Cuckoo: Opportunistic MapReduce on Ephemeral and Heterogeneous Cloud Resources

Jean-Emile Dartois, Heverson Ribeiro, Jalil Boukhobza, Olivier Barais

Cuckoo: Opportunistic MapReduce on Ephemeral and Heterogeneous Cloud Resources

Jean-Emile Dartois*†, Heverson B. Ribeiro*, Jalil Boukhobza*‡, and Olivier Barais*†
*b<>com Institute of Research and Technology, †Univ. Rennes, Inria, CNRS, IRISA, ‡Univ. Bretagne Occidentale
Email: jean-emile.dartois@b-com.com, heverson.ribeiro@b-com.com, boukhobza@univ-brest.fr, barais@irisa.fr

***Abstract*—Cloud infrastructures are generally over-provisioned for handling load peaks and node failures. However, the drawback of this approach is that a large portion of data center resources remains unused. In this paper, we propose a framework that leverages unused resources of data centers, which are ephemeral by nature, to run MapReduce jobs. Our approach allows: *i*) to run efficiently Hadoop jobs on top of heterogeneous Cloud resources, thanks to our data placement strategy, *ii*) to predict accurately the volatility of ephemeral resources, thanks to the quantile regression method, and *iii*) for avoiding the interference between MapReduce jobs and co-resident workloads, thanks to our reactive QoS controller. We have extended Hadoop implementation with our framework and evaluated it with three different data center workloads. The experimental results show that our approach divides Hadoop job execution time by up to 7 when compared to the standard Hadoop implementation.**

***Keywords*-cloud, scheduling, unused resources, ephemeral resources, spare resources, data placement, big data, mapreduce, hadoop, quantile regression.**

I. Introduction

Advances in technologies such as smart-phones and the Internet of things led us to a data deluge. According to recent estimations [**?**] by 2025 the amount of data generated will be about 160 zettabytes. MapReduce [1] is a programming model proposed by Google for processing such large amounts of data while providing high performance and fault tolerance. Hadoop [2] is an Open-source implementation of MapReduce that runs across clusters of a large number of computing nodes. Although processing massive data requires a significant amount of computing resources, maintaining such a large-enough dedicated infrastructures to process multiple types of jobs is undoubtedly expensive.

Cloud computing provides on demand access to scalable, elastic and reliable computing resources. Although these features make Cloud infrastructures good candidates for processing Hadoop workloads, a clear drawback is their operation cost. Furthermore, Cloud computing data centers are often over-provisioned in order to cope with workload variations [3] and nodes failure. This over-provisioning increases the Total Cost of Ownership (TCO) for Cloud providers and results in a low average resource utilization. In a previous study [4], authors shown that the average CPU usage lies between 20% to 50% on several data centers. Some studies proposed to reclaim these unused resources and offer them at a cheaper price [5] to increase resource utilization. This led to a benefit increase of 60% for Cloud providers [3].

Therefore, a promising alternative for optimizing the cost of processing data-intensive applications on Cloud infrastructures is to opportunistically exploit their allocated but unused computing resources. In order to achieve that some challenges must be tackled.

- **Cloud heterogeneity**: Hadoop is designed to run on homogeneous and dedicated clusters of nodes, whereas Cloud infrastructures are built upon heterogeneous resources to avoid vendors lock-in effect and due to frequent hardware updates. Running Hadoop on environments with different computing capacities may significantly degrade its performance [6]. Hadoop distributes data chunks uniformly across nodes and expects them to run tasks with the same execution time. Thus, running Hadoop in heterogeneous systems requires data chunks to be reallocated to feed available computing slots. This creates network traffic overhead, and therefore degrades the system performance. Hence, Designing a resource-aware placement strategy is a necessary feature that can reduce subsequent chunk reallocation in order to avoid degrading the overall performance of Hadoop jobs.
- **Resources volatility**: In Cloud systems, users are able to unilaterally reserve, consume and release computing resources on-the-fly. As a consequence, exploiting unused and ephemeral resources to run Hadoop jobs raises some issues. First, Hadoop relies on replication to provide reliable storage and fault tolerance. Relying on resources that may be preempted at any time increases the cost of maintaining replicas. Second, Hadoop does not replicate intermediate results. If resources holding these results are preempted, the resulting data vanishes and its associated reduce task may stall and require the re-execution of the initial map task. Clearly, the way with which users claim their allocated resources plays a fundamental role not only on the execution time of a Hadoop job, but also on its termination. Hence, it is necessary to have a precise estimation of the unused resources to exploit them efficiently in order

to minimize the total cost of ownership and the cost of processing Hadoop applications on the Cloud.

- **Users SLA guarantee**: When running Hadoop jobs on allocated but unused resources, one should hedge against violating SLA QoS guarantees for the users having reserved those resources. This may be caused either by bad interference between co-located users and deployed Hadoop jobs or by sudden changes in users behavior. Several studies have underlined that co-located jobs may interfere and result in unwanted performance glitches. This may be due to some hardware, system mechanisms (e.g. SSD, CPU, memory) or virtualization interference. On the other hand, in Cloud systems the nature of workloads is highly heterogeneous and their intensity may significantly vary (*i.e.,* abrupt grow or shrink) according to users behavior [7]. These unexpected changes makes prediction errors unavoidable. Thus, it is necessary to have the ability to quickly allocate, react and adapt the unused resource provisioning in a way to avoid degrading the QoS for the users that have reserved those resources.

In this paper we try to tackle the aforementioned challenges in order to make it possible to exploit unused resources on Cloud infrastructures in an opportunistic way to process Hadoop data-intensive workloads.

Our approach relies on three mechanisms. *i*) a Data placement planner to cope with cloud heterogeneity, *ii*) a Forecasting builder to predict resource volatility, and *iii*) a QoS controller to ensure users SLA guarantee by avoiding interference. The data placement planner relies on the Forecasting builder and decides about the distribution of Hadoop chunks to process according to resource availability. The Forecasting builder relies on quantile regression and machine learning algorithms to accurately predict the amount of unused resources and their availability (volatility) to feed the data placement planner. The QoS controller is used to avoid Hadoop interference on users workloads of the Cloud provider. It achieves that by increasing and decreasing the allocated resources to Hadoop containers on-the-fly.

We evaluated our approach using traces of three months of resources usage from three different data centers (*i.e.,* two private companies, and one university). We compared native Hadoop job execution time with our solution. Our simulation results show that Cuckoo improves Hadoop native job execution time between 500% and 700% on ephemeral resources.

The remainder of this paper is organized as follows. Section II presents some background information. Then, we describe our methodology in Section III. Section IV details the experimental evaluation we have performed. Section V presents some limitations of our approach. Section VI compares our approach with some related works. Section VII concludes the paper.

II. Background

In this section, we introduce the key concepts of MapReduce paradigm through its Hadoop implementation.

A. MapReduce programming model

MapReduce is a programming model, inspired by Lisp programming language, and proposed for processing large data sets, potentially using hundreds or thousands of distributed machines [1]. The MapReduce model hides complex tasks, such as partitioning large data sets, scheduling and executing programs across distributed computers, dealing with failures, and handling inter-machine communication from users. This is done with a simple abstraction based on two phases, namely *map* and *reduce*. For each phase, the user writes a specific function (i.e., one map and one reduce function). The map function takes an input data set and outputs a set of intermediate *<key,value>* pairs. After that, the intermediate pairs are grouped by the same key. Then, each set of values corresponding to a single key is forwarded to the reduce function. Finally, each reduce function merges the values trying to form a smaller set of values.

B. Hadoop framework architecture

Hadoop has a master/slave architecture organized into two main layers. The first layer consists of a single master node called *Jobtracker* and a set of slave nodes called *Tasktrackers*. The second layer consists of a master node called *NameNode* and slave nodes called *DataNodes*. *Tasktrackers* are in charge of executing map and reduce functions, while *DataNodes* store chunks of input data. Users interact with a Hadoop cluster by means of a *ClientNode*, which is used to send the input data, map and reduce functions to the cluster.

Specifically, a Hadoop user sends its map and reduce functions to *ClientNode*, which in turn, sends them the *Jobtracker*. Concurrently, *ClientNode* fetches the *block allocation* information (i.e., chunk-to-node mapping) from the *NameNode*. Then, the *ClientNode* splits the input file into even-sized data chunks and streams them to *DataNodes*, which are randomly replicated across the cluster for fault-tolerance. Hadoop runs map and reduce tasks simultaneously. Each task occupies one single processing slot, which is released when the task is completed. When a *Tasktracker* has an empty slot, it sends a *hearbeat* message to the *Jobtracker* requesting a new task. The *Jobtracker* scheduler keeps assigning new tasks to available *Tasktrackers* until all the tasks are done. Hadoop scheduling algorithm favors data locality and does not consider other factors such as system load and fairness.

III. Cuckoo: a Mechanism for Exploiting Ephemeral and Heterogeneous Cloud Resources

Our main goal is to provide a framework that leverages unused Cloud resources to run Hadoop jobs efficiently without interfering with the co-located workloads.

Our proposed architecture consists of the following actors.

- **Farmers**: data center owners. Organizations that seek to reduce their TCO by making unused computing resources available to other users.
- **Customers**: two types: *i) regular customers*: organizations that buy regular Cloud resources from the operators, *ii) ephemeral customers*: organizations that want to process data-intensive applications on the Cloud at a lower cost. In this paper, we are more concerned with the latter.
- **Operators**: organizations that are the interface between farmers and customers. Their objective is to minimize *farmers* TCO by offering unused resources to *Customers* while insuring SLA [1].

A. The Cuckoo Framework Architecture Overview

The Cuckoo framework relies on three modules, each of them addressing a specific challenge previously described in Section I (i.e., resource volatility, heterogeneity, SLA guarantees).

- **Forecasting builder**: This module predicts (as accurate as possible) the future resource utilization at the host level, and therefore the amount of unused resources and their availability. The Forecasting builder considers both CPU and memory as resources and its main goal is to estimate the volatility of resources.
- **Data placement planner**: This module uses the predictions of the Forecasting builder and applies a placement strategy in order to distribute data chunks across the cluster hosts. The Data placement planner solves the heterogeneity of available resources by tuning data chunks allocation according to CPU availability and volatility.
- **QoS controller**: This module guarantees that running Hadoop jobs of *ephemeral customers* do not interfere with regular workloads of *regular customers* in order to ensure the SLA. The QoS controller continuously monitors the resource utilization to detect if *regular customers* could be impacted by *ephemeral customers*. If it is the case some corrective actions are triggered. It also has a preventive mechanism that consists in preserving a certain amount of unused resources to absorb workload variation. To this amount of preserved resources we refer to as *safety margin*.

Figure 1 presents both actors and modules and shows how they interact among each other. The *Customer* starts by submitting *(1)* a Hadoop job using the *ClientNode*. Then, the *JobTracker* sends *(2)* a request to the *Data placement planner* to check if the *Operator* is able to provide enough resources to process the job within a time window of 24 hours. In order to verify that, *Data placement planner* retrieves *(3)* the latest resource predictions, which are continuously updated by the *Forecasting builder* module, and creates the block allocation information, which maps chunks to nodes for that specific job. After that, the *JobTracker* replies *(4)* to the *ClientNode* with either an acceptance or a rejection message depending on the amount of available resources. Following, the *ClientNode* fetches *(5)* the block allocation map and sends *(6)* the chunks to the *DataNodes*. Finally, the *QoS Controller* monitors *(7)* the real-time utilization of unused resources in order to adapt the amount of resources allocated to containers that run *TaskTracker* nodes, in the case of interference.

[1]One may note that the farmer and the operator role can be played by the same organization

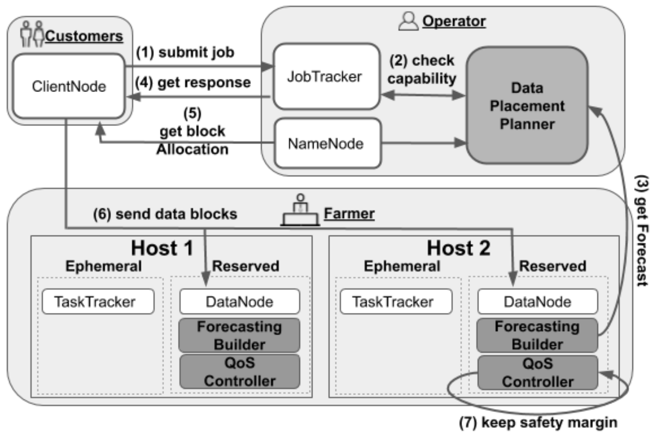


Figure 1. Overview of the Cuckoo architecture

B. Forecasting Builder

The objective of the *Forecasting Builder* module is to estimate the future amount of used resources for each host. By doing so, it can to estimate the available resources for running Hadoop jobs.

State-of-the-art studies have discussed [8] how to select the appropriate learning algorithm(s) to forecast time series with learning algorithms such as Autoregressive (AR), Integrated Moving Average (ARIMA) and other more complex algorithms such as Recurrent Neural Network (RNN), Support Vector Machine (SVM), Long short-term memory (LSTM), Gradient Boosted Decision Trees (GBDT), and Random Forest (RF). In a previous work [4] we have shown that quantile regression is a relevant approach to reclaim unused resources with SLA requirements. This work has shown that quantile regression may increase the amount of savings by up to 20% compared to traditional approaches.

Quantile regression provides the accuracy of machine learning algorithms with the flexibility of quantiles. Moreover, quantiles make it possible to reason about the trade-off between the amount of reclaimed unused resources and the potential SLA violations. In our work, we chose to use Gradient Boosting Decision Tree which gives the best trade-off between prediction accuracy and training time in order to forecast a 24-hour time window. We use quantile regression to implement our forecasting builder module.

C. Data Placement Planner

The objective of the *Data Placement Planner* is to find the best data block mapping in order to minimize the overall execution time given the available resources by minimizing data transfers. To achieve that, we use a modified version of the Weighted-Round-Robin (WRR) algorithm. WRR is designed to handle hosts with different processing capabilities by assigning a different weight to each [9].

In our approach, the weight was calculated by taking into account the predicted usage of resources and the processing capability for each host, estimated in *GFLOPS*[2] within a time interval of 24 hours. Then, data chunks are distributed proportionally to the weight assigned to each host. That is, hosts with higher weight receive more data chunks to process than hosts with lower values.

The safety margin value (described in Section III-D) denoted by sm is also taken into account. This value is used to remove the corresponding proportion of resources from the pool of unused resources that is the input of the *Data Placement Planner*. As mentioned before, the sm value is used to absorb unpredictable workload behavior and forecasting errors. For instance, if the forecast builder estimates that 12 cores are used in a 32 cores machine, then 20 cores would be available during the next 24-hour time window. If sm value is 10%, then 2 cores (from the 20 available) are removed from the the pool and only 18 cores are considered.

The following algorithms describe *i)* the process of calculating the host weight (Algorithm 1), and *ii)* the data chunk placement strategy (Algorithm 2).

Algorithm 1 Host Weight Calculation

```
1:  function CALW($host_{id}, sm, ti, sp$)
2:      $weight \leftarrow 0$
3:      $cap = \texttt{getFlops}(host_{id})$
4:      $predUsage = \texttt{ForecastingBuilder}(host_{id}, ti, sp)$
5:      for each $load \in predUsage$ do
6:          $load \mathrel{+}= sm$
7:          if $load < 100$ then
8:              $weight \mathrel{+}= (cap - ((load/100) * cap)) * sp$
9:          end if
10:     end for
11:     return $weight$
12: end function
```

Calculating the host weight: : Algorithm 1 has four input parameters: $host_{id}$, a safety margin (*sm*), time interval (*ti*) and sampling period (*sp*). The parameter *ti* is the time for which we calculate the weight. In our case, we used a *ti* of 24 hours during which we measure hosts resources usage with a sampling period (*sp*) of 3 minutes, thus 480 measures every 24 hours.

[2]Giga Floating Point Operations per Second

First, we retrieve the maximum processing capacity (*cap*) for the selected host `getFlops`$(host_{id})$ at *line 3*. Then, we request the Forecasting Builder module to estimate the available amount of resources (*predUsage*) for this host `ForecastingBuilder`$(host_{id}, ti, sp)$ at *line 4*. The *Forecasting Builder* returns a set of ti/sp data prediction points. Then, we iterate over these *predUsage* data points to compute the weight of the selected host *(lines 5–10)*. For each predicted data point, we add the safety margin to the predicted load *(line 6)*. Then, if the total used load is under 100 % (*i.e.,* the host has some unused resources), the weight is calculated by subtracting from the total processing capacity of the host noted *cap* the `predUsage` and *sm*. We then multiply the result by the specified sampling period to integrate the duration (*line 8*). The higher the free CPU resource and capacity, the higher the weight.

Data placement strategy: Algorithm 2 starts by computing weights for each host using Algorithm 1. For each job, first we initilize a matrix of Booleans for the chunk mapping called *blockAllocation* at *line 3* and we get the chunk replication factor used by Hadoop with function `getReplication()` at *line 4*. Second, for each chunk of the job *(lines 5–12)*, we retrieve the estimated processing costs using `getEstimatedTaskCost`$(chunk_{id})$ function at *(line 6)*. In this work, we have used fixed costs (see Section IV) but map or reduce tasks costs could be estimated as proposed by [10]. Then, we select a host according to the assigned weights (based on WRR) at *line 8*. The hosts with higher weights are selected first. Next, we update the matrix *blockAllocation* to indicate that the chunk ($chunk_{id}$) has been placed on the chosen host ($host_id$) at *line 9*. Finally, we dynamically update the weight of the host by decreasing its value according to the used resources (*cost*) and to updated predictions (*line 10*). We repeat these three steps for the default number of replicas (i.e., *nbReplicas*) initialized for Hadoop (lines 7–12). If there is not enough resource for processing a chunk, a rejection message is sent. When all the chunks of all the jobs have been placed, we send the allocation matrix *blockAllocation* to the *NameNode (line 14)* and an acceptance message. Finally, the block allocation matrix is retrieved by the *ClientNode*.

D. QoS Controller

The QoS controller implements a mechanism that reacts to under estimation of the used resources from the Forecasting builder (*e.g.,*. As discussed before, prediction errors may exist due to an unexpected variation of the *regular customers* workloads. The reactive policy of the QoS controller checks if the *regular customers* workloads are using more than a predefined threshold of the safety margin (tuned to 50% in our experiments). In this case, the allocated resources for the

Algorithm 2 Data Placement Algorithm

```
1: $weights = initWeights()$
2: for each $job \in JobTracker.all()$ do
3:   $blockAllocation[nbChunks, nbHosts] = false$
4:   $nbReplicas = getReplication()$
5:   for each $chunk_{id} \in job.chunks$ do
6:     $cost = getEstimatedTaskCost(chunk_{id})$
7:     repeat
8:       $selectedH = selectHost_{id}(hosts, weights)$
9:       $blockAllocation[chunk_{id}, selectedH] = true$
10:      $weights[host_{id}] = updateW(selectedH, cost)$
11:      $nbReplicas - -$
12:    until $nbReplicas$==0
13:  end for
14:  $send(job, blockAllocation)$
15: end for
```

ephemeral customers jobs must be reduced or completely released.

The QoS controller manages both the CPU and memory resources. In order to release resources the QoS controller proceeds as follows. The CPU control is done by adjusting dynamically the hard limits of the CPU cycles that a container is able to consume. In this way, Hadoop jobs cannot use more CPU than the amount of time set for the container. As a consequence, the map or reduce tasks will be slowed down without affecting regular customers workloads.

For the memory resource, Cuckoo has a more aggressive strategy and it acts as a system memory killer. Cuckoo kills proportionally the amount of map or reduce tasks necessary to free the safety margin related to memory occupation.

In case of CPU and/or memory starvation (*i.e.,* only the safety margin resources are available) the container is killed.

IV. Experimental Validation

This section describes the experiments conducted to validate the efficiency of the Cuckoo framework.

A. Experimental Methodology

We will try to answer three research questions (RQ) in order to tackle the 3 challenges mentionned in the section I (*i.e.,* resource heterogeneity and volatility and QoS guarantee):

RQ1: What is the overall performance of Cuckoo compared to native Hadoop implementation ?

RQ2: How does the forecasting builder accurately model the volatility of resources ?

RQ3: What is the effectiveness of Cuckoo with regards to the number of remote tasks?

We have conducted several experiments to evaluate our solution. We used a 3-months production data set from three different data centers and compared Cuckoo to standard Hadoop implementation. In our evaluations, we used two configurations related to the number of map tasks that is 514 and 640, corresponding to two different data sets of 40GB and 32GB respectively in a network of 50Mbps. In both cases we used 40 reduced-tasks. The processing costs for each Map and Reduce task is equal to 3100 FLOPS/Byte and 6300 FLOPS/Byte for the map and 1000 FLOPS/Byte for the reduce tasks.We set the chunk size to 64 MB and configured Hadoop to host three replicas for each chunk (default configuration) and each *TaskTracker* to run 20 slots of map and reduce tasks. According to [11], usually each task needs between 2 GB and 4 GB of memory which means that for a machine with 48GB of memory the Hadoop TaskTracker could run between 10 and 20 tasks in parallel.

The experimentation has three phases: *i*) infrastructure initialization, *ii*) deployment, and *iii*) injection. The infrastructure initialization phase configures the physical machines (*i.e.,* speed, number of cores, memory), the network (*i.e.,* topology, available bandwidth, latency) according to the three data centers. Then, the deployment phase consists in launching the Hadoop and Cuckoo modules (*e.g., JobTracker*, *DataNode*, *Data Placement Planner*). Finally, the injection phase consists in increasing and decreasing the CPU and Memory load over time according to data centers traces. The injection is done by replaying the traces from three data centers. As we fixed the forecast window to 24 hours on three months, this gives 92 windows per host.

Our experiments were performed using Simgrid 3.20 simulation tool and a customized version of MRA++ MapReduce [12] for handling the three phases.

B. Data sets

Each data set corresponds to a specific data center. The largest data center is PC-2 (i.e., private company 2) with 27 hosts providing a total of 3552 GFLOP/s and 3.8TB of RAM memory, followed by PC-1 and University. Table I shows the overall capacity of all data centers.

Table I
Total capacity of each data center

Name	Number of Hosts	CPU [GFLOP/s]	RAM [TB]
PC-1	9	2208	1.2
PC-2	27	3552	3.8
University	10	1363	1.5

Moreover, these data centers are heterogeneous. The PC-1 has six different configurations among its nine hosts, PC-2 has 13 different configurations among its 27 hosts, and finally the University data center has 6 different configurations. The average resource utilization of each data center is 13% for PC-1, 4% for PC-2 and 6% for University. These results motivated us toward reclaiming unused resources.

C. Experimental Results

We evaluated the overall execution time of a job according to the safety margin value, the number of remote tasks and the number of rescheduled tasks.

*1) **RQ1-Job Execution Time**:* To evaluate the benefits of Cuckoo compared to Hadoop we compared the job execution time according to different configuration of safety margin (*i.e.,* 0%, 5%, 10%, 15%, 20%, 25%, 30%).

Figure 2 shows the resulting job execution time. We observe that the minimum median completion time of Hadoop is 417 minutes for PC-1, 336 minutes for PC-2 and 377 minutes for University. On the other hand, the best median job completion time for Cuckoo is 57 minutes for PC-1, 49 minutes for PC-2 and 71 minutes for University with a safety margin of 5%. The smaller dispersion and best completion time was observed in the case of a 5% safety margin for all data sets. Then, with a safety margin greater than 5%, we notice that the job execution time increases. This is due to the fact that by increasing the safety margin size it leads to a decrease on the amount of reusable resources and thus the slow down of the Hadoop jobs. In addition, we observed that the best safety margin is the same regardless the data set evaluated. This means that the prediction of the forecast builder is accurate. Cuckoo is 7 times faster than native Hadoop strategy for PC-1 and PC-2 and 5 times faster for the University.

*2) **RQ2-Effetiveness of the Forecast builder**:* As mentioned in Section I, one way to increase data center utilization is to reclaim unused resources to run ephemeral containers that are executing map or reduce tasks. However, such containers could be evicted by the *QoS Controller* in case of interference with the regular customers workloads (see Section III-D). It means that the task should be relaunched causing a resource waste. Figure 3 shows the number of relaunched tasks for the three data sets with a safety margin of 0%. The lower the number of relaunched tasks, the better the volatility taken into account by the Forecast builder.

We observe that with the standard implementation of Hadoop, more than 15% of the tasks were relaunched while only less than 5% are in case of Cuckoo for the three data sets (3 times less). This confirms that the our prediction module helps Cuckoo to handle the volatility of resources efficiently.

In Cuckoo, PC-2 has the higher percentage of relaunched tasks with about 5% while the University the slower, with less than 1%. This means that PC-2 is wasting more resources when compared to University and PC-1. Moreover, a high number of relaunched tasks may lead to remote tasks, as explained in Section IV-C3.

*3) **RQ3- Effectiveness of Cuckoo and Remote Tasks**:* In this experiment we evaluated the percentage of remote tasks generated by Cuckoo and we compare it to the ones generated by Hadoop standard implementation. We have measured data movement during the experiments (see Section IV-A). A task becomes remote when a *TaskTracker* has an empty slot and no more available local chunks. In this case, a chunk is downloaded to be processed. In heterogeneous environments this situation is more likely to happen. So, by estimating the number of remote tasks we evaluate the ability of Cuckoo to manage heterogeneity and volatility.

Figure 4 shows the percentage of remote tasks. One may observe that Cuckoo outperforms standard Hadoop for all data sets. In the case of PC-1, Cuckoo has reduced the percentage of remote tasks by about 7 times, while in PC-2 and University by 6 and 19 times respectively. This confirms that Cuckoo handles both heterogeneity and volatility of resources effectively. As discussed in Section I, remote tasks take longer to execute as compared to local tasks due to the required data transfer time.

We also observe that PC-2 has more remote tasks. This can be explained by the fact that PC-2 is the largest data center with 27 hosts and since data chunks are distributed across hosts, more data transfers are generated. So for a given number of chunks to process, the higher is the replication factor, the lower the number of data movements are. In addition, the lower the number of hosts, the lower data movements for a given replication factor.

We conclude that for all three tested datasets, Cuckoo outperforms Hadoop in all the cases. This is explained by the fact that our data placement and the predictive provisioning strategies together make Cuckoo volatility and heterogeneity aware, and therefore able to required less remote and relaunched tasks, when compared to Haddop standard implementation.

V. Limitations

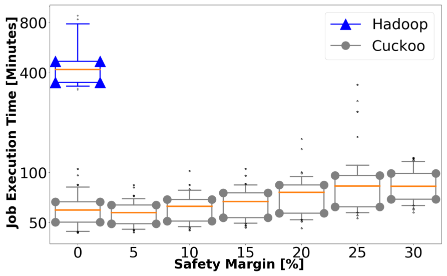
There are some potential issues that may impact on the results of this study. We will consider these issues in a future work. In the following we highlight some of these issues.

- We did not consider memory, network and storage for the data placement decisions. These variables may have an impact on the overall performance.
- The Map/Reduce tasks are single-threaded and cannot leverage the computing power of more than one core.
- The sampling period of the data center traces is 3 minutes which means that we cannot measure violations with smaller time sampling.
- We did not consider how co-located application workloads may interfere on Map or Reduce tasks. We have considered only a fixed capacity per host, while the capacity may depend on all running applications [13]

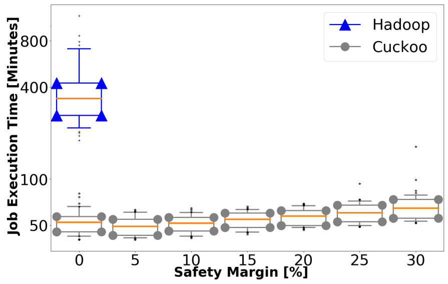
VI. Related Work

In this section, we discuss some studies that have already explored MapReduce in dynamic and heterogeneous environments.

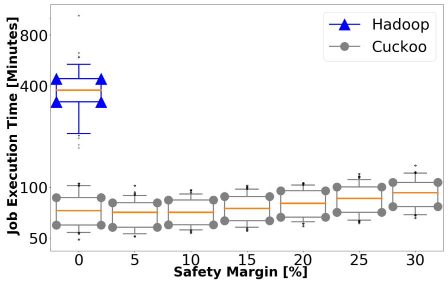
Many studies have already shown that heterogeneity significantly impacts Hadoop performance. In [14] authors proposed to improve data locality by delaying the task scheduling by a small amount of time for short jobs, which significantly improved performance. In [6] authors proposed



(a) **Private Company 1**



(b) **Private Company 2**



(c) **University**

Figure 2. Job Execution Time for standard Hadoop and Cuckoo

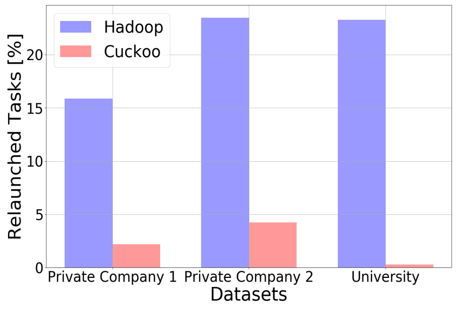


Figure 3. Median Percentage Relaunched Tasks comparison between Hadoop and Cuckoo

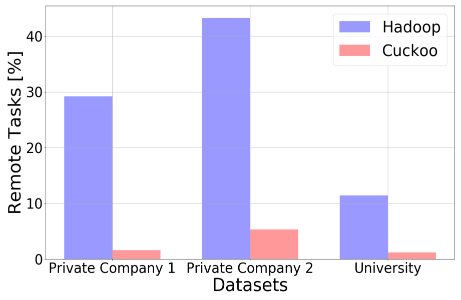


Figure 4. Median Percentage Remote tasks comparison between Hadoop and Cuckoo

to prevent from incorrect execution of speculative tasks by defining a fixed threshold in which the scheduler is able to select tasks to speculate. Their approach have improved the response time of MapReduce jobs by a factor of 2 in large heterogeneous clusters. In both cases they have improved execution time of jobs in heterogeneous environment. In our case, we consider not only the heterogeneity but also the volatility of nodes.

Some approaches considered the case of an open shared system and proposed a placement strategy that dispatches data chunks to hosts based on their availability rate [15]. The drawback of such approaches is their assumption about the inter-arrival time of interruptions to be independent and identically distributed across the system. There are some studies that handled volatility in open systems by using a small set of dedicated nodes in order to ensure the minimum amount of resources required to execute MapReduce jobs [16]. In [17] authors considered the case of pervasive grids and monitored nodes capacity (*i.e.,* processors and memory) to improve Hadoop scheduler decisions. Differently from those previous work, our contribution targets environments such as private Cloud data centers. In [18], authors have investigated the use of volatile spot instances as accelerators in public Clouds without considering any SLA constraints, while in [19] a hybrid infrastructure (i.e., mix of Cloud and volunteer computing) with up to 25% of unstable nodes is used to continuously execute Hadoop jobs without loosing performance.

Finally, some studies used a predictive approach to improve data locality and Hadoop performance. In [20] authors used a linear regression model to predict the execution time of map tasks which is used to prefetch input data. Another approach [21] used the Naive-Bayes-classifier method to predict node availability and therefore improve Mapreduce scheduler performance.

Our work improves state-of-the-art solutions by simultaneously considering both heterogeneity and volatility. Besides that, our approach guarantees SLA without interfering with other workloads sharing the same physical resources.

VII. Conclusion and Perspectives

Data center resources are underused. They are heterogeneous and their usage is not balanced among hosts. We argue in this paper that those resources could be used to process Big data Hadoop application at a low cost. In order to do so, several challenges need to be tackled: heterogeneity and volatility of resources and isolation with regards to regular customer workloads.

To tackle these issues, we have developed a heterogeneity and volatility aware data-placement strategy called Cuckoo. Volatility and heterogeneity are managed by our proposed forecasting and resource-aware data placement strategies. In addition, a QoS controller is used to avoid any interference with regular workloads by means of a safety margin.

Our results show that Cuckoo outperforms standard Hadoop implementation by a factor of 5 to 7, while avoiding any interference with regular customer workloads.

As future work we plan to evaluate check-pointing fault-tolerance techniques to avoid using reserved resources for hosting intermediate data. Although we only used simulations in this work, we plan to evaluate our strategies using a testbed for experimental research, such as Grid5000. We are also planning to use GPU and other processing elements.

Acknowledgment

This work was supported by the Institute of Research and Technology b-com, dedicated to digital technologies, funded by the French government through the ANR Investment referenced ANR-A0-AIRT-07.

References

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

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, pp. 1–10, IEEE, 2010.

[3] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes, "Long-term SLOs for reclaimed cloud computing resources," in *Proceedings of the 5th ACM Symposium on Cloud Computing*, pp. 1–13, ACM, 2014.

[4] J.-E. Dartois, A. Knefati, J. Boukhobza, and O. Barais, "Using quantile regression for reclaiming unused cloud resources while achieving sla," in *Proceedings of the 10th IEEE International Conference on Cloud Computing Technology and Science*, pp. 89–98, IEEE, 2018.

[5] P. Marshall, K. Keahey, and T. Freeman, "Improving utilization of infrastructure clouds," in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 205–214, IEEE, 2011.

[6] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments.," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pp. 29–42, USENIX, 2008.

[7] M. C. Calzarossa, M. L. Della Vedova, L. Massari, D. Petcu, M. I. Tabash, and D. Tessera, "Workloads in the clouds," in *Principles of Performance and Reliability Modeling and Evaluation: Essays in Honor of Kishor Trivedi on his 70th Birthday*, pp. 525–550, Springer, 2016.

[8] M. Amiri and L. Mohammad-Khanli, "Survey on prediction models of applications for resources provisioning in cloud," *Journal of Network and Computer Applications*, vol. 82, pp. 93–113, 2017.

[9] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted round-robin cell multiplexing in a general-purpose atm switch chip," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 8, pp. 1265–1279, 1991.

[10] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang, "Hadoop performance modeling for job estimation and resource provisioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 441–454, 2016.

[11] E. Sammer, *Hadoop Operations: A Guide for Developers and Administrators.* " O'Reilly Media, Inc.", 2012.

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

[13] J.-E. Dartois, J. Boukhobza, A. Knefati, and O. Barais, "Investigating machine learning algorithms for modeling ssd i/o performance for container-based virtualization," *IEEE Transactions on Cloud Computing*, vol. 14, pp. 1–14, 2019.

[14] 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 *Proceedings of the 5th European Conference on Computer Systems*, pp. 265–278, ACM, 2010.

[15] H. Jin, X. Yang, X.-H. Sun, and I. Raicu, "Adapt: Availability-aware mapreduce data placement for non-dedicated distributed computing," in *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems*, pp. 516–525, IEEE, 2012.

[16] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "Moon: Mapreduce on opportunistic environments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 95–106, ACM, 2010.

[17] L. A. Steffenel, O. Flauzac, A. S. Charão, P. P. Barcelos, B. Stein, G. Cassales, S. Nesmachnow, J. Rey, M. Cogorno, M. K. Pinheiro, *et al.*, "Mapreduce challenges on pervasive grids," *Journal of Computer Science*, vol. 10, no. 11, pp. 2194–2210, 2014.

[18] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz, "See spot run: using spot instances for mapreduce workflows," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pp. 7–14, USENIX, 2010.

[19] J. Anjos, K. Matteussi, P. De Souza, C. Geyer, A. D. S. Veith, G. Fedak, and J. L. V. B. V. Barbosa, "Enabling strategies for big data analytics in hybrid infrastructures," in *Proceedings of the 16th International Conference on High Performance Computing & Simulation*, pp. 869–876, 2018.

[20] M. Merabet, S. mohamed Benslimane, M. Barhamgi, and C. Bonnet, "A predictive map task scheduler for optimizing data locality in mapreduce clusters," *International Journal of Grid and High Performance Computing*, vol. 10, no. 4, pp. 1–14, 2018.

[21] B. Tang, M. Tang, G. Fedak, and H. He, "Availability/network-aware mapreduce over the internet," *Information Sciences*, vol. 379, pp. 94–111, 2017.