Salamander: a Holistic Scheduling of MapReduce Jobs on Ephemeral Cloud Resources

Mohamed Handaoui, Jean-Emile Dartois, Laurent Lemarchand, Jalil Boukhobza

Salamander: a Holistic Scheduling of MapReduce Jobs on Ephemeral Cloud Resources

Mohamed Handaoui*‡, Jean-Emile Dartois*†, Laurent Lemarchand‡, Jalil Boukhobza*‡
*b<>com Institute of Research and Technology, †Univ. Rennes, Inria, CNRS, IRISA,
‡Univ Brest, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
Email: {mohamed.handaoui, jean-emile.dartois}@b-com.com, {lemarchand, boukhobza}@univ-brest.fr

***Abstract*—Most cloud data centers are over-provisioned and underutilized, primarily to handle peak loads and sudden failures. This has motivated many researchers to reclaim the unused resources, which are by nature ephemeral, to run data-intensive applications at a lower cost. Hadoop MapReduce is one of those applications. However, it was designed on the assumption that resources are available as long as users pay for the service. In order to make it possible for Hadoop to run on unused (ephemeral) resources, we have designed a heterogeneity and volatility-aware holistic scheduler consisting of three different components: (1) A MapReduce task and job scheduler that relies on a global vision of resource utilization predictions, (2) a scheduler-based data placement strategy that improves the data locality, and (3) a reactive QoS controller that ensures customers' service-level agreement (SLA) and minimizes interference between co-located workloads. Our framework makes it possible to take advantage of ephemeral resources efficiently. Indeed, for a given set of jobs, it reduces the overall execution time by up to 47.6% and an average of 18.7% as compared to state-of-the-art strategies.**

***Index Terms*—Cloud, Ephemeral Resources, Big Data, Hadoop, MapReduce, Job Scheduling, Task Scheduling, Data Placement.**

I. Introduction

Cloud Computing attracts increasing interest of companies as it provides on-demand access to scalable, elastic, and reliable computing resources with a pay-as-you-go pricing model. These Cloud features come at a price of over-provisioning resources to deal with workload peaks and node failure and thus meet customers' demands [1]. Operating and managing such infrastructure is costly, resulting in an increase in the Total Cost of Ownership (TCO) for Cloud providers and low average resource utilization (*i.e.,* between 25-35% for the CPU and 40-50% for the RAM [2], [3]).

One way to improve Cloud resource utilization and thus reduce the TCO is to reclaim and make profit of the unused part [1], [4], [5]. These are called ephemeral (*i.e.,* volatile) resources because it is imperative to be able to release them instantly if their owner (*i.e.,* the customer that reserved those resources) needs them. One must note that these resources are in fact heterogeneous, meaning that their hardware architecture may vary from one machine to another. These ephemeral resources are an opportunity to process big data workloads at a lower cost since they require a considerable amount of computing resources [6], [7].

Hadoop is an open-source framework used for distributed processing of big data volumes over a large set of computing nodes. Not only does Hadoop provide high performance but also fault tolerance [8]. In Hadoop, data are divided into chunks and distributed across the nodes. A Hadoop job consists of a set of tasks where each task processes a chunk of data. A task is considered as *local* if the chunk it processes is stored in the same node. Otherwise, it is considered as *remote*. Hadoop considers three main concepts to efficiently process data: (1) *job scheduling*, provides an order in which jobs are selected, (2) *task scheduling*, consists in deciding when and on which node to execute a task, while respecting dependencies between tasks of the same job, (3) *data locality*, the closer a task is to the data to process (*i.e.,* on the same node, rack, etc.), the faster the computation, which increases the overall throughput of the system.

Most Big data frameworks such as Hadoop were designed and developed with fault tolerance built-in in case of hardware failures [8], [9]. However, the reclaimed resources may become unavailable at a much higher rate than the hardware failures. This means that the default Hadoop implementation is not adapted to ephemeral resources. As a consequence, the running tasks (on the preempted resources) have to be rescheduled. This may lead to several issues namely poor resource utilization, slow execution of tasks and resource bottlenecks such as network traffic overhead due to remote or speculative tasks [4], [10]. In addition, when a node is performing poorly, Hadoop launches speculative copies of the node's tasks on other nodes for potential faster execution. In order to determine slow nodes, Hadoop assumes that all resources are homogeneous and thus the tasks are supposed to have the same execution time. However, this assumption does not hold in the case of *heterogeneous* resources. In fact, it has been shown that running Hadoop on such resources significantly degrades performance [6].

In this paper, we aim to maximize data centers utilization by efficiently exploiting *ephemeral* resources. This should be done without interfering with the provider's regular customers' workloads. In other words, we investigated how to design a scheduler for Hadoop jobs that runs on heterogeneous and ephemeral resources while guaranteeing SLA.

Several studies have been conducted to address these challenges. In [4], the authors propose a framework to opportunistically run Hadoop jobs by leveraging unused resources. This work is closely related to ours. The authors propose a data placement strategy that relies on predictions of resource utilization. However, this work only considers data placement

and no suitable task and job scheduling are performed. This may lead to slow execution and poor resource utilization as it will be shown in the experimental validation section. Other studies have also been realized to improve the default Hadoop scheduler. In [11] and [10], authors propose solutions that reduce the risk of recomputation in case of volatile resources by using either data checkpointing or by scheduling costly tasks on reserved resources. One issue with these studies is that they require a minimum subset of reserved resources to run, which one does not necessarily have.

Our contribution consists of a heterogeneity and volatility-aware framework for Hadoop job and task scheduling and data placement on ephemeral Cloud resources. We investigated three solving strategies for the scheduling problem to achieve a trade-off between the scheduling quality and computation cost. Our contribution can be summarized as follows:

- An optimization problem formulation of jobs and tasks scheduling in the context of Cloud ephemeral resources;
- A comparative study of three solving strategies consisting of (i) an exact approach based on Constraint Programming (CP) and two metaheuristic algorithms: (ii) a Genetic algorithm (GA) and (iii) Local Search-based (LS) algorithms;
- A data placement strategy based on the proposed task scheduler;
- A QoS Controller that ensures users SLA guarantee by avoiding interference between co-located workloads (Hadoop jobs and regular workloads);
- An evaluation on real traces of 42 hosts from three in-production data centers for a 70 days time period.

Our simulation results show that Salamander improves the job execution time up to 47.6% as compared to Cuckoo [4], and in most cases, eliminates completely relaunched tasks. Furthermore, the results show that the data placement strategy based on the scheduler eliminates most remote tasks.

The remainder of this paper is organized as follows. Section II provides brief background information. Section III details our contribution and the different algorithms used for solving the scheduling problem. Then, Section IV describes the experimental evaluation and the results obtained. In Section V, we explore some limitations of our contribution. In Section VI, we discuss related work. Finally, Section VII concludes the paper and discusses about future work.

II. Background

This section gives some background on MapReduce programming model as well as key concepts of Hadoop. We also present Cuckoo, a state-of-the-art study having a similar architecture as the one used in our contribution.

A. MapReduce programming model

MapReduce is a programming model used for distributed processing of large datasets [12]. Two main computations are used and must be defined by the user as Map and Reduce functions. The processing starts by splitting the input data into chunks and then passing them to the Map phase where the mapping function produces key-value pairs as output. The next phase called Shuffle consolidates the relevant records from the Map phase. Finally, in the Reduce phase, the output values from the Shuffle phase are aggregated and returned.

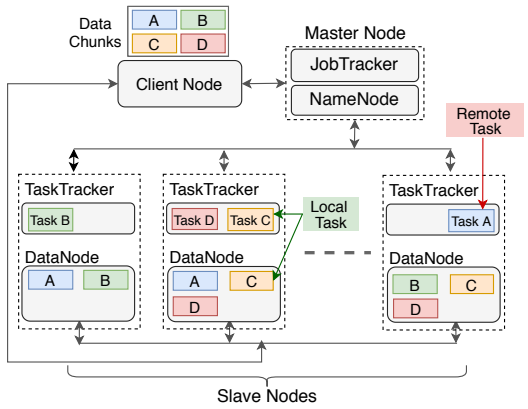


Fig. 1: Overview of Hadoop architecture

B. Hadoop MapReduce

Hadoop is an open-source implementation of Google MapReduce [8] to process large amounts of data and computation. Hadoop consists of two main components, a storage component called Hadoop Distributed File System (HDFS), and a processing component that is the MapReduce programming model. In Hadoop, there is a Master Node managing a set of Slave Nodes (see Fig.1).

Each node has two layers that are MapReduce and HDFS. The Master Node consists of a JobTracker and a NameNode. A Slave Nodes consist of a TaskTracker and a DataNode. The JobTracker is responsible of scheduling tasks on TaskTrackers and monitoring their execution. Each TaskTracker has a configurable number of Map and Reduce slots, which limits the maximum number of simultaneous tasks in a node. During the execution of tasks, a Slave Node signals (heartbeat) periodically (*e.g.,* 3 seconds) the Master Node to indicate that it is alive. If the heartbeat is not received for a long period of time (*e.g.,* 2 minutes) then the Slave Node is considered dead.

Users interact with the Master Node through the Client Node to submit computation in the form of Map and Reduce functions. The data to be processed is split into chunks (by default each chunk has a size of 128 MB [13]) and distributed uniformly to the DataNodes (see Fig.1). These chunks are replicated across the Slave Nodes with a certain replication factor (by default 3 replicas, *e.g.,* 2 in Fig.1) for fault tolerance. The NameNode contains metadata such as the number of chunks, the DataNode in which each chunk is stored, etc.

The default Hadoop task scheduler favors data locality. It is the process of moving tasks close to where the data chunk resides instead of the opposite. Two types of task execution are possible namely local and remote, which are differentiated according to the initial placement of data. A task is considered as local when both the task and the data chunk to process are initially placed on the same node (see Task C in Fig.1). Otherwise, it is a remote task (see Task A in Fig.1). The latter

case happens when nodes (with the data chunk to process) do not have sufficient resources to run a new task (due to limited slots). As a consequence, the chunk is transferred to the remote task's node. Hadoop scheduling strategy aims to minimize the number of remote tasks which in turn minimizes network bottlenecks and increases the performance of the system.

C. Cuckoo

Cuckoo [4] is a framework that leverages unused resources of data centers, which are ephemeral by nature, to run MapReduce jobs. The framework relies on three main modules:

1) **Forecasting builder:** this module is used to predict future resource utilization in a Cloud at the host level considering multiple resource metrics [2]. This module uses quantile regression to provide accurate predictions for reclaiming unused resources while guaranteeing SLA.
2) **Data placement planner:** this module is used to distribute data chunks across the cluster's DataNodes. It relies on the predictions from the *Forecasting builder* to distribute data chunks using a weighted round-robin algorithm that solves the heterogeneity and volatility of resources. As for tasks, Cuckoo uses the default Hadoop scheduler (see Section II-B) that favors data locality. Cuckoo implicitly improves the task scheduler by improving the data placement.
3) **QoS controller:** this module guarantees that running Hadoop jobs do not interfere with the regular customers' workloads in order to ensure the SLA. The QoS controller continuously monitors resource utilization to detect any interference. If it is the case, some corrective actions are triggered such as killing tasks. It preserves a certain amount of unused resources referred to as *Safety Margin* to absorb workload variation or mispredictions.

III. The Salamander framework

Our goal is to build a framework that allows for efficient execution of Hadoop jobs on heterogeneous and ephemeral resources while guaranteeing SLA. The main roles of our framework architecture are as follows:

- **Farmers:** data center owners, they seek to reduce their TCO by offering unused resources to other customers.
- **Customers:** two types of customers exist in this architecture. First, regular customers that buy and reserve stable Cloud resources. Second, ephemeral customers (customers using ephemeral resources) that want to process data-intensive applications on the Cloud at a lower cost.
- **Operator:** acts as the interface between farmers and customers and aims at minimizing farmers' TCO by offering unused resources to customers with SLA requirements.

A. Framework architecture overview

Our framework encompasses four main modules to improve the scheduling of MapReduce jobs on ephemeral resources:

1) **Forecasting builder:** this module was introduced in [2] and used in Cuckoo [4] for the data placement strategy. In our solution, we reused the predictions of this module in order to have a global overview of the amount of unused resources and their expected volatility for efficient scheduling of MapReduce jobs. This module is not detailed in this paper.
2) **Holistic scheduler:** in this module, we implemented three solving strategies to schedule MapReduce jobs and tasks, using Constraint Programming, Genetic and Local Search-based algorithms. The scheduler relies on predictions from the Forecasting builder and considers a safety margin for efficient execution of tasks.
3) **Data placement:** this module uses the task scheduling of the Holistic scheduler as a placement strategy to distribute data chunks across DataNodes. This strategy makes it possible to reduce significantly the number of remote tasks.
4) **QoS controller:** this module is an improved version of the QoS controller introduced in [4]. As explained earlier, the controller is a reactive mechanism that tries to eliminate interference between co-located workloads. While the previous version of the QoS controller relaunches tasks for any resource violation (*i.e.,* CPU, RAM), the upgraded version does so only for compressible resources (*i.e.,* RAM) and adjusts the incompressible ones (*i.e.,* CPU).

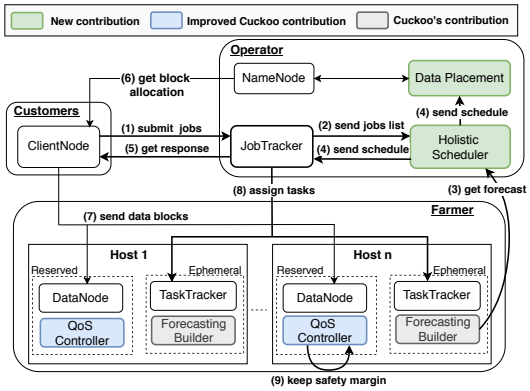


Fig. 2: Overview of Salamander's framework architecture

Fig.2 gives an overview of the Salamander architecture. It shows the different roles involved and the interaction between the aforementioned modules. First, the customers submit (1) MapReduce jobs using the *ClientNode*. Then, the *JobTracker* sends the list of jobs (2) to the *Holistic scheduler* which in turn requests the predictions (3) from the Forecasting Builder for the scheduling step. The module tries to schedule the submitted jobs in a 24-hour time window (forecasting horizon). This execution plan is then sent (4) to the JobTracker and the Data Placement module at the same time. Afterwards, a response is forwarded from the JobTracker to the ClientNode (5) indicating the schedulability of the jobs (whether it is feasible or not). Before the execution of any task, the ClientNode requests (6) the data chunks allocation from the NameNode in order to send them (7) to the appropriate DataNode (placed on reserved storage for temporarily hosting chunks and intermediate data). Finally, the JobTracker assigns the tasks (8) to TaskTrackers at a specific time using the generated scheduling. Periodically, the QoS Controller monitors (9) the utilization of resources in order to resize the amount of allocated resources to containers

(that run TaskTracker nodes) to minimize interference.

B. The Holistic Scheduler

This section describes the Holistic scheduler module. The scheduling problem is formalized as an optimization problem that we solve using different approaches, *i.e.,* CP, GA and LS.

1) Optimization problem formulation: First, the amount of unused resources in a cloud infrastructure is calculated as:

$$A(k,m,t) = C(k,m) * (1 - (U(k,m,t) + S(m))) \quad (1)$$

Where $A(k,m,t)$ is the amount of unused resources for a certain metric m (*e.g.,* CPU) in a given node k and time t. $C(k,m)$ is the capacity of a resource metric m in a node k. $U(k,m,t)$ represents the predicted percentage of resource usage in a node at a given time. $S(m)$ is the percentage of the safety margin (see Section II-C) for a resource metric m.

Then, we defined two objective functions that are lexicographically ordered for the scheduling problem:

$$maximize(\sum_{i \in J} p(i)) \quad (2)$$

$$minimize(\max_{i \in J}(ts(i) + Te(i,n(i))) \quad (3)$$

In function (2), $p(i)$ represents whether a task i was scheduled ($p(i) = 1$) or not ($p(i) = 0$). This function gives the total number of scheduled tasks and aims at maximizing it for a given time window. In function (3), $ts(i)$ is the starting time of a task i and $Te(i,n(i))$ is the estimated execution time of a task i in a given node $n(i)$ where the task is scheduled. This function calculates the time at which the latest task (*i.e.,* latest job) finishes its execution (*i.e.,* the makespan). This function aims at minimizing the makespan of the scheduled jobs. In fact, by maximizing the number of scheduled tasks and reducing the jobs' makespan, the overall infrastructure utilization increases.

Moreover, we have to fulfill three constraints for this scheduling problem:

$$\forall i \in J \ : \ 0 \leq ts(i) + Te(i,n(i)) \leq T \quad (4)$$

$$\forall i,j \in J \mid D(i,j) = 1 \ : \\ (n(i) \geq 0 \vee n(j) \geq 0) \implies (n(i) \geq 0 \wedge n(j) \geq 0) \\ \wedge (ts(j) \geq ts(i) + Te(i,n(i))) \quad (5)$$

$$\forall t \leq T, \forall k \in N, \forall m \in R \ : \\ \left(\sum_{\substack{\{i \mid (i \in J) \wedge (n(i)=k) \wedge \\ (0 \leq t - ts(i) \leq Te(i,n(i))\}}} Rq(i,m) \right) \leq A(k,m,t) \quad (6)$$

- In Eq.(4), the execution of all the scheduled tasks should be performed within a predefined time window. In other words, each task's ending time (*i.e.,* start time $ts(i)$ + execution time $Te(i,n(i))$) should be in the time window T.
- In Eq.(5), a job has to be either fully scheduled (*i.e.,* all of its tasks are scheduled) or rejected while respecting the dependencies according to MapReduce paradigm. $D(i,j)$ represents the dependency between a task i and task j ($D(i,j) = 1$, if task i depends on j, otherwise $D(i,j) = 0$). A task i is considered to be scheduled if it is assigned to a valid node $n(i)$ and a start time $ts(i)$.
- In Eq.(6), tasks can be scheduled on nodes only when there are sufficient available resources for the whole execution to avoid SLA violations. $Rq(i,m)$ represents the requirement of a task i in terms of a resource metric m. The sum of the requirements of all tasks scheduled on a node k running at a given time t should be smaller than $A(k,m,t)$.

TABLE I: Notation dictionary

Input variables: uppercase letters. **Output variables**: lowercase letters.

Type	Notation	Domain	Description
Input	N	-	List of available nodes' IDs
	J	-	List of MapReduce jobs with tasks' IDs
	R	-	List of resource metrics (*e.g.,* CPU, RAM)
	T	$\mathbb{N}$	Size of the scheduling window (*e.g.,* 24-hours with 3 minutes sampling = 480 points)
	$C(k,m)$	$\mathbb{N}$	Maximum capacity given a resource metric m in a node k
	$U(k,m,t)$	$[0,1]$	Predicted utilization of resource metric m in a node k at time t
	$S(m)$	$[0,1]$	Safety margin percentage for resource metric m
	$A(k,m,t)$	$\mathbb{N}$	Amount of available resources of metric m in a node k at time t
	$Te(i,n(i))$	$\mathbb{N}$	Estimated execution time of task i, it varies depending on the computational capacities of the assigned node $n(i)$
	$Rq(i,m)$	$\mathbb{N}$	Requirement of task i in terms of resource metric m
	$D(i,j)$	$\{0,1\}$	Presence of a dependency between task i and j
Output	$ts(i)$	T	Start time of task i
	$n(i)$	N	The node ID that the task i is assigned to
	$p(i)$	$\{0,1\}$	Presence of task i in the schedule, inferred from the start time $ts(i)$ or the task's node $n(i)$

2) Solving strategies: We propose three different solving strategies to schedule tasks. This is done for several reasons. First, from our study of the related work, we did not find a solution that schedules tasks using an exact approach but instead heuristics were used (see Section VI). Although it is known that task scheduling problem is either NP-hard or NP-complete [14] and thus exact solving strategies are not scalable, we had to test the limits of CP to draw conclusions about its scalability and usability for the studied problem. Nonetheless, we propose other solving strategies using metaheuristics which are heavily used for solving large optimization problems. In fact, many strategies exist and choosing one is not obvious. In [15], authors have curated important metaheuristics used in cloud scheduling. Among these metaheuristics, we selected a Multi-Objective Optimization (MOO) Genetic Algorithm and Local Search-based algorithm (both detailed below). These algorithms proved to be applicable to the context of cloud scheduling on stable resources. However, no prior work used them for scheduling on ephemeral resources, so it seemed

to be a reasonable choice. In what follows, we describe each of the proposed solving strategy.

Constraint Programming (*CP*): we used constraint programming to implement the exact approach. CP is a paradigm used to solve combinatorial problems such as scheduling [16]. We used CP optimizer [17] syntax to model the optimization problem formulation (see Section III-B1) in CP. CP optimizer uses time-based variables and built-in functions to accelerate solution search [17].

Local Search-based algorithms (*LS*): the proposed approach combines two local search-based algorithms namely Tabu Search [18] and Late Acceptance [19] to avoid getting stuck in a local optimum. The following details each component of the proposed LS approach:

- Initialisation: the idea is to find an initial random candidate solution that could be used as a good starting point of the search in order to reach better candidates. The initial candidate solution has to satisfy the problem's constraints formulated in Section III-B1. A random First Fit Decreasing (Algorithm.1) is used for such an initialization as it is efficient and simple to implement. In this algorithm, the decreasing part refers to the scheduling of the jobs with the longest execution time first (line 1). The random first fit refers to the fact that tasks are scheduled on the first randomly selected node with the earliest possible start time. In order to reduce the possible values of a task's start time during the search (line 4), we compute the earliest start time $tsMin$ which considers the dependencies between tasks. Then, we compute the latest possible start time $tsMax$ which takes into account the minimum estimated execution time of a task on all nodes. After that, a random node is selected and tested for its capacity to run a task (lines 5-12) with the earliest possible starting time in $[tsMin, tsMax]$.
- Operators: in order to explore the search space, operators are used to make local changes (*i.e.,* small changes) to candidate solutions. This means that these operators can produce infeasible candidate solutions to reach a feasible one. In this approach, three operators were used. One operator randomly selects a task and assigns it a random start time or node (even if it is not feasible). Another operator swaps the start time or node of two randomly selected tasks. The other operator selects a random task and tries to assign to it an earlier starting time to minimize the makespan.
- Evaluation: since the search operators can produce infeasible candidate solutions, we must start by checking whether the candidates are feasible according to the formulated constraints. If a candidate is, we evaluate its quality according to the two objective functions which are lexicographically ordered. This means that the evaluation is based on three properties ordered by 1) the feasibility of a candidate solution, 2) the number of scheduled tasks, 3) the makespan of the jobs. To implement the evaluation, we used three level scores, that is, hard, medium and soft score. The hard score represents the number of constraints violations by the current candidate (a solution is feasible if the hard score is zero). The medium score represents the number of scheduled tasks. Finally, the soft score holds the scheduling makespan.

Algorithm 1: Random First Fit Decreasing

```
1  jobs.sort();
2  foreach job in jobs do
3      foreach task in job.getTasks() do
4          [tsMin, tsMax] = getPossibleStartTimes(task);
5          for t = tsMin to tsMax do
6              node = selectRandomValidNode(task, t);
7              if node ≠ null then
8                  node.addTask(task);
9                  task.setStartTime(t);
10                 break;
11             end
12         end
13     end
14 end
```

Genetic algorithms (*GA*): the proposed approach is an MOO Genetic Algorithm using NSGA-III [20]. We used MOO as it gave us better results than Single Objective Optimization, this was also the case for other studies discussed in [15].

In this approach, a candidate solution (also called an individual or a chromosome) is represented by a set of tasks with their starting time and assigned nodes. The NSGA algorithm maintains a population of individuals and returns a subset of solutions according to the evaluation step (explained below) that satisfy the problem's constraints. Among the returned solutions, we choose the one that maximizes the number of scheduled tasks. If multiple solutions have similar scheduled tasks, then we consider the one that minimizes the makespan.

The following details each component in the proposed algorithm exploited by the NSGA-III framework:

- Initialization: individuals are initialized with a feasible solution using Algorithm.1.
- Evaluation: individuals are evaluated using the two formulated objective functions (see Section III-B1).
- Selection: the values of the objective functions are used to select individuals using tournament strategy [21].
- Crossover: the crossover selects two individuals (parents) for reproduction to create two new individuals (children). We used an adapted Order-based Crossover (OX2) [22] that allows us to generate feasible solutions while preserving the dependencies and resource capacities.
- Mutation operators: the mutation operators are used to explore the neighborhood of candidate solutions in the search space. We used several operators such as rescheduling random tasks, moving random tasks, balancing tasks between nodes, etc. Here we describe one algorithm that was able to improve the search. Algorithm.2 is used to reduce the makespan of the scheduled jobs by reassigning both tasks' nodes and start time. In order to avoid deteriorating the candidate solution, we make sure that the new makespan of the scheduling is better or equal to the current makespan. To do that, we first get the total $makespan$ of jobs (lines 1-2). Then, we select a random subset of tasks for mutation (line 3). For each task, we assign to it a random node and/or

start time without worsening the current total $makespan$ and while respecting the problem's constraints (lines 4-5). Finally, we attempt, when possible, to reduce the schedule makespan (lines 7) by assigning an earlier start time to the tasks without changing the assigned node.

Algorithm 2: Mutation operator

```
1  jobs = getScheduledJobs();
2  makespan = getJobsMakespan(jobs);
3  tasks = selectRandomTasksForMutation(jobs);
4  foreach task in tasks do
5      reassignNodeAndStartTime(task, makespan);
       /* makespan is the maximum allowed finish
          time for all tasks */
6  end
7  tryReduceScheduleMakespan(jobs);
```

C. Data Placement

The proposed scheduler in the previous module allows to determine the starting time of a task, but most importantly in which node it is executed. As presented in Section.II, the closer the chunk is to the task the lower is the execution time.

To that end, this module uses the task scheduling generated by the Holistic scheduler as a placement strategy to distribute data chunks across nodes. For example, each chunk a is processed by task i. If the scheduler assigned task i to the node k, then before the execution of the task, the data placement strategy sends the chunk a to the node k. This means, if there are no node failures, the executed task is local. Otherwise, the task has to be rescheduled, and the chunk has to be requested and placed on another node. As a consequence, some of the tasks might be executed as remote.

That being said, we do not consider data replication in this strategy. The replication has two main goals, that is, increasing the locality of data and fault tolerance. Our proposed strategy improves data locality. However, for fault tolerance, replication will be considered in a future work.

D. QoS Controller

The QoS controller is a reactive mechanism that minimizes interference between customers (regular and ephemeral) in case of unpredicted workload variations. This happens when the regular customers' workloads are using more than a predefined threshold of the safety margin (set to 50%).

Contrary to Cuckoo, the designed QoS controller considers the two types of resources namely, compressible (*e.g.,* CPU) and incompressible (*e.g.,* RAM). In the upgraded version, the compressible resource, that is the CPU, is throttled benignly. In this case, MapReduce tasks execution is slowed down. In case of incompressible resources, that is the RAM, MapReduce tasks are killed in case of violations (like in Cuckoo).

This difference between compressible and incompressible resources is particularly relevant when using the Forecasting builder [2]. Indeed, CPU predictions generate more errors than memory's. Thus, as compared to Cuckoo, much less tasks are killed because of CPU mispredictions.

Algorithm.3 gives the pseudo-code of the QoS controller for a given node (see Fig.2) which periodically (*e.g.,* every 100 ms) controls tasks' resource utilization. First, we get both the current available resources (resource limits for CPU and RAM) that can be used by the tasks (lines 1-2), and the current tasks' usage (lines 3-4). If tasks are using more than the CPU limit (when co-located workload's CPU usage increases), tasks are throttled (line 5). Otherwise, we increase the CPU limit of tasks (line 6). After that, we check whether the tasks are using more than the 50% safety margin threshold of RAM. In this case, tasks are killed proportionally to their address space size (*i.e.,* RAM usage) in the reverse order of scheduling (*i.e.,* latest first, so to avoid killing tasks that may almost finish their execution) until we release the safety margin (lines 7-12). If the available RAM space is lower than the safety margin, then all tasks should be killed (line 14).

Algorithm 3: QoS controller for a given node

```
   // get resources limits
1  cpuLimit = getCpuLimit();
2  ramLimit = getRamLimit();
   // get tasks' resources usage
3  ramUsage = getTasksRamUsage();
4  cpuUsage = getTasksCpuUsage();
5  if cpuUsage > cpuLimit then throttleTasksCpu(cpuLimit);
6  else increaseTasksCpuLimit(cpuLimit) ;
   // sm refers to the safety margin
7  if ramLimit ≥ sm then
8      smThreshold = sm * 50%;
9      while ramUsage > ramLimit + smThreshold do
10         task = killLastScheduledTask();
11         ramUsage -= task.getRamUsage();
12     end
13 else
14     killAllTasks();
15 end
```

E. Implementation

The solving strategies were implemented with the following frameworks:

- Constraint programming: we used IBM ILOG CP Optimizer [17], a CP-based system to model and solve optimization problems. It provides an algebraic language with mathematical concepts to model the temporal dimension of a combinatorial optimization problem.
- Genetic algorithm: a framework based on JMetal was used to implement the algorithm. JMetal stands for Metaheuristic Algorithms in Java [23]. It is an object-oriented Java framework allowing multi-objective optimization with metaheuristics. It offers many algorithms such as NSGA-III [20] (used in our approach) with the HyperVolume (HV) indicator to compare candidate solutions.
  The parameters set are as follows: population size = 50, mutation rate = 1, crossover rate = 0.5, tournament size = 2.
- Local Search-based algorithm: we used OptaPlanner framework, a constraint solver [24]. It is an object-oriented Java framework that offers several optimization algorithms such as Tabu Search and Late Acceptance used in our approach.

As for the parameters, the framework default values for the selected algorithms were used: entityTabuRatio = 0.2, lateAcceptanceSize = 500.

IV. Experimental validation

In this section, we detail the experimental setup and results used to validate the efficiency of our contribution and try to answer the following Research Questions (RQ):

RQ1: What is the overall performance of the proposed solving strategies compared to Cuckoo in terms of job execution time, relaunched and remote tasks?

RQ2: How do the solving strategies compare with each other in terms of solving time and quality of scheduling?

RQ3: How effective is the proposed QoS controller?

The following details the performed experiments and the evaluation metrics in order to answer the research questions.

Experiment for RQ1: the goal of this experiment is to evaluate the quality of Salamander with the solving strategies (*i.e.,* CP, GA, LS) by comparing it to Cuckoo. The evaluation is done according to three metrics:

1) Job execution time: it represents the time taken by jobs to finish their execution. In this metric, the cost of data transfers (*i.e.,* data placement, remote tasks' chunks, intermediate data of Map tasks) is taken into consideration.
2) Percentage of remote tasks: it represents the tasks which are executed on a different node where the chunk to process was initially placed.
3) Percentage of relaunched tasks: it represents the percentage of killed and rescheduled tasks due to violation.

Only for this experiment, the solving time of each strategy (*i.e.,* CP, GA, LS) was limited to 30 seconds and 2 CPU cores (limitation set in the used frameworks, see Section III-E). We evaluated that this was sufficient for CP to generate the best scheduling while GA and LS could generate a comparable scheduling in terms of quality.

Experiment for RQ2: in this experiment, we want to evaluate the scalability of each strategy to be able to choose the appropriate one. We evaluated the following metrics:

1) Solving time: it represents the time required by the solvers to generate a scheduling.
2) Quality of scheduling: that is the overall jobs' makespan.

Experiment for RQ3: this experiment is done to evaluate how effective is to consider the compressible and incompressible resources for minimizing interference between workloads using the QoS controller. The comparison is performed between Cuckoo with the original QoS controller and Cuckoo with the new QoS Controller. The evaluation metrics are 1) Job execution time, and 2) Percentage of relaunched tasks. We do not consider remote tasks in this case since we are only evaluating the QoS controller not the data placement strategy.

Salamander is not used in this experiment because Cuckoo's QoS controller performs poorly with it. Indeed, because of the high error rate of CPU predictions of the forecasting builder, Cuckoo's controller kills too many tasks causing the Salamander's generated scheduling to be rarely respected.

A. Experimental setup

We used a 70-days production dataset from three different data centers [2]. Table.II shows the overall capacity of all data centers which are heterogeneous. PC-1 (*i.e.,* Private Company 1) has 6 different configurations among its 9 hosts, PC-2 has 13 different configurations among its 27 hosts and University has 6 different configurations. All hosts were configured in the simulator with dedicated links of 50 Mbps and 10 us latency.

TABLE II: Total capacities 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	6	1363	1.5

In our evaluations, for *RQ1* and *RQ3*, we used one configuration of tasks, that is, 640 Map tasks and 40 Reduce tasks. We used a small number of tasks so that the strategies can generate a comparable scheduling within the limited solving time, and to be able to compare the results to Cuckoo. In order to increase the risk of being impacted by the prediction errors, we varied the execution time of tasks by using different chunk sizes. We used two configurations namely 128 MB and 256 MB, corresponding to two datasets of 80 and 160 GB respectively. Finally, to investigate the impact of mispredictions and thus potential SLA violations, we evaluated different safety margins of 0%, 5%, 10%, 15%, 20%, 25%, 30% (the higher the safety margin, the lower the impact of mispredictions).

As for *RQ2*, we used multiple configurations of Map and Reduce tasks, ranging from 500 to 2500 tasks to evaluate the scalability of each solving strategy.

The processing cost of tasks is similar to Cuckoo, each Map and Reduce task is equal to 3100 and 6300 FLOPS/Byte. We set each TaskTracker to run 20 slots of Map and Reduce tasks. As for memory, according to [25], each task needs between 2 and 4 GB of RAM, thus, we set tasks to use 3 and 4 GB with chunk sizes of 128 and 256 MB respectively.

Our experiments were performed using Simgrid 3.20 simulation tool and Cuckoo's version of MRA++ MapReduce [26] for handling the experimentation phases.

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). Then, the deployment phase consists in launching the modules of Salamander and Cuckoo. Finally, the injection phase consists in increasing and decreasing the load of resources over time (3 minutes sampling period) according to data centers utilization. We set the Forecasting builder to use 99*th* quantile level for future resource predictions for all the experiments.

B. Experiment results

1) RQ1-Overall performance, Salamander Vs Cuckoo:

Job execution time: Fig.3 shows the job execution time with 128 MB and 256 MB chunk sizes. A first observation

TABLE III: Percentage of relaunched tasks - University 256 MB chunk size, 5% safety margin

Strategy	Min	Max	Median	98*th* percentile
Cuckoo	58.67	126.91	89.7	126.38
Salamander-CP	0	8.53	0	0.18
Salamander-GA	0	6.03	0	0.36
Salamander-LS	0	7.79	0	0.36

we can draw is that the three strategies of Salamander behave similarly with a slight difference of less than 1% caused by mispredictions. Indeed, the solving strategies generated comparable scheduling of tasks.

When comparing the minimum median job execution time, the three solving strategies outperformed Cuckoo with faster execution time of 9.3%, 11.1% and 8.8% corresponding respectively to PC-1, PC-2 and University with 128 MB chunk size. With 256 MB chunk size, Salamander improved the median job execution time by 30.5% for PC-1, 5.2% for PC-2 and 47.6% for University. We notice that the lower the number of available nodes, the higher the improvements, contrary to 128 MB chunks. This can be explained by the overall execution time of tasks. In fact, the longer the execution time, the more tasks are prone to prediction errors. This is the case when increasing the chunk size or reducing the number of available nodes (*e.g.,* University). This means that Cuckoo is more impacted by mispredictions compared to Salamander (more details in **Relaunched tasks** paragraph).

When increasing the safety margin to 25% and 30% for the University data center, we can observe that Cuckoo performs better than Salamander. This is solely due to the overestimation of RAM usage by the Forecasting builder. For instance, if we consider the RAM in a node with $prediction = 80\%, current\ load = 60\%, safety\ margin = 30\%$, this gives us $prediction + safety\ margin > 100$ which means that Salamander cannot schedule tasks on this node according to this prediction. However, Cuckoo does not consider RAM but only CPU predictions for data placement thus, it can schedule tasks on the node since $current\ load + safety\ margin < 100\%$. That said, Cuckoo performs better at the expense of a higher risk of relaunching tasks and potentially violating the SLA which we want to avoid. Furthermore, depending on the error rate of predictions, we might not have to use a safety margin with more that 10% as shown next.

Relaunched tasks: Table.III shows the percentage of relaunched tasks with minimum, maximum, median and 98*th* percentile values. We show only the results of University with 256 MB chunks and 5% safety margin as it is the worst case for Salamander. We observed that Salamander reduces considerably the percentage of relaunched tasks compared to Cuckoo. The high percentage observed for Cuckoo is due to the fixed number of slots for Map and Reduce tasks that could be executed at the same time. In contrast, Salamander's task scheduling dynamically resizes the number of slots according to the predictions which helps avoid SLA violations.

Remote tasks: Table.IV represents the percentage of remote tasks execution for Salamander and Cuckoo. We observe that Salamander does not execute remote tasks while

TABLE IV: Percentage of remote tasks for Salamander and Cuckoo. (*) refers to CP, GA and LS

Strategy	chunk size (MB)	PC-1	PC-2	University
Cuckoo	128	7.3	6.04	8.3
	256	10.6	8.27	13.65
Salamander-*	128/256	0	0	0

Cuckoo executes about 10% of tasks remotely. Such results are obtained since Salamander's data placement strategy places chunks exactly where tasks are about to be executed which reduces data transfers between nodes. We notice that Cuckoo executes more tasks remotely for the University data center which has the least amount of resources. In fact, Cuckoo uses a replication factor of three to increase the availability of data in case of failure. This means that University should have the lowest percentage of remote tasks. However, this is not the case since University has the highest percentage of relaunched tasks which explains the results. Indeed, in this case, there is a clear correlation between relaunched and remote tasks.

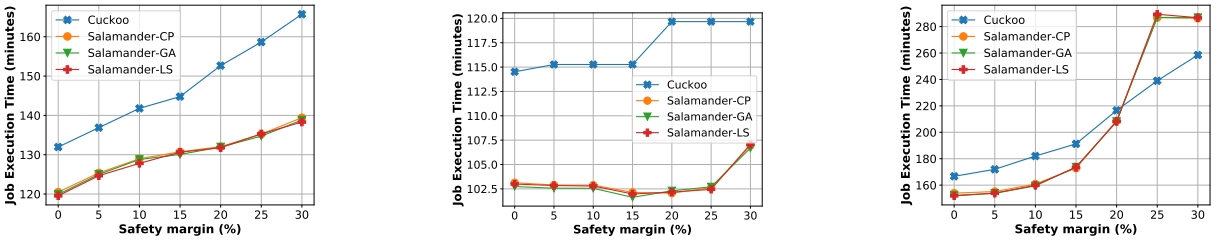
Through this experiment, Salamander improved the job execution time by up to 47.6% and an average of 18.7% compared to Cuckoo. It also reduced considerably both relaunched and remote tasks (as far as node failure does not occur).

2) RQ2-properties of the proposed strategies: In this experiment, we compared the scalability and quality of Salamander's solving strategies (*i.e.,* CP, GA, LS). Fig.4a depicts the results of the solving time of the three strategies for PC-1 (results proved to be similar for PC-2 and University). Fig.4b represents the scheduling quality of the strategies.

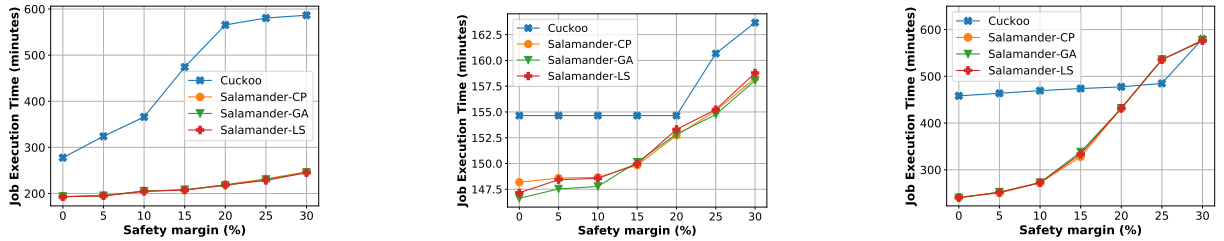
In Fig.4a, we observed that both GA and LS have almost similar solving time which is linear. However, the solving time of CP is exponential with a difference of up to 6x for 2500 tasks compared to GA and LS. Furthermore, we observe in Fig.4b that while CP took much longer time in solving, it only gave a better scheduling quality of around 4.34% to 10.52% compared to GA and LS. This means that CP scales up poorly. As for RAM usage, CP used up to 2.1 GB, GA up to 1.8 GB while LS used up to 1.3 GB.

Through this experiment, we observed that CP gave an optimal scheduling of tasks in a reasonable time for problems with up to 1500 tasks. However, for GA and LS, the scheduling quality was not far apart from CP with almost linear solving time with regards to the problem size. So, in order to choose the appropriate strategy, one must consider the problem size, the amount of resources and most importantly the solving time (*i.e.,* online/offline execution) available for the solvers. For our case, CP can be used for problems with less than 1500 tasks since it gave the optimal scheduling in a reasonable time. However, for a large number of tasks, one may use GA or LS since they have similar quality. That being said, LS can be used in case memory footprint is an issue.

3) RQ3-Effectiveness of the new Qos Controller: In this experiment, we evaluated the efficiency of the new QoS controller compared to the previous version, both using Cuckoo. Fig.5 represents the comparison results (*i.e.,* job execution time, relaunched tasks) for PC-1 with 128 MB chunk size

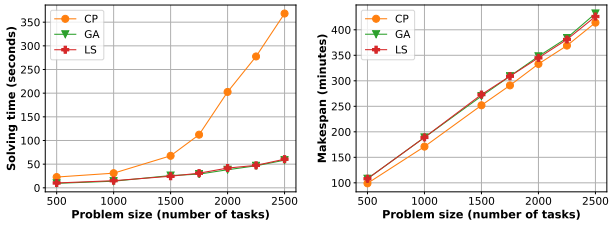


(a) PC-1 - 128 MB (b) PC-2 - 128 MB (c) University - 128 MB



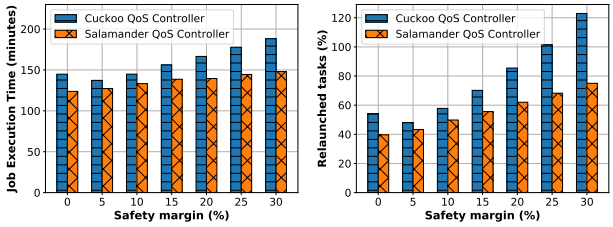
(d) PC-1 - 256 MB (e) PC-2 - 256 MB (f) University - 256 MB

Fig. 3: Job execution time for Salamander (with CP, GA, LS) and Cuckoo - 128 and 256 MB chunk sizes



(a) Solving time (b) Scheduling quality

Fig. 4: Comparison of the solving strategies - CP, GA, LS



(a) Job Execution Time (b) Relaunched Tasks

Fig. 5: QoS controller comparison in PC-1

which is the default setting in HDFS. We only show one dataset results because the rest are similar.

We observe in Fig.5a that Salamander's QoS controller outperforms Cuckoo's. Median value improvements are between 17.3% and 38.9% for relaunched tasks. This results in faster execution times of 9.8% up to 21.3%.

Through this experiment, we showed that considering compressible and incompressible resources in the QoS controller does reduce the number of relaunched tasks which increases the performance in terms of job execution time.

V. Limitations

Here are some potential limitations for our work:

- When the forecasting builder overestimates the future resource utilization, the proposed scheduler might not be able to use some nodes which are in reality available as seen in the experiments. This might be resolved by either having a dynamic safety margin adjusted to the prediction errors, or by lowering the quantile level of predictions.
- When using the predictions to generate a scheduling, we are unable to perfectly execute it because of prediction errors. In our solution, we only shift the plan when relaunching tasks. We believe that this can be improved using other strategies in order to adapt the execution plan, for instance by partially re-evaluating the scheduling.
- In Salamander, we considered a fixed capacity for the resources. However, the capacity can be reduced due to interference (*e.g.,* I/O [27]) between co-located workloads.
- We did not consider the different storage types such as SSD and HDD for optimizing performance and cost [28], [29].

VI. Related Work

We can classify state-of-the-art studies in two categories, predictive and reactive approaches.

Reactive approaches: Pado [11] relies on reserved nodes to save intermediate data and to run particular tasks that would cause high recomputation costs if evicted. However, in our case, one does not necessarily have reserved nodes. Contrary to Pado, Scavenger [30] relies only on unused resources to execute Spark jobs while reducing interference (*e.g.,* LLC cache) with the co-located VM. It reacts to VM's load changes by dynamically adjusting the available resources. Scavenger is easy to deploy but its objective is more about minimizing interference than maximizing the jobs on unused resources. Both Pado and Scavenger do not consider task scheduling taking into account the volatility and heterogeneity of resources.

Predictive approaches: TR-Spark [10] uses reserved resources to checkpoint intermediate results (for jobs ran on ephemeral resources) to avoid unnecessary recomputations. It uses VM's properties to determine when to checkpoint

data. Yet again, reserved resources are not always available. Moreover, the checkpointing rate, even if reduced, can be costly in terms of resources. Harvest [5], on the other hand, focuses on maximizing data centers utilization while minimizing recomputations. It uses services historical information to predict the availability of resources. The solution clusters resources according to CPU utilization patterns into classes. The computations are then assigned to those classes according to their estimated execution time. The main drawback of this method is that resources are clustered only according to CPU and no other metrics were considered. Similarity to Pado and TR-Spark, these resource classes do not account for the heterogeneity of resources.

VII. Conclusion and future work

Executing big data applications such as Hadoop on reclaimed unused resources raises several challenges related to the heterogeneity and volatility of resources as well as interference between co-located workloads of different customers.

To that end, we proposed Salamander, a heterogeneity and volatility aware framework that tackles the aforementioned challenges for efficient execution of Hadoop jobs. It provides a Holistic task and job scheduler with three different solving strategies that rely on future resource predictions. In addition, a scheduler-based data placement strategy is used to improve the locality of data. Finally, a reactive QoS controller, considering compressible and incompressible resources, was proposed. The integration of Salamander required merely 500 lines of code.

The results of our simulation show that Salamander was able to execute Hadoop jobs efficiently on ephemeral resources. It did so by better using volatile resources and avoiding relaunched and remote tasks. This contributes in minimizing interference with co-located customers' workloads.

As a perspective, we will integrate Hadoop's fair scheduler on top of Salamander to provide fairness between clients. We will consider data replication for fault tolerance. In addition, we will consider I/O and network as well as Salamander's limitations for deployment and testing on a real system.

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

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

[3] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGARCH Computer Architecture News*, 2014, pp. 127–144.

[4] J.-E. Dartois, H. B. Ribeiro, J. Boukhobza, and O. Barais, "Cuckoo: Opportunistic mapreduce on ephemeral and heterogeneous cloud resources," in *IEEE 12th International Conference on Cloud Computing*, 2019, pp. 396–403.

[5] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, Í. Goiri, and R. Bianchini, "History-based harvesting of spare cycles and storage in large-scale datacenters," in *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 755–770.

[6] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, p. 7.

[7] K. Chen, J. Powers, S. Guo, and F. Tian, "Cresp: Towards optimal resource provisioning for mapreduce computing in public clouds," in *IEEE Transactions on Parallel and Distributed Systems*, 2014, pp. 1403–1412.

[8] K. Shvachko, H. Kuang, S. Radia, R. Chansler *et al.*, "The hadoop distributed file system." in *IEEE 26th symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–10.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." in *HotCloud*, 2010, p. 95.

[10] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "Tr-spark: Transient computing for big data analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 484–496.

[11] Y. Yang, G.-W. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B.-G. Chun, "Pado: A data processing engine for harnessing transient resources in datacenters," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 575–588.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Communications of the ACM*, 2008, pp. 107–113.

[13] "Apache hadoop 3.0.0, hdfs architecture," 2019. [Online]. Available: http://bit.ly/2wj8FaE

[14] H. Zhang, H. Xu, and W. Peng, "A genetic algorithm for solving rcpsp," in *2008 international symposium on computer science and computational technology*, 2008, pp. 246–249.

[15] C.-W. Tsai and J. J. Rodrigues, "Metaheuristic scheduling for cloud: A survey," in *IEEE Systems Journal*, 2013, pp. 279–291.

[16] R. Barták, M. A. Salido, and F. Rossi, "New trends in constraint satisfaction, planning, and scheduling: a survey," in *The Knowledge Engineering Review*, 2010, pp. 249–279.

[17] "Cp optimizer," IBM ILOG, 2019. [Online]. Available: https://ibm.co/32JaJ83

[18] F. Glover and M. Laguna, "Tabu search," in *Handbook of combinatorial optimization*, 1998, pp. 2093–2229.

[19] E. K. Burke and Y. Bykov, "The late acceptance hill-climbing heuristic," in *European Journal of Operational Research*, 2017, pp. 70–78.

[20] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints," in *IEEE Transactions on Evolutionary Computation*, 2013, pp. 577–601.

[21] B. L. Miller, D. E. Goldberg *et al.*, "Genetic algorithms, tournament selection, and the effects of noise," in *Complex systems*, 1995, pp. 193–212.

[22] P. Moscato *et al.*, "On genetic crossover operators for relative order preservation," in *C3P Report*, 1989.

[23] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," in *Advances in Engineering Software*, 2011, pp. 760–771.

[24] G. De Smet and open source contributors, *OptaPlanner User Guide*, Red Hat, Inc. or third-party contributors, 2006. [Online]. Available: https://www.optaplanner.org

[25] E. Sammer, *Hadoop operations*. " O'Reilly Media, Inc.", 2012.

[26] 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," in *Future Generation Computer Systems*, 2015, pp. 22–35.

[27] J.-E. Dartois, J. Boukhobza, A. Knefati, and O. Barais, "Investigating machine learning algorithms for modeling ssd i/o performance for container-based virtualization," in *IEEE transactions on cloud computing*, 2019.

[28] D. Boukhelef, K. Boukhalfa, J. Boukhobza, H. Ouarnoughi, and L. Lemarchand, "Cops: Cost based object placement strategies on hybrid storage system for dbaas cloud," in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2017, pp. 659–664.

[29] D. Boukhelef, J. Boukhobza, K. Boukhalfa, H. Ouarnoughi, and L. Lemarchand, "Optimizing the cost of dbaas object placement in hybrid storage systems," in *Future Generation Computer Systems*, 2019, pp. 176–187.

[30] S. A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi, "Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 272–285.