the original process may time out and raise an exception; or (3) the stream encounters an error 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. For 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.

Speculation Algorithm: The semantics of cloning intro-

Algorithm 1 Cloning-enabled LATE speculator: Hadoop

A : set of all task attempts
n : number of running speculations
$specCap$: max # 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**
  $< s_a, type_a >$ = getCloneScore(a)
  if $s_a > s_{a^*}$ **then**
    $type_{a^*} = type_a$
    $s_{a^*} = s_a$
  end if
end for
if $n < specCap$ **then**
  if $type_{a^*} == CLONE$ **then**
    schedule speculative clone a'
  else
    schedule new speculation 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 a
$t_{clone} = \beta * (\bar{t}_{vm} + \frac{f_{clone}}{\bar{B}})$
$t_a^{cloned} = t_{clone} + (1-p)\bar{t}_a$
$t_a^{new} = \bar{t}_a$
if $t_a^{cloned} < t_a^{new}$ **then**
  return $< t_a^* - t_a^{cloned}, CLONE >$
else
  return $< t_a^* - t_a^{new}, NEW >$
end if

duces 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 as 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} as

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

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

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

The time taken to complete a new speculation t_a^{new} is

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

The speculation score for the combined algorithm is

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

The cloning-enabled speculator compares the overall benefits of a new speculation 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. Algorithm 1 shows the modified cloning speculator.

V. Experimental Framework

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 CloneHadoop approach 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.[2] Table I summarizes the hardware and other software configurations of the cluster. We further use the baseline Hadoop configuration (Table II) for characterizing the sensitivity of our solution.

Workload: HiBench [28] 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* sorts the lines in the input dataset lexicographically.
- *WordCount* collates all words in the input data into word-count pairs.
- *TeraSort* [29] is similar to Sort, but adds the capability for scaling Sort to multiple reducers. It 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* benchmarks the PageRank algorithm in Pegasus 2.0, which is built on top of Hadoop.
- *Bayesian Classification, K-means clustering* characterizes machine learning workloads present in Apache Mahout.

Other codes do not follow Hadoop's coding guidelines, e.g., even though application-specific I/O is discouraged by Hadoop, Hive creates temporary files without Hadoop's task attempt id as keys. With applications ranging from sorting, graph algorithms, ranking and machine learning, HiBench enables us to test our work on a representative set of problems ranging from a few MBs to 100s of GBs. We chose problem and task sizes (detailed later) so that we could more clearly observe speculation trends on experimental parameters without secondary effects from over-provisioning, given the

[2] We use CRIU's default Ethernet link for cloning. Alternatively, one could use CRIU's Infiniband interposition plugin for higher bandwidth to speed up cloning even further.

Tab. I: 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

Tab. II: 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

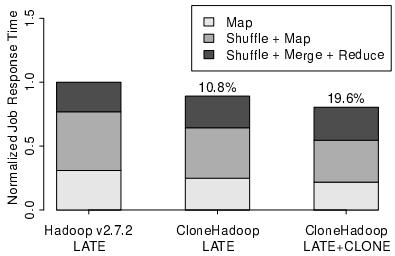


Fig. 6: Terasort: Runtime normalized to Hadoop

cluster size and network parameters. (If the cluster is already overprovisioned, speculation would not be effective anyhow).

VI. Experimental Analysis

We first assess the performance of our work on a sample problem and evaluate its sensitivity to experimental parameters. This is done 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.

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 runtime 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. 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, the stragglers would have limited progress by the time the first tasks finish, whereas smaller values of scale denote tasks that are slower than the original task but are making some progress as well.

Cluster parameter tuning: The network-related parameters are important for the speculation algorithm: Underestimating these parameters leads to missed speculation opportunities, overestimating them leads to cloning 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 collect cloning statistics from the jobs. While the jobs represent a single application, the cloning process is application agnostic and data collection allows us 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 40MB/sec. This is nearly a third of the theoretical network bandwidth or nearly half of a more realistic network bandwidth of 90MB in practice. Two effects explain this behavior, namely 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. 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.

Job Analysis: We compared the performance and runtimes of the speculators on a 128GB Terasort problem with 32 map and 64 reduce tasks, i.e., 4 GB of data per map and 2 GB per reduce task. [3] Additionally, we analyzed the runtime of the same job over multiple runs and observed minute variations in runtime: For the problem we are analyzing, we observed that the absolute value of standard deviation is no more than 1% of the mean job response times for all experiments, which range from 15 to 45 minutes.

Runtime Analysis: We analyze the beginning and end of MapReduce phases for a Terasort job and characterize the performance improvement with respect to CloneHadoop and cloned speculations. Fig. 6 shows the runtime for jobs normalized to the baseline Hadoop version (y-axis) under the different implementations over different phases (x-axis, stacked). 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 have completed (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 due to cloning amplify improvements for later phases.

The cumulative effect of removing the map task combiner

[3] We selected fewer map than reduce tasks because the latter are more I/O intensive compared to map tasks, especially with the burden of merging shifted onto the reduce task. Hence, larger map tasks compensate for the imbalance in overall time taken by map and reduce tasks.

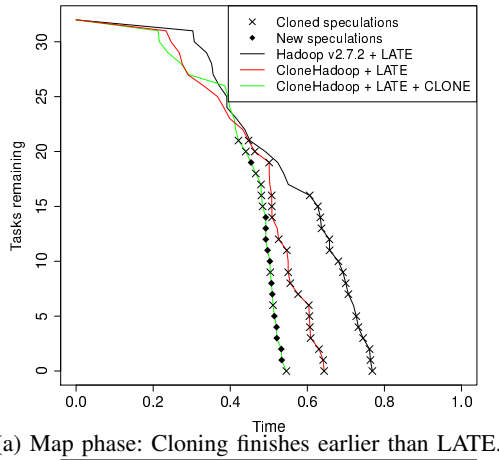
and adding cloned speculations allows map tasks to finish significantly faster in CloneHadoop. This then affects

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

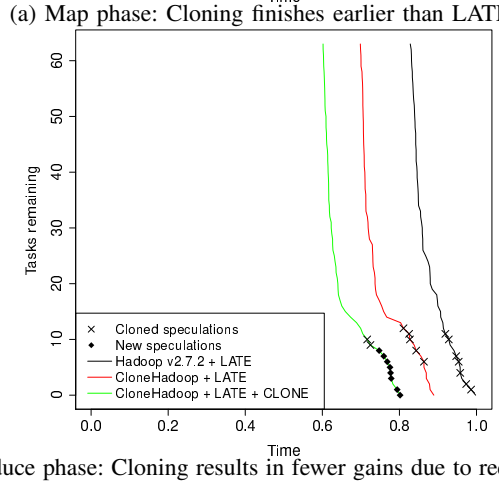
The cascade effect amplifies performance gains by translating improvements into an earlier startup of later phases.

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. Fig. 7a, continued in time in Fig. 7b, 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 task completion for all reduce tasks in Fig. 7b occurs after the completion of the map phase in Fig. 7a.

Observation 4: New speculations cluster at the beginning of the task completion whereas cloned speculations cluster near phase completion for both map and reduce phases.



(a) Map phase: Cloning finishes earlier than LATE.



(b) Reduce phase: Cloning results in fewer gains due to reduce tasks finish faster for Terasort.

Fig. 7: Phase Completion times of Terasort (same params)

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 (e.g., around 0.45 for map and 0.7 for reduce in Figures 7a and 7b, respectively). However, later speculations are likely to be cloned speculations if the tasks have made sufficient progress (after 0.65 and 0.9 for the same respective figures).

Observation 5: The addition of cloning speculations increases the number of successful speculations.

Figures 7a and 7b 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.

Sensitivity Analysis: We next characterize our speculation algorithm with respect to sensitivity to task size and the straggler distribution:

Sensitivity to straggler distribution:

We parameterized the log-normal delay shape for this experiment and benchmark normalized job response times (y-axis) with straggler distributions (x-axis) in Fig. 8.

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 symmetric distributions (shape=1) and in long tailed distributions (shape=10), but the highest performance gains are observed in relatively short-tailed distributions (shape=2). This is because 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 runtimes, 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.

Sensitivity to task sizes: We vary the number of tasks for Terasort 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.

Fig. 9 measures the difference in normalized performance (y-axis) over changing map task sizes (x-axis). We observe a slight performance degradation under cloning for small map tasks (512MB-1GB). For such small task sizes, the cloning time exceeds the overall task runtime of some tasks so that the original tasks finish first, but only after the cloning overhead had slowed them down. Larger map tasks (2-4GB) always require multi-phase merges, and we observe increasing performance improvements since cloned speculations become

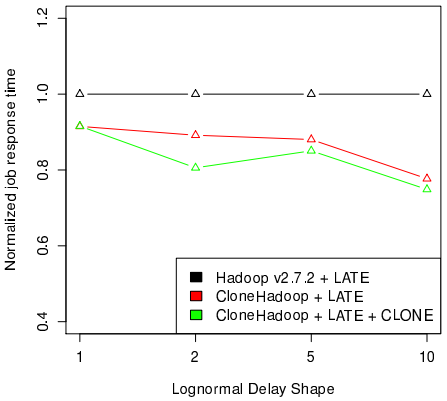


Fig. 8: Straggler distribution of runtimes for same Terasort parameters per shape

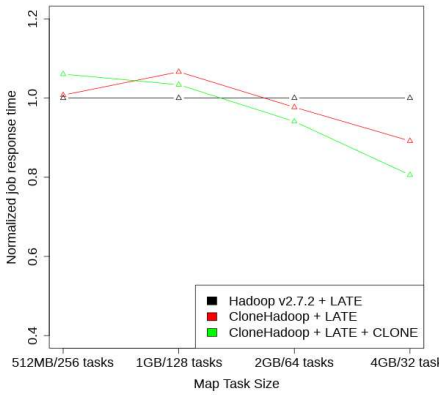


Fig. 9: Map task scaling: Runtimes for same Terasort params (128GB total size) over map tasks sizes of 0.5-4GB

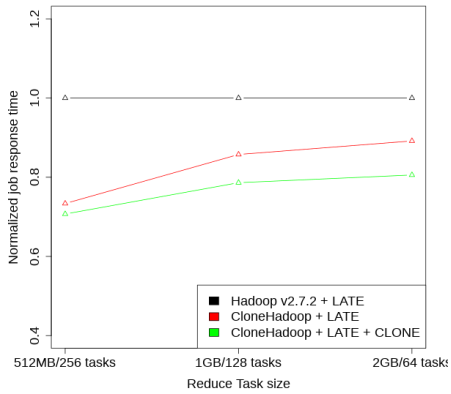


Fig. 10: Reduce task scaling: Runtimes for same Terasort params (128GB total size) over reduce task sizes of 0.5-2GB

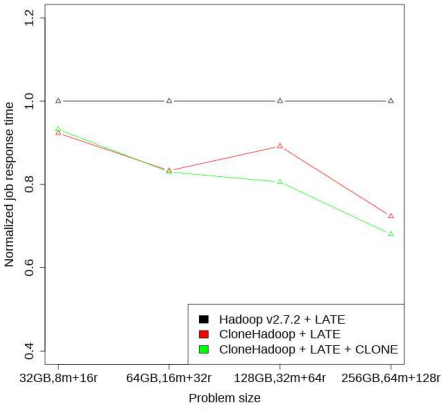


Fig. 11: Weak scaling Terasort: sizes 32-256GB, map/reduce tasks 4/2GB.

Tab. III: Application Params

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

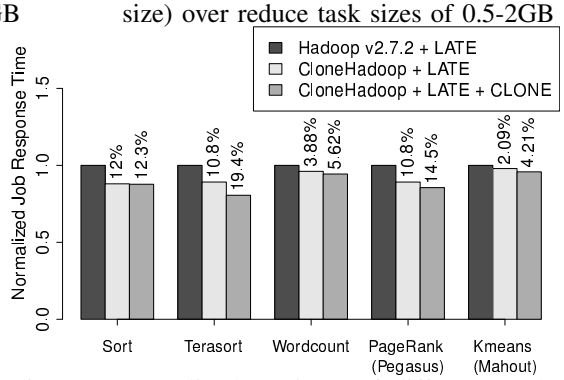


Fig. 12: Normalized runtimes of different workloads for $\beta = 1.2, shape = 2$

more competitive and finish before stragglers, even with the cloning overhead.

Observation 8: Larger reduce tasks improve performance under cloned speculations.

Fig. 10 shows the variation of normalized job response time over reduce task sizes. With an increase in reduce task size, the performance of CloneHadoop tends towards the reference Hadoop version: Larger task sizes counteract any gains by 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 compared to CloneHadoop with the LATE speculator.

Observation 9: Larger task sizes scale well under cloning.

In general, we observe performance improvements for larger task sizes with cloned speculations, even for reduce tasks whose file clone set increases for larger task sizes. While new reduce speculations are launched faster than cloned ones, 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.

Problem scaling: We analyzed the scalability of the speculation algorithm with different problem sizes. We measured normalized job response times for Terasort, ranging from 32 GB to 256 GB, assigning map tasks and reduce tasks of 4GB and 2GB data, respectively. Fig. 11 shows the comparison of normalized job response times with job sizes.

Observation 10: Performance improvements of cloning increase with larger job sizes.

We observe that the cloning-enabled speculator performs better for larger problem sizes. For smaller sized problems, fewer tasks limit the capability of the cloning speculator to improve performance. CloneHadoop with cloned speculations improves performance by up to 25%.

Application sensitivity: We analyze the performance of CloneHadoop on several large problems from HiBench for applications built on top of Hadoop. Table III describes the problem sizes for each application. Fig. 12 shows the performance of our approach 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 CloneHadoop. 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. Fig. 12 shows similar performance benefits for multi-job applications over longer time periods. The normalized job response plot shows cumulative effects of improvements on multiple jobs and average performance improvements of the speculative algorithm are similar to single job improvements.

Discussion on Cloning Overhead: The decision on whether to clone or spawn a new speculative task is captured by the CloneScore metric in terms of timing overhead. The resource overhead of cloning includes: 1. The network overhead required for cloning is determined by the bandwidth limitation of the interconnect since the process image is written to a RAM disk. 2. The I/O overhead includes (a) the incomplete spill file (up to spill size) for cloned map task attempts and (b) the size of input files fetched for cloned reduce task attempts, which is upper bounded by the size of input files for a new reduce task attempt. 3. The memory overhead includes the size of the RAM disk during cloning, albeit only for the duration of cloning since it is then freed again. Of course, finishing a job earlier (cloning) makes compute and I/O resources available earlier, too. Our performance benefits of up to 25% do not come at any additional cost of others when comparing to Hadoop's straggler model, which adds straggler tasks to the workload. We add clone tasks to the workload, but only as many as straggler tasks, and they perform *less* work (since there is no recomputing, unlike stragglers).

VII. 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 exist that can improve job performance. Cloned speculations complement existing speculation strategies and do not adversely affect the job performance. In fact, cloned speculations improve efficiency and increase the number of successful speculative attempts. Overall, speculative tasks have a higher probability of catching up to stragglers (and subsequently improving completion times) if they resume from the current point of execution via cloning instead of repeating the straggler work. CloneHadoop thus has the potential to reduce the long tail of Hadoop's MapReduce paradigm.

Acknowledgment

This work was funded in part by NSF grants 1217748, 1525609, and NIH grant 1R01AR072013-01.

References

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

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 2–2.

[4] A. S. Foundation, "Hadoop," https://hadoop.apache.org.

[5] S. Owen, R. Anil, T. Dunning, and E. Friedman, "Mahout in action. greenwich, ct," 2011.

[6] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *International Conference on Data Mining*, 2009, pp. 229–238.

[8] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI*, vol. 10, no. 1, 2010, p. 24.

[9] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Why let resources idle? aggressive cloning of jobs with dolly," *Memory*, vol. 40, no. 60, p. 80, 2012.

[10] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *European conference on Computer systems*, 2010, pp. 265–278.

[11] N. J. Yadwadkar and W. Choi, "Proactive straggler avoidance using machine learning," *White paper, University of Berkeley*, 2012.

[12] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in *ACM Symposium on Cloud Computing*, 2014, pp. 1–14.

[13] Z. Ren, J. Wan, W. Shi, X. Xu, and M. Zhou, "Workload analysis, implications, and optimization on a production hadoop cluster: A case study on taobao," *IEEE Transactions on Services Computing*, vol. 7, no. 2, pp. 307–321, 2014.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Symposium on Cloud Computing*, 2013, p. 5.

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies*, 2010, pp. 1–10.

[16] "Criu: Checkpoint restore in userspace," https://criu.org/.

[17] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006, p. 494.

[18] O. Laadan and S. E. Hallyn, "Linux-cr: Transparent application checkpoint-restart in linux," in *Linux Symposium*, vol. 159, 2010.

[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *OSDI*, vol. 8, no. 4, 2008, p. 7.

[20] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones." in *NSDI*, vol. 13, 2013, pp. 185–198.

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

[22] ——, "Skewtune: mitigating skew in mapreduce applications," in *SIGMOD Int'l Conference on Management of Data*, 2012, pp. 25–36.

[23] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: Trimming stragglers in approximation analytics," in *NSDI*, 2014.

[24] K. Kawachiya, K. Ogata, D. Silva, T. Onodera, H. Komatsu, and T. Nakatani, "Cloneable jvm: A new approach to start isolated java applications faster," in *Virtual Execution Environments*, 2007, pp. 1–11.

[25] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop acceleration through network levitated merge," in *Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 57:1–57:10.

[26] J. Li, C. Pu, Y. Chen, V. Talwar, and D. Milojicic, "Improving preemptive scheduling with application-transparent checkpointing in shared clusters," in *Middleware Conference*, 2015, pp. 222–234.

[27] H. Xu and W. C. Lau, "Optimization for speculative execution of multiple jobs in a mapreduce-like cluster," *CoRR*, vol. abs/1406.0609, 2014, withdrawn. [Online]. Available: http://arxiv.org/abs/1406.0609

[28] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai, "Hibench: A representative and comprehensive hadoop benchmark suite," in *ICDE Workshops*, 2010.

[29] O. OMalley, "Terabyte sort on apache hadoop," pp. 1–3, 2008, http://sortbenchmark.org/Yahoo-Hadoop.pdf.