2016 IEEE 9th International Conference on Cloud Computing

# Hadoop in Flight: Migrating Live MapReduce Jobs for Power-Shifting Data Centers

Chili Johnson David Chiu
Department of Mathematics and Computer Science
University of Puget Sound
{creavesjohnson,dchiu}@pugetsound.edu

***Abstract*—Renewable energy sources such as wind and solar are unpredictable for power utilities, which must produce exactly as much power as is needed at any given time. To help manage the demand, some utilities have begun deploying real-time energy prices to their customers. Data centers, which often run Hadoop jobs on thousands of machines, have become some of the utilities' largest consumers. In fact, recent studies have shown that, when processing at full capacity, data centers can require as much power as a mid-sized U.S. city. By implementing a method in which data centers can offload their work to locations on different power grids, they can take advantage of the lower-priced energy and thereby minimize operational costs. To this end, we have designed and implemented a new mechanism directly within the Hadoop 2 codebase that allows users to pause, migrate, and resume a job at arbitrary points of execution. We have evaluated this scheme using popular applications and show that energy can be delayed and shifted to a different location with reasonable overheads. Our experiments justify the migration use-case, showing that it saves both energy and time over either restarting the job remotely or allowing it to complete locally.**

***Keywords*—hadoop, data center, power, migration**

## I. Introduction

Due to its ease of programming and ability to process large volumes of data, Hadoop, the open-source implementation of Google's MapReduce programming model [1], has become ubiquitous within data center operations. For instance, Amazon has been offering its popular *Elastic MapReduce* for a number of years [2]. Facebook uses Hadoop and its components in every product they offer [3]. Each day, hundreds of Hadoop jobs are run on Facebook's clusters performing tasks such as generating reports, fighting spam [4], and storing the 500+ TB of data produced daily by its users [5]. Yahoo!'s Webmap uses Hadoop to graph all known web pages [6].

As data centers grow in computational capacity, so does their hunger for power, adding to the costs of operation. Larger organizations operate data centers at multiple geographical locations, and each location may require 10s to 100s of megawatts (MW) of power, enough to sustain a midsized U.S. city. Furthermore, a recent study reported that data centers collectively consume up to 2.2% of the world's energy and is projected to increase significantly over the next decade [7].

Simultaneously, power engineering advancements have increased the integration of renewable energy on the electrical grid. This allows grid operatives (*e.g.*, California's CAISO) to offer real-time dynamic pricing (RTP) based on current demand and energy availability [8]. RTP therefore serves as an incentive for data centers to regulate their power usage. Consider an organization that operates a set of $n$ geographically isolated data centers $\{d_1, ..., d_n\}$. We denote $P_t(d_i)$ as the RTP at time $t$ at location $d_i$. Intuitively, when $P_t(d_i)$ is considerably lower than $P_t(d_j)$, it would make financial sense to maximize energy usage at $d_i$ while powering down $d_j$. This model is depicted in Figure 1 and is generally accepted in the current state of research [9], [10], [11], [12], [13].

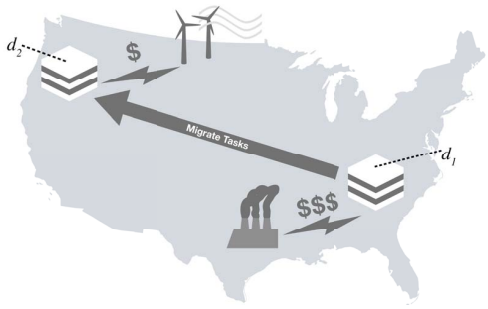


Fig. 1. Workload Migration for Cost Reduction.

Because Hadoop jobs represent a significant workload at data centers, and due to their known power inefficiencies [14], a method that can efficiently migrate jobs between isolated and geographically distributed sites will allow data centers to exploit differences in renewable-energy availability. One policy is to use scheduling decisions: $d_i$ can could simply kill and/or delay the execution of a potentially large set of Hadoop jobs and restart them at a later time $t + u : P_t(d_i) > P_{t+u}(d_i)$. Another policy is to monitor the RTP prices at all $n$ locations. Then $d_i$ can once again kill the execution of its set of jobs, while starting them at a less costly site $d_j$ instantaneously at time $t : P_t(d_i) > P_t(d_j)$. This approach requires that the input data are also available at $d_j$, which is not an unreasonable assumption. For example, replicating data across multiple sites increases performance and data availability, and is therefore not uncommon practice [15], [16], [17], [18].

Both approaches, however, suffer from a loss of progress. In the former approach, the delayed jobs make no progress toward completion, and worse, the killed jobs lose any progress they have already made. This could possibly violate service-level agreements (SLAs) on time-critical applications. The latter approach of restarting the job at a disparate location ameliorates the progress issue to an extent. However, any unfinished Hadoop job at $d_i$ would still lose their intermediate state (which may have taken significant time and energy to compute), and must again be restarted from the beginning at location $d_j$. This is because Hadoop does not feature *pause*

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.20

and *resume* operations. The focus in this paper is on providing these operations natively within the Hadoop framework.

In this paper we describe the design and implementation of the *pause* + *migrate* + *resume* operations for Hadoop. The *pause* operation saves the current state of the job. It captures all intermediate state that has been generated, *i.e.*, spill files and key-value pairs in memory. The *migrate* operation allows the user to specify a destination for job resumption. Finally, *resume* will expand the state-file and resume execution, without losing any partial results. This operation can be invoked at arbitrary points in the map, shuffle, or reduce phases. Our experimental results show that, while job migration incurs an overhead, the energy savings is significant and offsets this cost.

The remainder of this paper is organized as follows. In Section II, we present some background on Hadoop's structures, objects, and runtime behaviors. We describe our methods for checkpointing during each phase of a MapReduce job in Section III. Experimental results are presented in Section IV. Related works is discussed in Section V, and we conclude and describe plans for future works in Section VI.

## II. Background

In this section, we present the background and terminologies in Hadoop 2.7, which is the current version at the time of writing [19]. The behavior of a Hadoop job is represented internally with a set of three state machines: Job, Task, and TaskAttempt, and all of which are controlled by the application master.

### A. Hadoop State Machines

The Job state machine tracks the state of the singular job and includes states and transitions for its setup, starting tasks, killing, failing, and termination. The Task state machine represents one task that is associated with a job—these are typically Map Tasks or Reduce Tasks. It should be emphasized that Map Tasks and Reduce Tasks are not to be confused with mappers or reducers. Mappers and reducers are the user-defined classes which "map" and "reduce" key-value pairs. In contrast, Map Tasks and Reduce Tasks are responsible for calling the respective user-defined methods on these objects in addition to many other setup, cleanup, and progress-tracking.

Tasks are also responsible for creating and monitoring TaskAttempts. A TaskAttempt's state machine represents a single attempt at performing a Map or Reduce Task and is responsible for requesting containers from YARN. A Job that runs without Task or node failures will typically have one TaskAttempt associated with each Task. However, if speculative execution is enabled, or if an attempt fails and is restarted, then a Task may have multiple associated TaskAttempts.

The lifecycle of a Hadoop job begins with the Job object. A Job begins in its `INIT` and `SETUP` states by constructing the configuration from user parameters, cluster parameters, *etc.*, which define the parameters for the job being executed. The Job also performs some setup, including computing the sizes of *input splits* (logical blocks of data). After setup, the Job creates Tasks and distributes Task-specific configurations to them, *e.g.*, input-split paths. Tasks begin in a `SCHEDULED` state in which they create a TaskAttempt.

TaskAttempts wait in their `UNASSIGNED` state until sufficient resources are assigned from YARN, after which it launches its remote JVM in the allocated YARN container. After the launch of the remote JVM, both the TaskAttempt and its associated Task transition into their `RUNNING` states. After all TaskAttempts succeed and have disassociated from their remote containers, their associated Task transitions into its terminal `SUCCEEDED` state. Finally, once all Tasks have completed, the Job handles various cleanup procedures and eventually transitions into its terminal `SUCCEEDED` state.

### B. Rationale for Native Support

To efficiently utilize hardware, data centers often use virtualization [20], allowing virtual machines (VM) to coexist on the same physical hardware. Recent VM research has made it possible to seamlessly migrate VMs from one host to another [21], [22], which is an ostensibly natural method to migrate Hadoop jobs. We approached Hadoop job migration from the level of Hadoop's implementation because we believe that it offers a level of use-case granularity which may not be possible with VM migration.

Suppose a Hadoop cluster is set up over a collection of VMs, *i.e.*, HDFS datanodes and YARN node managers are running within VMs, then there is no guarantee that one user's job's tasks and data will exist on an exclusive subset of a cluster's virtual machines. Multiple users' file blocks in HDFS are likely to be co-located on any one virtual machine, not only to satisfy block replication, but also to help with data locality for running tasks. Additionally, for any running job, YARN is likely to assign containers on a single VM to multiple users' jobs simultaneously, again as an effect of Hadoop's goal to improve performance through data-locality.

For these reasons, if one were to determine which VMs were responsible *both* for storing HDFS block relevant to a single job, *and* which VMs were running that job's containers, then migrating those VMs would likely migrate irrelevant data blocks and containers in use by others. Not only would this extra data impact the performance of a migration, but it would also break the execution of the other user's job. Therefore, to migrate a single job (among many) the entire cluster must be migrated to preserve correctness.

For this reason, VM migration does not provide the job-level migration granularity that would be necessary for such a solution to be adopted in a production setting. Therefore we approached job migration by extending Hadoop's source code. By working at this lower level, we have the granular access to job-by-job data which allows us to migrate jobs without affecting the execution of other jobs on a cluster. The infeasibility of using VM migration to Hadoop jobs was also observed by Leverich and Kozyrakis [14].

## III. Checkpointing Implementation

Our goal in adding the new *pause* + *migrate* + *resume* operations is two-fold: (1) to introduce minimal disruption to the core Hadoop source and state machine, (2) to introduce no runtime overhead (until our operations are invoked), and (3) from the usage perspective, the new features are transparent

and appear additive, thereby preserving all original functionalities of Hadoop. In this section, we provide a nuanced description on the modifications we have made.

We store a Hadoop Job's *state* in two parts: First, a directory is created in the Job-owner's working directory within HDFS to store checkpoint data. The files within this directory will be various representations of intermediate data from the TaskAttempts running at the time when a *pause* request is received. Second, a metadata file is generated, containing paths to intermediate data files, byte-offsets for input splits, reducer inputs, *etc*. This metadata is used to reconstruct and resume the paused job.

We consider the process by which a Hadoop job's state is saved on a Task-by-Task basis. When Hadoop receives a request to pause a Job, that Job's application master is notified. It is the application master's responsibility to notify the Tasks of the *pause* request, as well as to collect and save the metadata related to each Task's checkpoint data. Each Task is therefore responsible for saving its own checkpoint data to HDFS and reporting its metadata to the application master.

### A. State Machine Modifications

To accommodate job-pausing, we created "decorator" objects that modify the state machines which model the execution of MapReduce jobs. Two new states (`PAUSING` and `PAUSED`) were added to each of the three existing state machines: Job, Task, and TaskAttempt. The `PAUSING` state can be reached by each machine from the `RUNNING` state, and is maintained while each Job, Task, or TaskAttempt is saving checkpoint data, unless interrupted by an error or kill signal in which case the machine responsible will transition to `KILL_ABORT`. If a machine receives a pause signal while in the `SCHEDULED` or `INIT` states, it simply transitions to the `KILL_ABORT` state and is terminated.

Each state machine will transition to its terminal `PAUSED` state only after its associated task has ensured that its checkpoint data has been properly saved, and metadata has been collected. The Job state machine is responsible for notifying each of its associated tasks of the pause request. Shown in Figure 2, only after a Job has been notified that all containers have been terminated will it transition from `PAUSING` to `PAUSED`.

Figure 3 shows the modified Task state machine. Each Task is responsible for notifying only one of its TaskAttempt of the pause request. Typically the notified TaskAttempt is the only one associated with that given Task. However, if a TaskAttempt has previously failed and been restarted, the most recent running TaskAttempt will be notified of the pause request. In the case of a Task with speculative execution, the running TaskAttempt with the most progress will be notified of the pause request and all other running TaskAttempts will be killed. Only once a Task's TaskAttempts have all reached a terminal state will the Task transition to `PAUSED`.

Each TaskAttempt and its associated process running in a remote container are responsible for saving checkpoint data onto HDFS and delivering checkpoint metadata to the application master. When both tasks are complete, the remote container terminates, and the TaskAttempt state machine transitions into the `PAUSED` state. This is shown in Figure 4.

### B. Checkpointing

A user may invoke a pause request at arbitrary points of Job execution. We have considered four discrete Hadoop Job phases, each of which has a different protocol for saving checkpoints. Map Tasks only have one protocol for saving checkpoint data, while Reduce Tasks have three protocols, one protocol for the sort, shuffle, and reduce phases of the task.

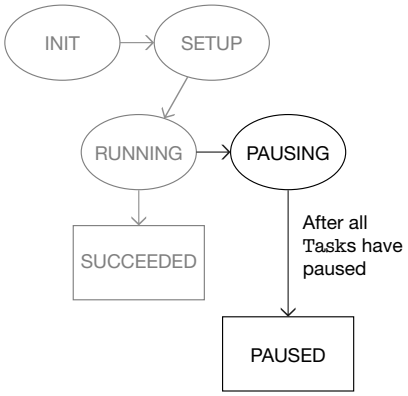


Fig. 2. `Job` state machine highlighting additional states.

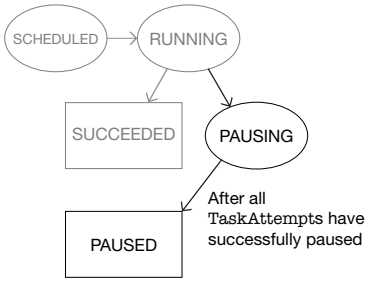


Fig. 3. `Task` state machine highlighting additional states.

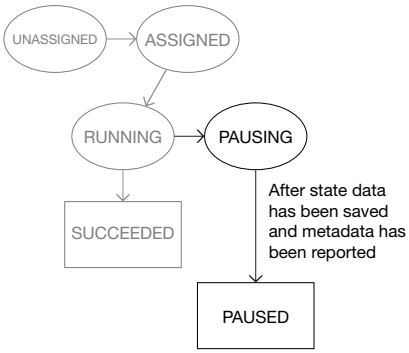


Fig. 4. `TaskAttempt` state machine highlighting additional states.

*1) Map Phase:* When a Map Task receives a pause request, it saves two sets of files into HDFS. The first set contain intermediate "spill files," which are the intermediate lists of key-value pairs as generated by the mapper that are spilled out of the mapper's in-memory buffer and onto the disk. If any key-value pair has not been spilled from the in-memory buffer when the pause request is received, a final spill is completed

before all spill files are transferred into HDFS. The second set of files stored as part of the map phase checkpoint are spill-index files, *i.e.*, those which map keys to sectors within the spill files for quick lookup of key-value pairs during the sort and shuffle phases. By working at the file level, we ensure that the map phase's pause mechanism is compression-agnostic.

Each Map Task reports metadata to the application master. This metadata comprises (1) an input split byte offset and length, (2) HDFS paths to the saved spill files, (3) HDFS paths to the saved spill-index files, and (4) a Task ID. The byte-offset and length define the region of the input file which the map task has yet to read. The resumed map task will seek to the byte offset in its input file and continue mapping until it has read up to its defined length. The spill paths and index paths are relative to the checkpoint storage directory and allow Hadoop to re-acquire the mapper's intermediate data for restart. When resuming, the Task ID is used to ensure that spill files can be correctly associated with the mapper reading the original input split from which they were generated.

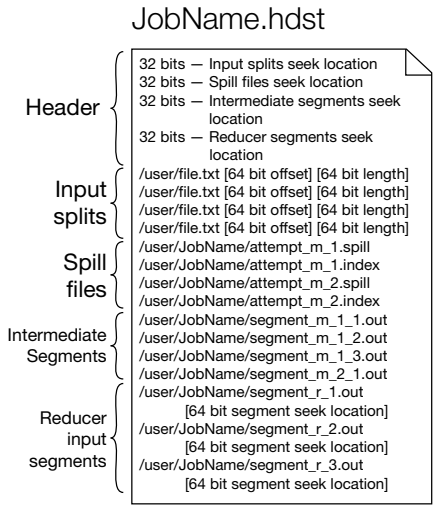


Fig. 5. An example Hadoop job state metadata file.

The metadata file is split into segments for each type of checkpoint data associated with each phase of the Job's lifecycle. The *header* contains the locations within the metadata file to which to seek in order to begin reading any given segment. Some segments (*e.g.*, input splits section for instance) contain not only file paths, but 64-bit integers associated with those files which are used during resume to recreate various states.

By storing input-byte offsets on the abstract file level, the pause and resume mechanisms can be used to migrate Jobs between clusters with HDFS configured to use *different* block sizes. A resumed mapper may be required to operate over different HDFS block sizes, depending on the relative sizes of each cluster's configured block size. In the case where a mapper would be required to operate over multiple HDFS blocks (where the block size on the first cluster is significantly larger than the block size on the second cluster), input splits are still calculated along block boundaries, which results in what was one map task when paused becoming multiple map tasks when resumed. If these conditions are reached, only one map task is responsible for the spill files saved from the original map task.

*2) Sort Phase:* The design of the sort phase mechanism is a simplified version of the map phase counterpart. When a map task enters the sort phase and receives a pause signal, the pause mechanism will wait until all map-spill files have been merged into one intermediate file and one index file before transferring both files to HDFS. Because a mapper will have completed by the time a map task reaches its sort phase, instead of reporting a byte offset and length for the input file, the task reports that the entire split's start offset and length will be excluded from the Job when it is resumed.

*3) Shuffle/Merge Phase:* We checkpoint during the shuffle and merge phases of a Reduce Task using a single protocol. When a Reduce Task receives a pause request during its shuffle and merge phase, the task begins blocking all shuffle fetcher threads, preventing any new intermediate map outputs from being consumed by the reduce task. These intermediate outputs are stored either in memory or on disk before being merged into the final input of the reduce task.

When pausing, the routine is blocked by any in-memory or on-disk merger threads. Once all fetcher threads have finished their current fetch task, and in-memory merger and on-disk merger threads have completed their current merges, the pause routine continues. All in-memory map task outputs are converted to on-disk map outputs, while all in-memory intermediate merge segments are converted to on-disk merge segments. These segments are then transferred into HDFS. Before finally terminating, the task reports the metadata to the application master. First, all file paths to the saved segments are reported. Second, a task partition identifier is reported in order to ensure that, during a resume, fetchers continue to fetch data from the correct partition in the intermediate map outputs. Third, map task identifiers are reported so that after a resume, a fetcher will not fetch duplicate data by requesting outputs from a map task it has already queried, and lastly this task's identifier is reported to ensure that all segments make their way back to the same task following a resume.

During a resume, each data structure used to hold map output segments or merged segments are recreated from the saved on-disk segments in HDFS. This leaves the in-memory segment buffers free to continue accepting more segments, if any are required.

*4) Reduce Phase:* The reduce phase checkpoint protocol is modeled very similarly to the map phase's. However, unlike a map task, which takes a single file split as an input, a reduce task can read from both an in-memory buffer of data segments and an on-disk collection of inputs to feed the reducer. The abstract "reducer input" is represented internally as a priority queue of both types of segments. Priority is highest for segments whose next unread key-value pair has a key which is identical to the key of the last-consumed pair. Priority is lower for those whose next pair has a different key.

While saving a checkpoint, each of these in-memory segments or on-disk input files are individually treated very similarly to the input file splits in map tasks. When a reduce task receives a pause signal, the reducer loop blocks the task from pausing until it has successfully completed reducing and outputting the key-value pair on which it was reducing

when the signal was received. Once this condition is met, the task continues its pause routine. All in-memory segments are converted to on-disk segments and subsequently saved to HDFS. The locations of these segments along with this task's identifier is reported to the Application Master. Additionally, the byte locations up to which the reduce task read into each segment is saved as metadata.

To resume a Reduce Task, each on-disk segment is loaded into the local filesystem and a seek is performed to the byte offset location stored in the metadata. These segments are then fed back into the reducer's input priority queue and return to their original order based on the queue's definition of priority. Once this is complete the Reduce Task starts the reducer and the reducer continues from where it left off.

## IV. Experimental Evaluation

In this section we first describe our experimental setup, the test applications, and the data set used for evaluation. We remind the reader that a data center's energy cost is a function of power consumption, and in practice, a change in current pricing might invoke migration. The purpose of this evaluation, however, focuses on the performance and energy impacts of migration. Affects on costs will be address in future work.

### A. Experimental Setup

Our setup consists of two small homogeneous clusters containing three nodes each. Each node is equipped with 16 GB of RAM, a 3.4 GHz 8-core Intel Xeon processor, 1 TB of local disk storage, and are on the same network. All machines serve as HDFS datanodes (with a replication factor of 3) and have a YARN node manager. One node in each cluster serves as the namenode. The *combiner* is enabled to minimize intermediate data and network load. To collect energy consumption data, each cluster was installed a *Watts up? PRO ES* meter. One node in each cluster was responsible for polling its respective load meter every second and recording those values to disk for the duration of a test. To collect network activity data we used *Ganglia* distributed system monitor [23].

The applications we used were: (1) `inverted index` over the Apache Software Foundation Public Mail Archives (52.5 GB) [24], retrieved from a public snapshot on AWS. Shorter jobs, (2) `grep` and (3) `wordcount` were also run over an 11 GB text file containing concatenated copies of the classic Charles Dickens novel, *"A Tale of Two Cities."* We stored a copy of these data sets into HDFS in both clusters.

### B. Results

Figure 6 charts power consumption and network activity data for our inverted-index program running over our test data set without any migrations. This job took 1323 seconds to complete, using 355 kJ of energy. This run serves as the baseline for comparison.

Next, we evaluate our mechanism by invoking pause and migration at various points of execution. These points were selected arbitrarily but within each of these phases: map, shuffle/sort, and reduce.

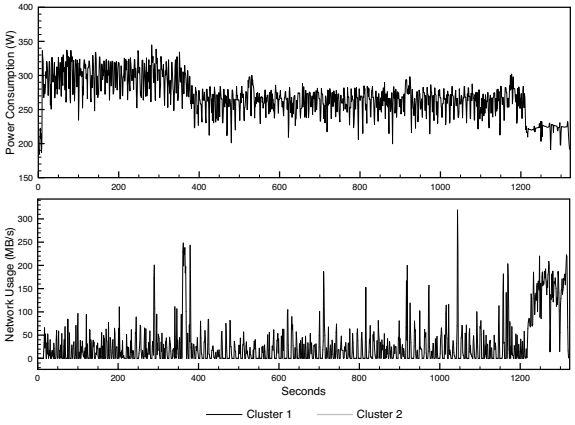


Fig. 6. Power consumption and network activity over the duration of `inverted` without migration.

*1) Map Phase Migration:* Figure 7 shows the power consumption and network activity for inverted index running in cluster 1 (black line), being paused before completion, its checkpoint data being migrated to cluster 2 (grey line), and finally being restarted on our second cluster. The power consumption is shown in the top plot, while the bottom chart plots the network activity. The initial job was paused at 27% map completion and 0% reduce completion.

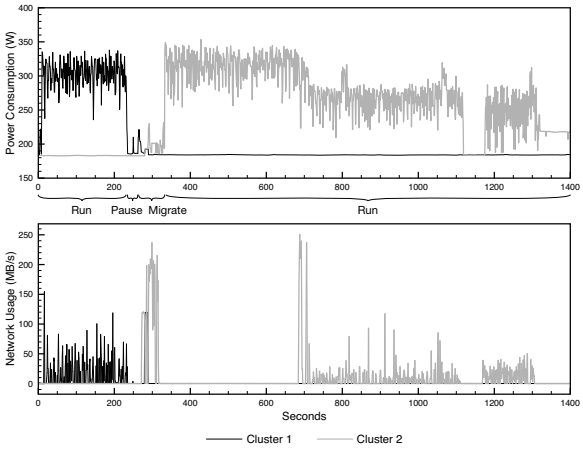


Fig. 7. Power consumption and network activity over the duration of `inverted` migrated between two clusters. The job was paused at 27% map progress, 0% reduce progress.

As can be seen, the pause interval took 27 seconds, constituting a 2% overhead over the baseline execution time. Power consumption and network activity drop significantly during this period, because all mappers have been stopped and any unsaved state data is being saved to HDFS. During this interval, the application master polls running containers for state metadata and saves this data to HDFS, which can be seen as short spikes during the pause interval in Figure 7 (top).

Next, the migration step occurs, taking 70 seconds (5.3% overhead) and is characterized by a large spike in network activity in both clusters. The small spike in cluster 1's network activity is due to the `hdfs dfs -get` command used to

retrieve the checkpoint data from HDFS. During the actual copying during migration, cluster 1's network activity is equal to that of cluster 2's network activity because the checkpoint files (908.7 MB) are being transferred between clusters. After the copying is complete, and the checkpoint files begin transferring back into HDFS, cluster 2's network activity nearly doubles as a result of HDFS replication.

Finally, for the remainder of the map phase, the network activity drops due to the mappers working on local input splits. Toward the end of the map phase (at 700 seconds), power consumption decreases and a large spike in network activity can be seen. This is explained by map tasks, which are resuming on cluster 2. The spike in network activity is the resuming map tasks taking their designated checkpoint data from HDFS and rendering that data available to reduce task data fetchers. Because no checkpoint data was saved for any reduce tasks, the job resumes normally after 725 seconds.

The migration added 95 seconds to the overall job completion time, a 7.2% overhead of the baseline job completion time. If cluster 1 is put into low-power mode after migration, it uses 87 kJ of energy (24.5% of the baseline).

*2) Mixed Phase Migration:* Next, we evaluate migration time when a job is in both the map and reduce phases. Figure 8 charts power consumption and network activity data for a job paused at 50% map and 17% reduce progress. This test is characterized by many of the same migration-related features as the previous test. In comparison with the experiment in Figure 7, the pause interval exhibits much more network activity during the pause interval due to a much larger amount of checkpoint data which must be saved to HDFS, 3.6 GB, but is very similar otherwise. In this case the migration mechanism added about 232 seconds to the total job time, an 18% overhead of the baseline. If cluster 1 is put into low-power mode after migration, it uses 148 kJ of energy (41.7% of the run).

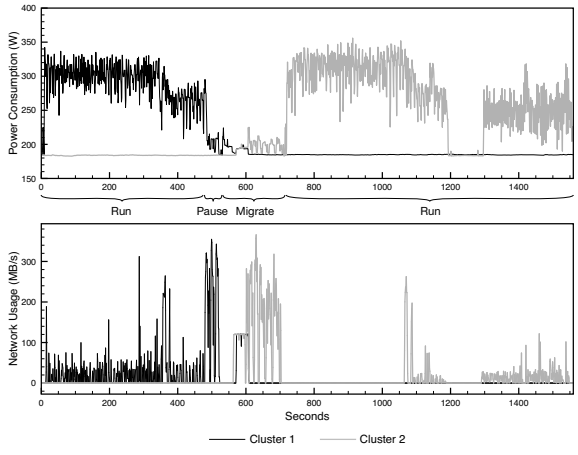


Fig. 8. Power consumption and network activity over the duration of `inverted` migrated between two clusters. The job was paused at 50% map progress, 17% reduce progress.

*3) Reduce Phase Migration:* Finally, we evaluate migration overheads when a job is in only the reduce phase. Figure 9 shows power consumption and network activity data for a job paused at 100% map and 83% reduce progress, the point at which the reducer has reached 50% completion. The checkpoint data totaled 5.5 GB, the largest so far. This positive correlation between checkpoint data size and execution time before pause is expected, because as more input splits are processed, more intermediate data is produced. However, the extent of this correlation can vary wildly with the type of job being run (as we will show later). In total, the time cost of this migration was 329 seconds, or a 25% increase in baseline execution time. If cluster 1 is put into low-power mode after migration, it uses 354 kJ of energy (99% of baseline), marking the first time that a migration would be less efficient than allowing the job to finish on cluster 1.

In this experiment, network speed during a migration and resume created a significant bottleneck. High network utilization and low power consumption suggest that the rate at which a migration can be performed is limited by the rate at which data can be retrieved from and transferred into HDFS. The drop in network activity during the latter run phase can be explained by data locality. Due to the replication factor, 3, of HDFS on the test clusters of size 3, the necessary data required to recreate the data structures which drive the reduce phase are all local to the reducer, so no block transfers over the network were necessary.

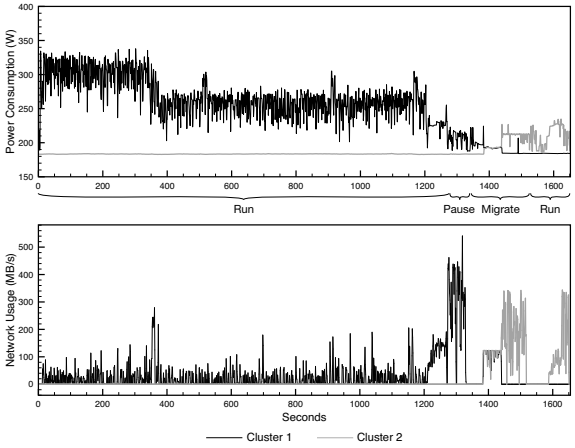


Fig. 9. Power consumption and network activity over the duration of `inverted` migrated between two clusters. The job was paused at 100% map progress, 83% reduce progress (50% reducer completion).

*4) Short Jobs:* The overhead of job migration is much more apparent for shorter jobs. Figure 10 shows the `grep` application running over 11 GB of text, and Figure 11 shows the `wordcount` application from the same `jar` running over the same text with combiners. The `grep` task is very much CPU-bound in comparison to the inverted index program, so very few network resources are required during the execution or migration of such a job. There is, however, still the overhead of waiting for containers to close cleanly, and that overhead is very apparent during the pause phase in Figure 10. Overall, this run exhibited 24% overhead during the mixed phase.

The `wordcount` program, being relatively I/O-bound, requires more network activity throughout its execution. Somewhat surprisingly, the pause and migration time required for `wordcount` was 20 seconds faster than the comparable `grep` job, resulting in only 6% overhead. This is likely due to

the pause happening at a serendipitous point in the YARN container life cycle; the application master may not have needed to wait very long for running containers to close.

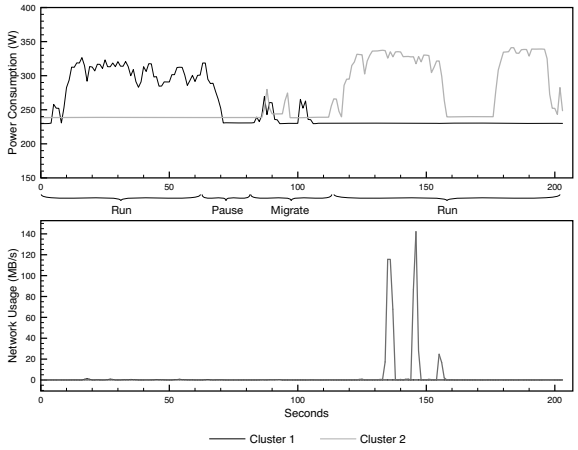


Fig. 10. Power consumption and network activity over the duration of `grep` migrated between two clusters. The job was paused at 54% map progress, 17% reduce progress.

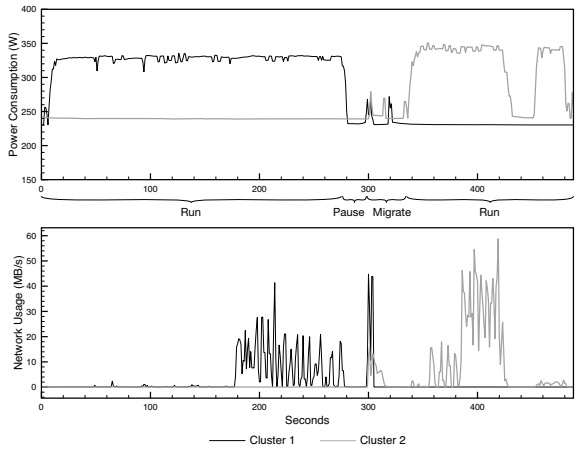


Fig. 11. Power consumption and network activity over the duration of `wordcount` migrated between two clusters. The job was paused at 72% map progress, 23% reduce progress.

## V. Related Works

Reducing data centers' energy usage has been at the forefront of recent systems research, motivating the need for "greening" data centers [25], [26], [27], [28], [29], [30], [31], [32]. These efforts consider lowering energy usage unilaterally at a data center location by integrating on-site renewable energy resources (*e.g.*, co-locating solar panels and energy storage systems). More in line with our work is the *follow-the-renewables* policy, where workloads are routed among various green data centers to take advantage of their local renewables. If the participating data centers are located in different energy markets, then opportunities for energy-cost reduction can be exploited. We focus on two major veins of approaches toward mechanizing the *follow-the-renewables* policy.

The first approach involves dynamically routing web traffic. Web traffic is a natural workload to target due to its volume and intrinsic routing support. For instance, an HTTP/S request can be rerouted to a different locale given changes committed to the DNS table. Rao, *et al.* minimize overall costs by solving for optimal resource allocation and web-request rates at multiple data centers [11]. Chen, *et al.* presented a centralized scheduler that migrates workloads across data centers in a manner that minimizes brown energy consumption while ensuring the jobs' timeliness [33]. Zhang, *et al.* additionally consider meeting a budget cap [34]. Other works further consider on-site intermittent renewable energy availability based on location [10], [35]. Aikema *et al.* study the feasibility of using data centers as ancillary services [36]. This class of works is orthogonal, but complementary, to our own.

A second class under the *follow-the-renewables* policy involves the migration of batch jobs. Para-virtualization (*e.g.*, Xen [20] and KVM [37]) has enabled $M$ VMs to be condensed onto a set of $N$ physical servers for some $M > N$. VM consolidation not only increases CPU utilization, but the "empty" physical servers can be powered off or placed on low-power mode, saving energy costs [38]. VM migration techniques followed that can copy a *running* VM instance to a new physical destination with minimal interruption [21], [22].

Le, *et al.* propose policies for VM migration across multiple data centers in reaction to power pricing in a cloud for high performance applications [39]. Liu, *et al.* proposed the GreenCloud architecture, which combines online monitoring of physical resources with a technique for finding power-saving VM placements [28]. Beloglazov, *et al.* present heuristics on VM placement to optimize power savings [29]. Buchbinder, *et al.* propose online solutions for migrating batch jobs to optimizing costs [40]. In Akoush, *et al.'s Free Lunch* [41], the authors argue for either pausing VM executions or migrating VMs between sites based on local and remote energy availability. In contrast to the above works, our migration mechanism is implemented directly in Hadoop. Only the intermediate state must be transferred, avoiding the overhead of transferring entire VMs. Moreover, we have earlier described the impracticalities of using VM migration as a tool to transfer Hadoop jobs across sites.

## VI. Conclusion and Future Works

In this paper we propose a checkpoint and migration scheme implemented directly in Hadoop. The checkpoint includes intermediate data generated, as well as a metadata file to reconstruct the state for resumption. We evaluated our approach during all phases of a Hadoop Job. Our results provide a justification for Hadoop migration. We showed that both time and energy can be saved by migrating an already-running job, rather than (1) allowing the job to finish on the local cluster, or (2) killing the job and restarting it on the remote cluster. We also showed that there are cases when migration would not be beneficial, *i.e.*, when the state is expected to be large and the job is nearly finished. Therefore, a tradeoff exists and will be exploited in future works.

One future work would be to design a "pipelined" migration in which intermediate data at the time a job is paused is migrated, while allowing the job to continue generating new

intermediate data until the first data was successfully migrated. This would allow some of the time cost of performing a pause and migration to be hidden during execution of different phases of a job, minimizing the net cost in job execution time.

In the migration step, performance hits a bottleneck both when `get`ing state data from HDFS then storing it locally, and when `put`ing state data into HDFS. This may also be an issue if the local volume used to migrate state data is not of sufficient size to fully contain that data (in which case the data must be retrieved and migrated in iterations). Developing a method in which files can be transferred between HDFS volumes without the need to use an intermediate local volume would improve the performance of state-data migration.

## ACKNOWLEDGMENT

This work was supported in part by a McCormick Research Grant from the University of Puget Sound.

## REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] "Amazon emr, https://aws.amazon.com/elasticmapreduce/."

[3] D. Henschen. (2013) Facebook on big data analytics: An insider's view. http://www.informationweek.com/big-data/big-data-analytics/facebook-on-big-data-analytics-an-insiders-view/d/d-id/1109106.

[4] J. S. Sarma. (2008) Hadoop. https://www.facebook.com/notes/facebook-engineering/hadoop/16121578919.

[5] "Under the hood: Scheduling mapreduce jobs more efficiently with corona," https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-\with-corona/10151142560538920, 2012.

[6] E. Baldeschwieler. (2008) Yahoo! launches world's largest hadoop production application. https://developer.yahoo.com/blogs/hadoop/yahoo-launches-world-largest-hadoop-production-application-398.html.

[7] J. Koomey, "Growth in data center electricity use 2005 to 2010," Tech. Rep., 2011. [Online]. Available: http://www.analyticspress.com/datacenters.html

[8] FERC, "Caiso daily report archives," http://www.ferc.gov/market-oversight/mkt-electric/california/caiso-archives.asp.

[9] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs, "Cutting the electric bill for internet-scale systems," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 123–134, Aug. 2009.

[10] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. Andrew, "Greening geographical load balancing," in *SIGMETRICS'11*.

[11] L. Rao, X. Liu, L. Xie, and W. Liu, "Minimizing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment," in *Proceedings of the 29th IEEE International Conference on Computer Communications (INFOCOM'10)*, 2010.

[12] D. Chiu, C. Stewart, and B. McManus, "Electric grid balancing through low-cost workload migration," *ACM Sigmetrics Performance Evaluation Review (GreenMetrics'12)*, 2012.

[13] Y. Li, *et al.*, "Towards dynamic pricing-based collaborative optimizations for green data centers," in *Workshops Proceedings of ICDE'13*.

[14] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of hadoop clusters," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, pp. 61–65, Mar. 2010.

[15] J. Gray and P. Shenoy, "Rules of thumb in data engineering," in *Proceedings of ICDE '00*.

[16] S. Venugopal, R. Buyya, and K. Ramamohanarao, "A taxonomy of data grids for distributed data sharing, management, and processing," *ACM Comput. Surv.*, vol. 38, no. 1, Jun. 2006.

[17] T. Bicer, D. Chiu, and G. Agrawal, "Time and cost sensitive data-intensive computing on hybrid clouds," in *Proceedings of the 2012 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*, 2012.

[18] D. Boru, D. Kliazovich, F. Granelli, P. Bouvry, and A. Y. Zomaya, "Energy-efficient data replication in cloud computing datacenters," *Cluster Computing*, vol. 18, no. 1, pp. 385–402, Mar. 2015.

[19] "Apache hadoop. https://hadoop.apache.org."

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. ACM, 2003, pp. 164–177.

[21] C. C. Keir, C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005, pp. 273–286.

[22] "Vmware vmotion: Migrate virtual machines with zero downtime, https://www.vmware.com/products/vsphere/features/vmotion."

[23] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: Design, implementation and experience," *Parallel Computing*, vol. 30, 2003.

[24] "Apache software foundation public mail archives," https://aws.amazon.com/datasets/apache-software-foundation-public-mail-archives/.

[25] C. Stewart and K. Shen, "Some joules are more precious than others: Managing renewable energy in the datacenter," in *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower'09)*.

[26] A. Berl, E. Gelenbe, M. D. Girolamo, G. Giuliani, H. D. Meer, M. Q. Dang, and K. Pentikousis, "Energy-efficient cloud computing," *The Computer Journal*, vol. 53, no. 7, 2009.

[27] R. Nathuji and K. Schwan, "VirtualPower: coordinated power management in virtualized enterprise systems," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, vol. 41. ACM, Oct. 2007, pp. 265–278.

[28] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen, "Greencloud: a new architecture for green data center," in *Proceedings of ICAC'09*. ACM, 2009.

[29] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, May 2011.

[30] I. Goiri, R. Beauchea, K. Le, T. D. Nguyen, M. E. Haque, J. Guitart, J. Torres, and R. Bianchini, "Greenslot: scheduling energy consumption in green datacenters," in *SC'11*, 2011, pp. 20:1–20:11.

[31] I. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "Greenhadoop: Leveraging green energy in data-processing frameworks," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. ACM, 2012, pp. 57–70.

[32] "HP Unveils Architecture for First Net Zero Energy Data Center, http://www.hp.com/hpinfo/newsroom/press/2012/120530c.html."

[33] C. Chen, B. He, and X. Tang, "Green-aware workload scheduling in geographically distributed data centers," in *CLOUDCOM*, 2012.

[34] Y. Zhang, Y. Wang, and X. Wang, "Electricity bill capping for cloud-scale data centers that impact the power markets," in *ICPP*, 2012.

[35] M. Lin, Z. Liu, A. Wierman, and L. L. H. Andrew, "Online algorithms for geographical load balancing," in *Proc. Int. Green Computing Conf.*, San Jose, CA, 5-8 Jun 2012.

[36] D. Aikema, R. Simmonds, and H. Zareipour, "Data centres in the ancillary services market," in *Proceedings of the 3rd International Green Computing Conference (IGCC'12)*, 2012.

[37] "Kvm, http://www.linux-kvm.org."

[38] J. S. Chase, D. C. Anderson, P. N. Thakar, and A. M. Vahdat, "Managing energy and server resources in hosting centers," in *In Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, 2001.

[39] K. Le, R. Bianchini, J. Zhang, Y. Jaluria, J. Meng, and T. D. Nguyen, "Reducing electricity cost through virtual machine placement in high performance computing clouds," in *Proceedings of SC'11*, 2011.

[40] N. Buchbinder, N. Jain, and I. Menache, "Online job-migration for reducing the electricity bill in the cloud," in *Proceedings of the 10th international IFIP TC 6 conference on Networking - Volume Part I*, ser. NETWORKING'11. Springer-Verlag, 2011.

[41] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper, "Free lunch: exploiting renewable energy for computing," in *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS'11)*. USENIX, 2011.