RESEARCH-ARTICLE

Symbolic execution-driven extraction of the parallel execution plans of Spark applications

LUCIANO BARESI, Politecnico di Milano, Milan, MI, Italy

software engineering, cloud computing, edge computing, self-adaptive systems, mobile apps.

GIOVANNI DENARO, University of Milan, Milan, MI, Italy

GIOVANNI QUATTROCCHI, Politecnico di Milano, Milan, MI, Italy

Published: 12 August 2019

ESEC/FSE '19: 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
August 26 - 30, 2019
Tallinn, Estonia

Conference Sponsors:
SIGSOFT

ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (August 2019)
https://doi.org/10.1145/3338906.3338973
ISBN: 9781450355728

Symbolic Execution-Driven Extraction of the Parallel Execution Plans of Spark Applications

Luciano Baresi
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy
luciano.baresi@polimi.it

Giovanni Denaro
Dipartimento di Informatica, Sistemistica e Comunicazione, Università di Milano Bicocca, Italy
denaro@disco.unimib.it

Giovanni Quattrocchi
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy
giovanni.quattrocchi@polimi.it

ABSTRACT

The execution of Spark applications is based on the execution order and parallelism of the different jobs, given data and available resources. Spark reifies these dependencies in a graph that we refer to as the (parallel) execution plan of the application. All the approaches that have studied the estimation of the execution times and the dynamic provisioning of resources for this kind of applications have always assumed that the execution plan is unique, given the computing resources at hand. This assumption is at least simplistic for applications that include conditional branches or loops and limits the precision of the prediction techniques.

This paper introduces *SEEPEP*, a novel technique based on symbolic execution and search-based test generation, that: i) automatically extracts the possible execution plans of a Spark application, along with dedicated launchers with properly synthesized data that can be used for profiling, and ii) tunes the allocation of resources at runtime based on the knowledge of the execution plans for which the path conditions hold. The assessment we carried out shows that *SEEPEP* can effectively complement *dynaSpark*, an extension of Spark with dynamic resource provisioning capabilities, to help predict the execution duration and the allocation of resources.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; • **Software and its engineering** → *Massively parallel systems*; *Search-based software engineering*.

KEYWORDS

Apache Spark, Symbolic Execution, Search-based Test Generation, Elastic Resource Provisioning

ACM Reference Format:
Luciano Baresi, Giovanni Denaro, and Giovanni Quattrocchi. 2019. Symbolic Execution-Driven Extraction of the Parallel Execution Plans of Spark Applications. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338973

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338973

1 INTRODUCTION

Big-data applications —executed onto large clusters of physical/virtual machines— are increasingly becoming central in industry and academia [13]. Special-purpose frameworks divide the computation into different phases and split the input dataset into partitions that are stored in a distributed way and processed in parallel.

Apache Spark [35] (hereafter *Spark*), probably the de-facto standard for batch computations, exploits in-memory processing and storage as means to reuse partial results. Spark applications exploit two types of dedicated operations: *action*s and *transformation*s. Actions trigger (distributed) computations that return results to applications. Transformations carry out data transformations in parallel. Spark groups operations into *stage*s and then into *job*s. A stage comprises a sequence of transformations that do not require data shuffling, while a job identifies a sequence of transformations between two actions. For each job, Spark computes a *Parallel Execution Plan* (*PEP*) to maximize the parallelism while executing an application. In fact, a stage is, by definition, executed in parallel but different stages can also be executed concurrently. For this reason, Spark materializes *PEPs* as direct acyclic graphs of stages[1] while the complete *PEP* of an application is simply the sequence of the *PEPs* of its jobs.

The growing importance of big-data applications has motivated diverse analysis techniques to estimate the execution time of Spark applications and to allocate resources properly. For example, Islam et al. [18] propose a solution to statically allocate resources to deadline-constrained Spark applications while minimizing execution costs. Sidhanta et al. [29] estimate the duration of Spark applications using a closed-form model based on the size of the input dataset and the size of available cluster. Alipourfard et. al [3] use Bayesian optimization to generate performance models of Spark applications and compute the best configuration for their execution. Baresi et al. [5, 6] propose *dynaSpark*, an extension of Spark that exploits control theory and containers[2] to scale allocated resources elastically given the execution times of interest and the other applications running on the same cluster.

All these approaches implicitly assume that the *PEP* of an application is unique for any possible input dataset and input parameters, but this is at least simplistic for all but the most trivial applications. In general, application code embeds branches and loops, and different input data and parameter values may make the application traverse different program paths, and thus materialize a different

[1] This is not the control flow graph of the application, but the specific execution flow of the tasks that can be executed in parallel. It is a graph and not a sequence because of the parallelism.
[2] https://www.docker.com.

PEP. For example, a Spark application can evaluate partial results through actions, and then use these results in conditional expressions of program branches.

This paper proposes the use of an original combination of lightweight symbolic execution and search-based test generation to properly infer the actual *PEP*, given used data and parameters. Our approach is called *SEEPEP* (*Symbolic Execution-driven Extraction of Parallel Execution Plans*) and relies on symbolic execution of the application's code to derive a representative set of execution (path) conditions of the *PEPs* in the application. It then uses these conditions with a search-based test generation algorithm to compute sample input datasets to execute each *PEP*. These inputs allow us to profile the application with respect to when it uses each specific *PEP*. This way, *SEEPEP* identifies what we call the *PEP** of applications, that is, the set of all possible *PEPs*, along with their respective path conditions and profiling data. The computed *PEP** feeds $dynaSpark_{SEEPEP}$, a custom version of *dynaSpark*, to evaluate all feasible *PEPs* against the concrete values of symbolic variables and identify those that remain feasible —i.e., their path conditions continue to hold. $dynaSpark_{SEEPEP}$ then tunes allocated resources on the worst feasible *PEP*.

Our preliminary evaluation shows that i) *SEEPEP* efficiently extracts all the possible *PEPs*, that is, all the possible different execution of an application, and ii) *SEEPEP* helps $dynaSpark_{SEEPEP}$ improve *dynaSpark* and allocates resources more efficiently.

The rest of the paper is organized as follows. Section 2 recalls how Spark works, briefly introduces *dynaSpark*, and exemplifies the problem addressed in the paper. Section 3 explains our symbolic execution of Spark applications and Section 4 how *dynaSpark* uses it. Section 5 presents the experiments we carried out. Section 6 surveys related approaches and Section 7 concludes the paper.

2 *DYNASPARK*

Spark deploys a master/worker architecture on a cluster of machines. A Spark application (*driver program*) starts by creating a *context* that interacts with the *master node*, that is, the manager of the actual computing resources, called *worker node*s. Each worker node is installed on a dedicated machine and contains one or more *executor*s that run for the entire lifetime of the application.

Each executor performs multiple *task*s in parallel using a pool of threads, and the number of parallelized tasks depends on the number of CPU cores it has been provisioned with. Tasks can persist the results of their computations on a distributed storage layer (e.g., HDFS [28]), hosted on the same cluster, or on dedicated machines. When multiple applications are executed on the same cluster, each application has its own context, master and worker nodes are shared, while executors must be assigned to the different applications before starting their execution.

Spark operations manipulate *RDD*s (Resilient Distributed Datasets). An RDD is an immutable and fault-tolerant collection of *record*s that is split into multiple redundant *partition*s to facilitate parallel computations. Transformations create new RDDs, while actions generate values. Spark starts executing a program by identifying its *job*s, that is, sequences of transformations delimited by the presence of actions in the code, and *stage*s (within jobs), delimited by transformations that require data to be shuffled (i.e., moved among executors). These transformations are called *wide*, while the others are called *narrow*.

For each job, Spark identifies the operations that are to be executed, given the specific input data and parameter values, and creates a *PEP* (Parallel Execution Plan). *PEPs* do not contain branches and loops: they have already been unrolled and each *PEP* only considers a specific execution. A *PEP* defines the execution order among stages and the extent to which they can be executed in parallel. Note that a task performs all the operations of a particular stage on a partition of the input RDD of that stage. These tasks are executed in parallel, depending on the number of available executors and the number of CPU cores provisioned to them. The different jobs of an applications are then executed following a LIFO approach.

dynaSpark[3] extends Spark and uses containers [33] (i.e., Docker) to support a more flexible and advanced management of CPU cores and meet set execution times (deadlines). Spark requires that resources be provisioned statically: if the execution completes faster than expected, fewer resources could have been provisioned, but if it takes longer, more resources are needed. *dynaSpark* moves the allocation problem to run-time and deals with these cases.

dynaSpark augements the *Master Node* with a *Stage Scheduler*, a *Task Scheduler*, and a heuristic-based *Application Level Controller* for each running application. Stages are key for *dynaSpark*: the *Stage Scheduler* intercepts the beginning and end of each stage and uses the heuristic embedded in *Application Level Controller* to compute an execution *deadline* for each stage. The heuristic considers the remaining amount of time, with respect to the global deadline set for the whole application, and some performance data collected through a profiling phase. This profiling is also in charge of providing the *PEP*: as said before, *dynaSpark* assumes that the *PEP* does not change with respect to input data, and this is the key reason that has motivated this work.

dynaSpark constrains executors to specific stages, while Spark would allow executors to execute the tasks of different stages. This way, the resources (dynamically) provisioned to a given executor can only impact the performance of the stage associated with it, and *dynaSpark* can obtain a finer-grained control of the execution of the different stages, and thus of the whole application. *dynaSpark* controls stages individually and distributes computation and data over the whole set of nodes equally.

Executors are wrapped in containers and individually controlled by a control-theoretical planner (*Container Level Controller*). These planners exploit a feedback loop that monitors the progress of the executors and allocates computing resources to match estimated deadlines. Resource contention on *Worker Node*s could occur because executors bound to different applications/stages are deployed onto it. This is why *dynaSpark* uses a *Worker Level Controller* to gather all the requests for cores from the control-theoretical planners and, if their sum is greater than those available, scales them down according to different configurable strategies.

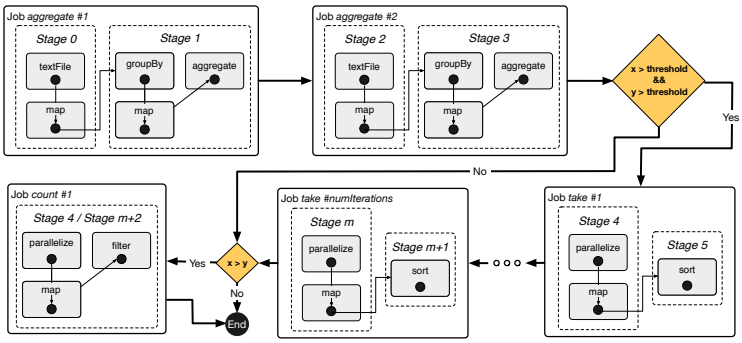
2.1 Example Applications

Most of the approaches in the literature (e.g., [17, 18, 21, 29]) uses the execution graph (i.e., the *PEP*) to reason on the work to do, the

[3] https://github.com/deib-polimi/dynaSpark

```
1  from pyspark import SparkContext
2  def run(numIterations, threshold):
3    sc = SparkContext('local','example')
4    x = sc.textFile(...).map(...).groupBy(...)
5     .map(...).aggregate(...)
6    y = sc.textFile(...).map(...).groupBy(...)
7     .map(...).aggregate(...)
8    if x > threshold and y > threshold:
9      for i in range(numIterations):
10      z = sc.parallelize(...).map(...).sort(...).take(10)
11   if x > y:
12     w = sc.parallelize(...).map(...).filter(...).count()
```

(a) Source code



(b) The four *PEPs* of the application

Figure 1: Example Spark application with conditional branches and loops.

degree of parallelism, the duration of tasks, and other application-specific characteristics. They also assume that the graph does not change since many of the conditional branches and loops are mimicked in the code (e.g., filter, map). As said, this is wrong when the code contains explicit loop and conditional statements.

For example, one can think of a simple two-job application. The first job retrieves some records from a data source (e.g., a file) and filters then according to a given criterion; the second job sorts them and returns the first x records. To avoid problems, one may constrain the execution of the second job to the fact that the cardinality c of filtered records, that is, the result produced by the first job, is greater than zero. The *PEP* would then comprise two jobs $if\ c \geq 0$; it would only comprise the first job otherwise. This example shows how Spark can return partial results (c) through actions to the driver program and use them to evaluate conditional (loop) expressions, and thus produce different *PEPs*.

To overcome this problem, *dynaSpark* and many solutions [18, 29] exploit an initial profiling phase to retrieve the *PEP* and collect some performance metrics. Back to the simple example above, the initial profiling would return the *PEP* implied by the data used to run the application. This single choice impacts the quality of obtained results and there is currently no means to adjust the graph with respect to the different data. Even if one adopts a conservative approach and retrieves the *PEP* that corresponds to the worst case (i.e., two jobs in the previous example), this would result in over-allocating resources and/or over-estimating execution times. If one adopted the best case (one job), too few resources and too short execution times would be foreseen.

The *PEP* can also depend on user parameters or local variables and they must be considered in a sound analysis. For example, Figure 1a shows the code of an example application that takes two input parameters *numIterations* and *threshold*; its *PEP* depends on both these user parameters and input dataset. The first two *aggregate* jobs are always executed (line 4 and 6) and the results are assigned to variables x and y, respectively. Line 8 checks if both variables are greater than *threshold*. If it is the case, a *take* job (line 10) is repeated *numIterations* times (`for` loop). Finally, if $x > y$ (line 11) *count* (line 12) is executed.

This code corresponds to four possible *PEPs* (Figure 1b): i) the sequence of the two *aggregate*s (if both conditional statements are false) ii) the sequence of the two *aggregate*s and *take* repeated *numIterations* times (if the first conditional statement is true and the second is not) iii) the sequence of the two *aggregate*s and *count* (if the second conditional statement is true but not the first), and iv) the concatenation of the two *aggregate*s, *take* repeated *numIterations* times, and *count* (if both conditional statements are true).

3 *SEEPEP*

SEEPEP (Symbolic Execution-driven Extraction of Parallel Execution Plans) is an original combination of lightweight symbolic execution and search-based test generation that allows us to extract the *PEP** of Spark applications. A *PEP** associates each control-flow path of the target application with the *PEP* generated by its execution, the relative profiling data, and the path condition that activates the path.

SEEPEP consists of four main phases: i) it relies on a lightweight symbolic execution of the driver program of the Spark application to derive a representative set of execution conditions of the control-flow paths in the program; ii) it exploits those execution conditions, with a search-based test generation algorithm, to compute sample input datasets that make each path execute; iii) it executes the target application with those datasets as input, to profile the *PEP* generated by each path, and synthesize the *PEP** accordingly; iv) it generates an artifact called *GuardEvaluator* that returns the feasible *PEPs* given a partial set of concrete values of the symbolic variables. We exploit the information in the *PEP** computed with *SEEPEP* to extend *dynaSpark* (see Section 4) with the ability of tuning its adaptation strategy according to the worst-case behaviour of the application. At runtime, our extended version of *dynaSpark* exploits the *GuardEvaluator* to refine the control policy by recomputing the worst-case estimation every time the current worst-case refers to a path for which the execution condition stored in the *PEP** becomes unsatisfiable. Below, we describe each phase of *SEEPEP* in detail.

3.1 Lightweight Symbolic Execution

SEEPEP relies on a lightweight symbolic execution of the driver program of the Spark application to identify the execution conditions of the feasible program paths of the driver program. To this end, *SEEPEP* models with unconstrained symbolic values the results of

$$SE(s \equiv \langle \ell, vv, pc \rangle, t \equiv \langle \ell, op \rightarrow \ell' \rangle) \triangleq \begin{cases} \langle \ell', vv[\texttt{x} \leftarrow [\![\texttt{e}]\!]_{vv})], pc \rangle & \text{if } op \equiv \texttt{x := e} \\ \langle \ell', vv, pc \wedge [\![\texttt{c}]\!]_{vv} \rangle & \text{if } op \equiv \texttt{assume(c)} \\ \langle \ell', vv[x \leftarrow \delta], pc \rangle & \text{if } op \equiv \texttt{x := parallel\_op(...)} \text{ e.g. } \begin{cases} \texttt{sparkCtx.textFile(...)} \\ \texttt{data.map}(\lambda) \\ \texttt{data.filter}(\lambda)) \\ \ldots \end{cases} \\ \langle \ell', vv[x \leftarrow \alpha_\ell], pc \rangle & \text{if } op \equiv \texttt{x := aggregate\_op(...)} \text{ e.g. } \begin{cases} \texttt{data.count()} \\ \texttt{data.collect()} \\ \texttt{data.take(n)} \\ \ldots \end{cases} \end{cases}$$

Figure 2: Symbolic execution algorithm of *SEEPEP*.

the parallel computation jobs issued in the driver program, thus abstracting from the details of those computations, and symbolically analyzes the dependencies of the program paths on these symbolically modeled results. *SEEPEP* leaves to the subsequent test generation phase the burden of identifying concrete input datasets that make the parallel computation jobs encompassed in the driver program yield results that satisfy the path constraints identified during symbolic execution.

This section formalizes the symbolic execution algorithm of *SEEPEP* for a simple imperative programming language in which all operations are either assignments of program variables or assume operations. The assignments are in the form `x := e`, where `x` is a program variable and `e` is an expression of values of program variables. The assume operations are in the form `assume(c)`, where `c` is a condition on the values of program variables, with the semantics that the program continues to execute only if the condition `c` evaluates to *true*. A program in this language defines a transition system with a finite set of program locations $L \triangleq \{\ell_1, \ell_2, \ldots, \ell_n\}$, a specified initial location $\ell_{init} \in L$, and a transition relation $T \triangleq \{t \equiv \langle \ell, op \rightarrow \ell' \rangle\}$ that states the semantics of the program that can move from $\ell \in L$ to $\ell' \in L$ by executing a valid assignment or assume operation *op*.

Two special classes of assignments, that is, assignments of the form `x := parallel_op(...)` and `x := aggregation_op(...)`, respectively, define parallel computations. The assignments `x := parallel_op(...)` assign the variable `x` of the special type *Dataset* to the result of the expression `parallel_op(...)`, which in our context can refer to evaluating any of the parallel computation operations allowed in Spark (e.g., `map`, `filter`, `reduceByKey`). The assignments `x := aggregation_op(...)` assign the variable `x` to the result of a data aggregation operation, e.g., `count`, `collect`, etc, evaluated against a dataset computed in parallel.

Figure 2 defines the symbolic execution algorithm of *SEEPEP*. We denote the symbolic states computed during the analysis with $s \equiv \langle \ell, vv, pc \rangle$, being ℓ the program location to which this symbolic state refers, vv the set of program variables assigned so far, and pc (the path condition) the path constraint due to the assume operations traversed so far. The algorithm starts from the initial state $s\_init \equiv \langle \ell_{init}, \emptyset, true \rangle$ (no variable assigned, unconstrained path), and unfolds the transitions of each program path by recursively executing the atomic step $s' \leftarrow SE(s, t)$ of Figure 2, where s' is the state reached from s when executing transition t.

Figure 2 defines the algorithm as a list of four cases. The first two cases describe the classic symbolic execution algorithm that handles (i) the assignment operations `x := e` by setting variable `x` to the value of expression `e` in the current state ($x \leftarrow [\![e]\!]_{vv}$), and (ii) the assume operations `assume(c)` by conjoining the current path condition with the value of condition `c` in the current state ($pc \wedge [\![\texttt{c}]\!]_{vv}$). The last two cases in Figure 2 define the abstract modeling of the assignments that involve parallel operations: (iii) the assignments `x := parallel_op(...)` result in setting the variable `x` to the unique symbolic value δ, which we use to symbolically model every dataset accessed and computed in the program; (iv) the assignments `x := aggregation_op(...)` result in setting the variable `x` to a new unconstrained symbolic value α_ℓ that models the result of the aggregation operator called at program location ℓ. For the sake of simplicity the figure omits the additional incremental index that we use to symbolically model the results of subsequent assignments at a location that is traversed multiple times in the same program path.

The right part of Figure 2 exemplifies a set of both `parallel_op` and `aggregation_op` operations. These examples include the Spark operations that appear used in this paper. Beyond these examples, with reference to the RDD Programming Guide [4], the `parallel_op` operations of Figure 2 encompass the complete list of *transformation* and *shuffle* operations, while the `aggregation_op` operations encompass all *action* operations.

An important remark about the algorithm is that the conditions of the assume operations defined in the driver program cannot explicitly predicate on the internal state of variables of type *Dataset*. In fact, although the variables of type *Dataset* undergo parallel computations, the data produced with these computations may propagate in the driver program only indirectly, as the result of invoking some `x := aggregation_op(...)`. Thus, the assume operations in the driver program may predicate only on variables assigned as `x := e` and `x := aggregation_op(...)`. This guarantees that the symbolic value δ that models the assignments `x :=`

`parallel_op(...)` never appears in a path condition. This is the reason why we can adopt the simplification of using this single symbolic value to abstractly model all the datasets that the target driver program may manipulate.

SEEPEP uses the algorithm of Figure 2 to symbolically analyze the paths of the target driver program, and returns the path condition computed for each path. As usual in symbolic execution, we use a constraint solver to check if any path condition becomes unsatisfiable at some point of the analysis, and dismiss the analysis of the program paths with unsatisfiable path conditions. Our current *SEEPEP* prototype implements the algorithm described in this section on top of the symbolic executor JBSE [10] that relies on Z3 [14] as the constraint solver.

For example, for the Spark application in Figure 1a, when analyzing the paths of the driver program that do not enter the loop at line 9, (let α_5 and α_7 be the symbols that represent the results of the `aggregate` actions at line 5 and line 7, respectively, and $thresh$ and $iters$ the symbols that represent the input values of parameters `threshold` and `numIterations`, respectively) *SEEPEP* computes the path conditions:

- $\alpha_5 \leq thresh \wedge \alpha_5 > \alpha_7$;
- $\alpha_5 \leq thresh \wedge \alpha_5 \leq \alpha_7$;
- $\alpha_5 > thresh \wedge \alpha_7 \leq thresh \wedge \alpha_5 > \alpha_7$;
- $\alpha_5 > thresh \wedge \alpha_7 > thresh \wedge iters \leq 0 \wedge \alpha_5 > \alpha_7$;
- $\alpha_5 > thresh \wedge \alpha_7 > thresh \wedge iters \leq 0 \wedge \alpha_5 \leq \alpha_7$;

while it identifies the unsatisfiable path condition $\alpha_5 > thresh \wedge \alpha_7 \leq thresh \wedge \alpha_5 \leq \alpha_7$.

For programs with loops, like the one in Figure 1a, *SEEPEP* bounds the iterations of the loops to an user-defined maximum value, thus guaranteeing the analysis of a finite amount of paths.

3.2 Search-Based Test Generation

SEEPEP exploits the path conditions identified with symbolic execution to generate test cases –one test case for each path condition– that comprise the input values and input datasets that make the target Spark application concretely execute the paths of the driver program that correspond to the path conditions. The goal is to use these test cases to profile the behavior of the *PEP* generated by the execution of each path of the driver program.

Note that we cannot rely on the classic approach of solving the path conditions to satisfy assignments by means of an SMT solver. Our path conditions capture partial symbolic representations of the datasets, since they do not characterize the functions passed to Spark transformations. This means that SMT solvers would miss the proper constraints to produce valid datasets.

To generate a test case for a given path condition, *SEEPEP* incrementally explores the space of the possible test cases in a search-based fashion, steering the search with a fitness function that quantifies the extent to which each incrementally considered test case is close to (or far from) satisfying the path condition at hand.

Below, we first describe the *SEEPEP* search algorithm in detail, and then explain the test execution sandbox that the algorithm uses to speed up the execution of the test cases.

3.2.1 Search Algorithm. The *SEEPEP* search algorithm generates test cases that call the target application with the inputs (both the input parameters and the input datasets) assigned to concrete values (both concrete values of the parameters and datasets). The algorithm samples the possible values of the inputs as *genetic algorithms* do. It starts generating a *population* of test cases that comprise randomly-selected inputs, and then *evolves* from the initial population, by incrementally generating a series of next-generation populations, each obtained by manipulating the test cases in the previous population by means of *mutation* and *crossover* operators. Mutation operators generate new test cases by randomly modifying some inputs of a test case of the previous population. Crossover operators generate new test cases as the children of some pairs of the test cases of the previous population, by conjoining inputs taken from either test case of the pair.

The *SEEPEP* fitness function quantifies the goodness of each generated test case with respect to the goal of satisfying a previously identified path condition yielding a value that we interpret as the distance of the current test case from a satisfying test case. If the fitness function yields a distance of 0, the current test case is indeed a satisfying test case, and the search algorithm returns it as result. If the fitness function yields a value greater than zero, the search algorithm exploits it to comparatively order the test cases of the current population. The search algorithm proceeds by probabilistically favouring the application of mutation and crossover operators to test cases with lower distance from the goal, thus increasing the chances of eventually converging to a satisfying test case.

In detail, *SEEPEP* computes the fitness of a test case with respect to a path condition as follows. First, it executes the test case, and collects the results of the Spark aggregation actions that the driver program executes thereby. Next, it evaluates the path condition for the valuation of the symbolic values induced by the execution of the test case, that is, by assigning the symbolic values that model input parameters to the concrete values set in the test case, and the symbolic values that model the results of aggregation actions (the α_ℓ symbols of Figure 2) to the corresponding results collected while executing the test case. If the test case does not execute an aggregation action referred to in the path condition, we assign the corresponding symbol to the special value $Undef$.

Then, if there are no $Undef$ symbols, and if the path condition evaluates to $true$ for the concrete assignment induced by the test case, the fitness is zero: indeed the test case satisfies the path condition. Otherwise, the fitness becomes the positive value computed by the formula reported in the following, where t is the test case, and pc the path condition:

$$fitness(t, pc \equiv c_1 \wedge c_2 \wedge \cdots \wedge c_n) = \sum_{i=1}^{n} distance(t, c_i)$$

where c_i are the atomic conditionals in the path condition pc, and the function $distance$ that appears in the summation recursively computes the distance of each atomic conditional from being satisfied. In turn, the function $distance$ is defined as follows (let $c \equiv o_1 \bowtie o_2$ be a conditional, where $\bowtie$ is a comparison operator and o_1, o_2 are operands, either literals or symbolic expressions):

$$distance(t, c) \triangleq \begin{cases} 0, & \text{if } t(o_1) \bowtie t(o_2) = true \\ 1, & \text{if } t(o_1) = \text{Undef} \ \vee \ t(o_2) = \text{Undef} \\ 1 - \dfrac{1}{1 + |t(o_1) - t(o_2)| + \epsilon}, & \text{otherwise} \end{cases}$$

where $t(o_1)$ and $t(o_2)$ are the values of operands o_1 and o_2, respectively, for the concrete assignments set in the test case t, while ϵ is an arbitrary small number. Note that $t(o_1)$ and $t(o_2)$ are set to $Undef$ if they depend on any symbol assigned to $Undef$ after executing the test case.

Since function *distance* always yields a value in the interval [0, 1], the overall $fitness$ ranges in the interval [0, n] for a path condition with n atomic conditionals. Function *distance* yields zero (first case in the formula) for satisfied conditionals, and thus the overall $fitness$ is zero only for a test case that satisfies all conjuncts, that is, a satisfying test case, as expected. Function *distance* yields the maximum value 1 (second case in the formula) for conjuncts that refer to any symbol assigned to $Undef$, and thus the overall $fitness$ is never zero if it depends on any symbol assigned to $Undef$, as expected. Function *distance* yields values increasingly closer to zero (third case in the formula) if the operands of the referred conditional evaluate to increasingly mutually-closer values, meaning that the corresponding test cases do not satisfy the conditionals for increasingly smaller amounts. Thus, the closer to zero the overall $fitness$ is, the higher the number of satisfied or close-to-be-satisfied conditionals is, as expected.

3.2.2 Test Execution Sandbox. Each fitness evaluation issued in the above search algorithm requires, at least in principle, the execution of the parallel application under test, which can quickly become computationally infeasible given the many test cases that the algorithm generates during the search. To address this issue, the *SEEPEP* search algorithm executes the test cases in a test execution sandbox that specializes the RDD-typed datasets of the target Spark application as a custom type of datasets that we call *sparse diversity data (SDD) datasets*.

An SDD dataset synthetically represents an RDD object with many data points that hold the same value. An SDD dataset is modelled as a list of data blocks, each with two attributes, namely, `size` and `value`: A data block with `size` equal to s and `value` equal to v stands for a set of s data points, all with the same value v. *SEEPEP* uses this format to model datasets in which the amount of distinct values is significantly much smaller than the overall amount of values in the dataset. For example, a dataset with 20^9 data points in which half of the data points have value 100 and the other half -100 can be very concisely represented with an SDD dataset with two data blocks, both with `size` equal to 10^9, and `value` equal to 100 and -100, respectively.

The test execution sandbox recasts the computation of the parallel operations allowed on RDD objects (e.g., operations like `map`, `filter`, etc.) to sequential operations executed on the data blocks in the SDD objects. For example, a `map`(λ) transformation executed on an SDD dataset D with data blocks $[b_1, b_2, ..., b_n]$ yields a new SDD dataset D' with data blocks $[b'_1, b'_2, ..., b'_n]$ such that, for all $i = 1..n$, $b'_i.size := b_i.size$ and $b'_i.value := \lambda(b_i.value)$.

Similarly, a `filter`(λ) transformation on D yields D'' with the subset of the data blocks of D that satisfy condition $\lambda(b_i)$. Yet, a `count()` action on D yields value $\sum^i b_i.size$ as result. Our SDD objects handle all transformations and actions defined in the RDD Programming Guide [4].

The crossover and mutation operators of the *SEEPEP* search algorithm manipulate the input SDD datasets of the target application by modifying, adding, and removing data blocks (mutation operators) or combining the data blocks from the SDD datasets in the parent test cases (crossover operator). Our current *SEEPEP* prototype exploits the SUSHI test generation framework [8, 9] to implement the search algorithm described in this section. SUSHI converts the path conditions generated with JBSE in fitness functions as the ones described in this section, and adapts the test genetic search procedure of EvoSuite [15] to use these fitness functions.

For example, given one of the aforementioned path conditions computed for the Spark application of Figure 1a, *SEEPEP* would compute a test case similar to the following one.

```
Test:
1  threshold = 152;
2  numIterations = 0;
3  D1 = new SDD(size = 1000000, value = 721);
4  D2 = new SDD(size = 3000000, value = 814);
5  setInputTextFile(..., D1);
6  setInputTextFile(..., D2);
7  run(numIterations, threshold);
```

This test sets the input parameters `threshold` and `numIterations` to concrete values (lines 1–2), builds two SDD datasets, both with a single data block (lines 3–4), sets these datasets as the input files the application reads as input (lines 5–6), and executes the application with these inputs.

3.3 Synthesis of the *PEP**

SEEPEP uses the test cases generated with the search algorithm to execute the target Spark application and profile the *PEP* generated by the execution of each path of the driver program. In this phase, *SEEPEP* replaces the SDD datasets that appear in the test cases yielded by the search algorithm with proper RDD datasets that mimic the blocks in the SDD, but comprise millions of replicas of each block. As a result, these RDDs satisfy the same path conditions as the corresponding SDDs, and thus exercise the corresponding *PEPs*, while producing execution times that are representative approximations of real datasets.

SEEPEP executes the test cases against the target application in parallel. While executing each test case, it stores the *PEP* that the Spark engine produces before starting the execution of each job, and monitors the parallel execution of the jobs to collects the timing data that are relevant for the control policy.

SEEPEP builds the *PEP** model as a set of triples

$$\langle pc \rightarrow PEPs, times \rangle$$

where each triple represents the sequence of *PEPs* and the timing data ($times$) associated with the execution of the test case that corresponds to the path condition pc.

Together with the *PEP**, *SEEPEP* produces a *GuardEvaluator* that takes as input a partial set of concrete values of the symbolic variables, evaluates the path conditions of the triples in the *PEP** against these values, identifies which path conditions evaluate to $false$ for these values, and returns the subset of the *PEP** with only the triples with non-falsified path conditions. The control policy, described in the next section, invokes the *GuardEvaluator* at runtime, and feeds it with the concrete values of the input parameters and incrementally with the concrete values of the executed actions, to keep

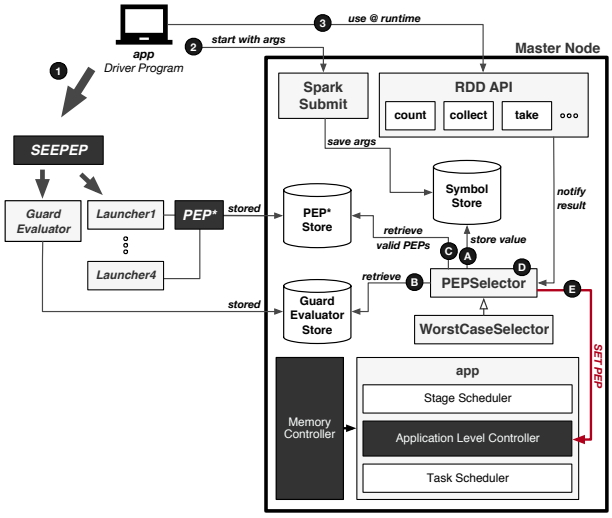


Figure 3: dynaSpark$_{SEEPEP}$

the still-reachable program paths updated at every intermediate execution state.

SEEPEP also addresses the possible incompleteness of either its symbolic execution or search phase. As we already commented above, in the symbolic execution phase, *SEEPEP* analyzes the loops in the program up to a finite (user-configured) amount of iterations, and the analysis may thus produce incomplete results if we dismiss some program path (if any) that iterates a loop more times than that amount. In this case, *SEEPEP* tracks the path conditions $\hat{pc}$ that corresponds to the interrupted prefix of the non-analyzed paths, and stores these path conditions in the *PEP** as special triples with missing data $\langle \hat{pc} \rightarrow \emptyset, - \rangle$. Similarly, if the search algorithm fails to converge to the optimal solution for some path condition $\tilde{pc}$, *SEEPEP* stores the corresponding triples with missing data $\langle \tilde{pc} \rightarrow \emptyset, - \rangle$. These special triples in the *PEP** allow the control policy described in the next section to anticipate when an un-profiled path is going to be executed at runtime, and act to mitigate the impact of these unforeseen situations.

4 DYNASPARK$_{SEEPEP}$

This section describes how *SEEPEP* integrates with *dynaSpark*: the resulting tool-chain is called *dynaSpark*$_{SEEPEP}$.

Figure 3 shows the main elements of the tool-chain and exemplifies it on profiling and controlling the example application of Figure 1a —*app*, hereafter. As first step, *SEEPEP* generates the 4 (n, in general) launchers, which activate the four (n) *PEPs* of the program, and a *GuardEvaluator* for these *PEPs*.

The toolchain associates each *PEP* with its path condition, uses the generated launchers to obtain the profiling data of each *PEP*, and synthesizes the *PEP** for the application. It stores the *PEP** and the *GuardEvaluator* in the master node in components *PEP* Store* and *GuardEvaluator Store*, respectively. The *GuardEvaluator* implements a common interface to be dynamically instantiated and used without importing it statically.

After this phase, *dynaSpark*$_{SEEPEP}$ can execute *app*. During the execution, *dynaSpark*$_{SEEPEP}$ traces the concrete values that correspond to the symbols referred to in the path conditions, that is, the concrete values of the application input parameters and the results of the actions executed at runtime. Specifically, component *SparkSubmit* stores the values of the application parameters in component *SymbolStore*, and our modified version of the *RDD API* notifies the result of any action is executed to component *PEPSelector*, which stores these results in *SymbolStore*.

PEPSelector is in charge of selecting the *PEP* and its profiling data, then used by *Application Level Controller* to compute the local deadlines for the next stages and thus oversee the provisioning of resources. Upon storing new values in the *SymbolStore*, the *PEPSelector* retrieves the *GuardEvaluator*, and feeds it with the *SymbolStore*, to retrieve the list of *PEPs* whose path conditions still hold. This operation has negligible overhead, since it requires only to evaluate the path conditions with symbols assigned to the corresponding concrete values in *SymbolStore*, without checking any RDD.

For instance, at the beginning of the execution of function *run* of *app*, four *PEPs* are valid since neither x nor y have been resolved to a value. The job at line 4 produces the value of x and if the value is less than or equal to $threshold$, the if statement of line 8 is not evaluated. Therefore, even if the value of y is still unknown, *GuardEvaluator* only returns two *PEPs*, that is, the two whose path conditions still hold: it excludes all the path conditions that depends on the expression $x > threshold$. This way, since the selected *PEP* is constantly updated, *dynaSpark*$_{SEEPEP}$ becomes aware of of the amount of work still to be completed, and can use this information to refine resource provisioning.

Note that *PEPSelector* receives all the feasible *PEPs* and computes the next *PEP* to use. The selection mechanism can be customized. Currently, we always select the worst-case *PEP*, that is, the *PEP* with the greatest number of remaining stages, to be conservative and minimize deadline violations. If one wanted to optimize different performance indicators (e.g., deadlines are not strict and used resources must be minimized), the selection might privilege a *PEP* that corresponds to an average case instead of the worst one.

5 EVALUATION

We evaluated *SEEPEP*/*dynaSpark*$_{SEEPEP}$ to control the parallel execution of two example Spark applications. Our evaluation addressed two main research questions:

- **RQ$_1$** Can *dynaSpark*$_{SEEPEP}$ effectively control the execution times of Spark applications?
- **RQ$_2$** To what extend can *dynaSpark*$_{SEEPEP}$ improve the resource allocation capabilities of *dynaSpark*, given it used a single, constant *PEP*?

Below we introduce the applications we used as experimental subjects, describe the setting of our experiments, and discuss the obtained results.

5.1 Subjects

The subjects of our experiments are the applications called *PromoCalls* and *Louvain*.

Table 1: *PromoCalls* paths.

Path	F	#J	#S
0	Y	6	6
1•	Y	3	3
2	Y	7	7
3	Y	6	6
4	Y	8	8
5	Y	7	7
6†	Y	9	9
7	Y	8	8

Table 2: *Louvain* paths.

Path	F	#J	#S
0	Y	11	149
1	Y	17	364
2	Y	17	364
3	Y	11	149
4	N	-	-
5	N	-	-
6†	Y	17	364
7•	Y	8	73

PromoCalls is a paradigmatic example that was developed at our lab[4]. It mimics a batch application of a telecommunication company that wants to offer promotional discounts based on the amount of long (longer than a parametric threshold) local and long-distance calls that customers make per day. If a customer makes more than min_l long local lcalls or more than min_a long long-distance calls (or both) in a day, in the last m months, or in the current month, s/he may receive discounts. S/he may get some or all the discounts according to the possible combinations of the two triggering conditions. PromoCalls exploits Spark to efficiently analyze the data of all calls and compute applicable discounts. We used PromoCalls to guide the development of *SEEPEP*, and to initially evaluate the precision of the technique.

To evaluate our approach on a real-world application, we selected Louvain, a Spark implementation of the Louvain algorithm [7] that we downloaded from a highly rated GitHub repository[5]. Louvain exploits *GraphX*, a Spark module dedicated to graph processing, to represent large networks of users and analyze communities in these networks.

5.2 Experimental Setting

For each subject application, our experiments include three steps. First, we ran *SEEPEP* to retrieve the path conditions and generate the launchers for each corresponding path. Next, we profiled the subject application with the generated launchers, and retrieved the *PEPs* of each path. Finally, we used $dynaSpark_{SEEPEP}$ to control the execution of the application when it is fed with input datasets of an order of magnitude larger than the ones used in the launchers. We generated at least one large dataset for each profiled path. We also set the goal of guaranteeing a reasonable deadline, which we defined as 20% longer than the minimum feasible deadline measured on the datasets used in these experiments, and Spark configured to use all the resources available in the cluster.

To compare $dynaSpark_{SEEPEP}$ against the original version of *dynaSpark*, for each considered application, we identified the best and the worst cases, and the paths with the lowest and the highest number of stages[6]. This way, we quantify the error that *dynaSpark* can make, being unaware of which *PEP* was used in the profiling phase. We executed the applications on Microsoft Azure using machines of type *Standard_D14_v2* VMs with 16 CPUs, 112 GB of memory, and 800 GB of local SSD storage. This kind of VM is optimized for memory usage, with a high memory-to-core ratio. Each machine ran Canonical Ubuntu Server 14.04.5-LTS, Oracle Java 8, Apache Hadoop 2.7.2, Apache Spark 2.0.2 and *dynaSpark*. We dedicated 5 VMs to HDFS (to store the input datasets) and 5 to *dynaSpark* (5 for *dynaSpark* and 5 for $dynaSpark_{SEEPEP}$) of which 1 for the master and 4 for the workers. The datasets were generated randomly by using *SEEPEP*.

5.3 Results

Tables 1 and 2 show the results produced by $dynaSpark_{SEEPEP}$ for PromoCalls and Louvain, respectively, up to the profiling phase. Column *Path* enumerates the paths that *SEEPEP* identified: in both cases it identified 8 unique paths. Column *F* (*Found*) indicates whether or not (*Y* or *N*, respectively) *SEEPEP* succeeded in generating a test case (and thus a corresponding profiling launcher) for the identified paths: It successfully generated a test case for each path of PromoCalls, and for 6 out of 8 paths of Louvain.[7] The test case generation took 30 minutes for Louvain, and less than 10 minutes for PromoCalls. It incurred low storage overhead since input datasets are concisely represented as SDDs, that is, small linked lists of data blocks. Then, for profiling, we rendered each SDD with an RDD with 1 million replicas of each data block, thus we used RDDs whose size ranges between 3 millions and 10 millions of data records. Column *J* (*Jobs*) and column *S* (*Stages*) report the number of jobs and stages we collected when profiling the launchers with *dynaSpark*. We got between 3 and 9 jobs, and 3 and 9 stages for Promocalls, and 11 and 17 jobs, and 73 and 364 stages for Louvain, respectively. These data are not available for the two paths of Louvain for which *SEEPEP* did not generate a launcher. In both tables, we marked with • and † the paths that correspond to the best and worst case, respectively, of each application.

Tables 3 and 4 summarize the results of our experiments about the effectiveness of $dynaSpark_{SEEPEP}$ to control the execution of the subject applications when these applications execute with large datasets as inputs. Furthermore the tables include the results produced by *dynaSpark* tuned on the worst and best case datasets above to compare $dynaSpark_{SEEPEP}$ against *dynaSpark*. The different columns are explained below.

Column *Experiment* indicates, for each profiled path P_i, the data computed with $dynaSpark_{SEEPEP}$ ($dynaSpark_s$), and *dynaSpark* tuned on the worst-case dataset ($dynaSpark_w$) and on the best-case dataset ($dynaSpark_b$), respectively. Column *dl* shows the imposed deadline in seconds. Column *et* reports the actual execution time of the application in seconds: this is the average of 5 repetitions of the experiments for a total of 120 executions of PromoCalls (8 paths × 5 repetitions × 3 techniques) and 90 executions of Louvain. Column *V* (*Violation*) indicates whether the deadline was violated (i.e., $et > dl$). Column ϵ quantifies the error defined as:

$$\epsilon = \frac{|dl - et|}{dl} \cdot 100\%$$

[4] https://github.com/seepep/promocalls
[5] https://github.com/Sotera/spark-distributed-louvain-modularity
[6] In case of two paths with the same number of stages, we used the one with the shortest/longest execution time.
[7] We manually inspected the two paths of Louvain for which *SEEPEP* did not identify a corresponding test case. However, due to the complexity of the code, we were not able to either prove that these paths are infeasible, or devise input datasets that exercise these paths. Thus, we currently have no evidence of whether Louvain might indeed execute these program paths.

Table 3: Results for *PromoCalls*

Experiment	dl [s]	et [s]	V	ϵ	ca [$\frac{c}{s}$]	pen
$dynaSpark_s$	91.4	90.3	N	1.2%	41.3	–
P_0 $dynaSpark_w$	91.4	88.0	N	3.8%	53.0	28.3%
$dynaSpark_b$	91.4	143.0	Y	56.4%	30.6	∞
$dynaSpark_s$	56.4	46.0	N	18.4%	56.2	–
P_1 $dynaSpark_w$	56.4	45.3	N	19.6%	56.5	0.5%
$dynaSpark_b$	56.4	55.3	N	1.9%	38.2	-33.6%
$dynaSpark_s$	107.8	106.3	N	1.3%	39.2	–
P_2 $dynaSpark_w$	107.8	104.0	N	3.5%	52.1	32.9%
$dynaSpark_b$	107.8	175.0	Y	62.4%	29.0	∞
$dynaSpark_s$	87.5	86.0	N	1.7%	42.4	–
P_3 $dynaSpark_w$	87.5	83.0	N	5.1%	53.5	26.2%
$dynaSpark_b$	87.5	138.0	Y	57.8%	30.1	∞
$dynaSpark_s$	147.6	146.0	N	1.1%	37.0	–
P_4 $dynaSpark_w$	147.6	130.0	N	11.9%	51.4	38.9%
$dynaSpark_b$	147.6	228.0	Y	54.5%	28.4	∞
$dynaSpark_s$	77.0	75.3	N	2.2%	41.0	–
P_5 $dynaSpark_w$	77.0	70.0	N	9.1%	53.7	30.9%
$dynaSpark_b$	77.0	122.0	Y	58.4%	29.7	∞
$dynaSpark_s$	122.2	120.3	N	1.5%	39.2	–
P_6 $dynaSpark_w$	122.2	120.7	N	1.2%	43.6	11.3%
$dynaSpark_b$	122.2	204.0	Y	67.0%	27.9	∞
$dynaSpark_s$	112.1	110.7	N	1.3%	39.2	–
P_7 $dynaSpark_w$	112.1	100.0	N	10.8%	53.0	35.2%
$dynaSpark_b$	112.1	180.0	Y	60.6%	28.8	∞

Table 4: Results for *Louvain*

Experiment	dl [s]	et [s]	V	ϵ	ca [$\frac{c}{s}$]	pen
$dynaSpark_s$	184.3	180.1	N	2.3%	35.9	–
P_0 $dynaSpark_w$	184.3	142.0	N	23.0%	46.1	28.4%
$dynaSpark_b$	184.3	222.3	Y	20.6%	15.2	∞
$dynaSpark_s$	228.0	227.0	N	0.4%	32.3	–
P_1 $dynaSpark_w$	228.0	222.0	N	2.6%	33.2	2.8%
$dynaSpark_b$	228.0	329.3	Y	44.4%	7.4	∞
$dynaSpark_s$	292.8	290.7	N	0.7%	32.3	–
P_2 $dynaSpark_w$	292.8	289.0	N	1.3%	32.5	0.5%
$dynaSpark_b$	292.8	429.0	Y	46.5%	7.0	∞
$dynaSpark_s$	228.7	226.0	N	1.2%	35.5	–
P_3 $dynaSpark_w$	228.7	211.3	N	7.6%	41.4	16.6%
$dynaSpark_b$	228.7	292.0	Y	27.7%	16.0	∞
$dynaSpark_s$	163.0	159.4	N	2.2%	38.4	–
P_6 $dynaSpark_w$	163.0	158.0	N	3.0%	39.8	3.8%
$dynaSpark_b$	163.0	242.0	Y	48.5%	8.5	∞
$dynaSpark_s$	156.0	139.0	N	10.9%	33.2	–
P_7 $dynaSpark_w$	156.0	131.5	N	15.7%	43.6	31.4%
$dynaSpark_b$	156.0	152.7	N	2.1%	30.9	−7.0%

that is, the percentage ratio between the distance between the actual execution time and the deadline and the deadline itself. In general, in case of non-violations, the smaller ϵ is, the more efficient the resource allocation is, since fewer resources can be used to accomplish the goal. On the other hand, in case of violations, the smaller the error is, the shorter the delay is. Note that if the deadline were considered strict, the penalty for a violation would be considered to be infinite [27] (more on that later). Column *ca* reports the average core allocation during execution defined as:

$$ca = \frac{\sum_{s=0}^{et} coresAllocatedAtSecond(s)}{et}$$

Note that the maximum value of *ca* is 64 core/second since 64 were the cores available in the cluster used for these experiments. Finally, column *pen* quantifies how $dynaSpark_s$ performed compared to the experiments with *dynaSpark*: *pen* (*penalty*) is defined as:

$$pen = \begin{cases} \frac{ca_{WORST|BEST} - ca_{SEEPEP}}{ca_{SEEPEP}} \cdot 100\%, & \text{if } V = N \\ \infty, & \text{if } V = Y \end{cases}$$

Since $dynaSpark_s$ never violated the deadlines, *pen* measures how many resources were used by $dynaSpark_{w|b}$ with respect to $dynaSpark_s$. For example, if *pen* is equal to 30% it means that $dynaSpark_{w|b}$ used a quantity of resources that was 30% greater than the ones used by $dynaSpark_s$. In contrast, a negative *pen* implies that $dynaSpark_{w|b}$ used fewer resources than $dynaSpark_s$. Finally, if the deadline is violated by $dynaSpark_{w|b}$ we considered an infinite penalty [27].

The data in Tables 3 and 4 indicate that $dynaSpark_b$ violates the deadline in 7 cases out of the 8 paths in the experiments with PromoCalls, and 5 out of 6 paths in the experiments with Louvain. This is due to the mistakenly optimistic estimations made in the profiling phase. For example, if we consider PromoCalls, $dynaSpark_b$ computes the local deadlines and the resource allocation as if the *PEP* always consisted of 3 stages. This means that, in all the experiments but P_1 (that truly corresponds to the best-case path) *dynaSpark* under-allocated the resources, and the resulting execution time eventually exceeded the deadline by 51.9%. The maximum error (67.0%) was measured when executing P_6 (the worst-case path) where *dynaSpark* experiences the largest mismatch between the estimations computed during profiling and the actual work to do at runtime.

$dynaSpark_w$ does not violate any deadline, conversely it causes the earlier termination of the applications in most of the cases, with an error (ϵ) between 1.2% and 19.6% in the case of PromoCalls, and between 1.3% and 23% in the case of Louvain. The earlier terminations are due to the profiling-based pessimistic estimations that mistakenly assume the worst-case path of the applications as representative of all the possible paths. In particular, when dealing with paths P_1, P_4, P_5, and P_7 of PromoCalls, and P_0, P_3, and P_7 of Louvain, the error is greater than 7%, leading to significantly sub-optimal resource allocations.

In contrast, $dynaSpark_s$ does not violate any deadline and successfully provides an efficient resource allocation. The error measured in our experiments is on average equal to 3.6% for PromoCalls, where $dynaSpark_b$ and $dynaSpark_w$ make an average error of 52.4% and 8.1%, respectively, and equal to 2.9% for Louvain, where $dynaSpark_b$ and $dynaSpark_w$ make an average error of 31.6% and 8.9%, respectively. The data in columns *ca* further witness that $dynaSpark_s$ outperforms the performance of $dynaSpark_b$ and $dynaSpark_w$. $dynaSpark_b$ underestimates allocated resources so as

to make *dynaSpark* violate the deadlines in all experiments, but path P_1 in PromoCalls and path P_7 in Louvain (the best cases). In this two cases, profiled data match what happens at runtime and, therefore, $dynaSpark_b$ outperforms both $dynaSpark_w$ and $dynaSpark_s$ and minimizes the error and used resources. Compared to $dynaSpark_w$, $dynaSpark_s$ allocates on average 25.5% fewer resources with PromoCalls, and 13.9% with Louvain.

To summarize, our experiments suggest a positive answer to both research questions RQ_1 and RQ_2 since $dynaSpark_{SEEPEP}$ effectively and precisely controls the allocation of resources to execute PromoCalls and Louvain, and keeps the execution within considered deadlines with significantly smaller errors and fewer resources than the original version of *dynaSpark*.

5.4 Threats to Validity

We conducted a total of 226 experiments using two different applications: a paradigmatic example and a real-world application taken from GitHub. We showed that *SEEPEP* was able to find a test-case for 14 out 16 of the application paths statically identified with symbolic execution, and how a tool such as *dynaSpark* could benefit from the integration with *SEEPEP*. In this section, we highlight the threats that may constrain the validity of our current results [34]:

5.4.1 Internal Threats. To run the experiments we slightly modified the test cases generated by *SEEPEP* to increase the size of the datasets (without breaking the path conditions). This way, we are sure to test the desired paths and reliably obtain different repetitions of the experiments. We also ran some initial experiments with datasets created randomly and we obtained similar results to those presented; a more complete evaluation with random data will be conducted in the future.

5.4.2 External Threats. The main limit to the generalization of our results refers to considered subjects. We used two Spark applications: one uses core transformations of Spark, and one uses *GraphX* to analyze graphs. In the future, we plan to increase the number and the types of Spark applications, including those that implement machine learning solutions and use SQL.

Another limitation of the current approach lies in the profiling phase. Currently we need a profiling run for each test case (launcher) found by *SEEPEP*, but this could be an issue if the number of possible paths becomes high. In the future we plan to improve this part of the tool-chain by adopting a branch-based criterion for selecting profiling executions, instead of a path-based one, since executing all program branches guarantees that any possible *PEP* of the program be profiled. This optimization might also mitigate the issues with paths that our test generator cannot cover, as the two uncovered paths we experienced with Louvain.

6 RELATED WORK

Our work touches both the problem of managing big-data applications at design/run time, and the use of symbolic execution for analyzing parallel algorithms.

Spark natively provides means for monitoring application and analyze the response time at each stage. Spark is however not aware of the program control-flow graph, and thus only provides a rudimentary facility to adjust allocated resources at runtime [1].

Some research efforts focus on monitoring big-data applications [2, 11, 12, 16, 20, 23] with the assumption that their *PEP* is constant over different executions. Others concentrate on predicting the response time by using optimization techniques [17, 18, 29], formal approaches [21], and machine learning solutions [3]. In general, they compute a static resource allocation plan to avoid deadline violations based on the execution graph, the size of the dataset, and the nominal performance measured during profiling. These techniques do not aim to address any dynamic, elastic provisioning of resources as our approach does. Rather they compute and use a fixed resource allocation, thus failing to account for possible runtime deviations. [19, 25] address the self-adaptive resource allocation for MapReduce applications to meet deadlines by modifying the scheduler of Hadoop. Compared to our work, all these approaches are unaware of the control-flow graph of the application, and thus can achieve precise predictions only when the execution graph is truly unique, that is, only for simple applications.

Some approaches use formal analysis, including symbolic execution, to verify and improve parallel applications [22, 24, 26, 30–32]. In particular, Siegel et al. [30] use symbolic execution to analyze the equivalence between a parallel program and a corresponding sequential program, assumed to be available as specification of the parallel version. Siegel and Zirkel [32] verify assertions in message-passing parallel programs with unbounded loops by means of symbolic execution with novel loop invariants for multi-thread programs. Raychev et al. [26] rely on symbolic execution to parallelize the computation of user-defined aggregations in MapReduce platforms. None of these solutions deals with Spark applications. To the best of our knowledge, this paper is the first that combines symbolic execution and search based testing to automatically generate test cases that exercise the control-flow paths of a parallel program.

7 CONCLUSIONS AND FUTURE WORK

This paper presents *SEEPEP*, and $dynaSpark_{SEEPEP}$, to use symbolic execution and search-based test generation to compute all the parallel execution paths that are actually embedded in complex Spark applications. $dynaSpark_{SEEPEP}$ misses fewer deadlines and allocate resources more efficiently than *dynaSpark*. As for future work, we plan to optimize the profiling phase by using a branch-based selection criterion, instead of the simple path-based solution adopted in this paper.

ACKNOWLEDGMENT

We thank Davide Bertolotti for his important contribution to the implementation of this project. This work has been partially supported by the GAUSS national research project (MIUR, PRIN 2015, Contract 2015KWREMX).

REFERENCES

[1] 2017. Dynamic Resource Allocation in Spark. https://spark.apache.org/docs/latest/job-scheduling.html.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, 29–42. https://doi.org/10.1145/2465351.2465355

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of the 14th USENIX*

Conference on Networked Systems Design and Implementation (NSDI'17). USENIX Association, 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[4] Apache.org. 2019. RDD Programming Guide - Spark 2.4.0 documentation. https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[5] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi. 2018. *Fine-grained Dynamic Resource Allocation for Big-Data Applications*. Technical Report. http://hdl.handle.net/11311/1057275

[6] Luciano Baresi and Giovanni Quattrocchi. 2018. Towards Vertically Scalable Spark Applications. In *Euro-Par 2018: Parallel Processing Workshops - Euro-Par 2018 International Workshops*. Springer, 106–118. https://doi.org/10.1007/978-3-030-10549-5_9

[7] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/p10008

[8] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining Symbolic Execution and Search-based Testing for Programs with Complex Heap Inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 90–101. https://doi.org/10.1145/3092703.3092715

[9] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2018. SUSHI: a Test Generator for Programs with Complex Structured Inputs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018*. ACM, 21–24. https://doi.org/10.1145/3183440.3183472

[10] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2016. JBSE: a Symbolic Executor for Java Programs with Complex Heap Inputs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*. ACM, 1018–1022. https://doi.org/10.1145/2950290.2983940

[11] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. 2005. When Can We Trust Progress Estimators for SQL Queries?. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. ACM, 575–586. https://doi.org/10.1145/1066157.1066223

[12] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. 2004. Estimating Progress of Execution for SQL Queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, 803–814. https://doi.org/10.1145/1007568.1007659

[13] Min Chen, Shiwen Mao, and Yunhao Liu. 2014. Big Data: A Survey. *Mobile Networks and Applications* 19, 2 (2014), 171–209. https://doi.org/10.1007/s11036-013-0489-0

[14] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[15] G. Fraser and A. Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[16] H. Gao, Z. Yang, J. Bhimani, T. Wang, J. Wang, B. Sheng, and N. Mi. 2017. AutoPath: Harnessing Parallel Execution Paths for Efficient Resource Allocation in Multi-Stage Big Data Frameworks. In *26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9. https://doi.org/10.1109/ICCCN.2017.8038381

[17] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna. 2016. Stage Aware Performance Modeling of DAG Based in Memory Analytic Platforms. In *IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 188–195. https://doi.org/10.1109/CLOUD.2016.0034

[18] M. T. Islam, S. Karunasekera, and R. Buyya. 2017. dSpark: Deadline-Based Resource Allocation for Big Data Applications in Apache Spark. In *IEEE 13th International Conference on e-Science (e-Science)*. IEEE, 89–98. https://doi.org/10.1109/eScience.2017.21

[19] K. Kc and K. Anyanwu. 2010. Scheduling Hadoop Jobs to Meet Deadlines. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, 388–392. https://doi.org/10.1109/CloudCom.2010.97

[20] Kukjin Lee, Arnd Christian König, Vivek Narasayya, Bolin Ding, Surajit Chaudhuri, Brent Ellwein, Alexey Eksarevskiy, Manbeen Kohli, Jacob Wyant, Praneeta Prakash, Rimma Nehme, Jiexing Li, and Jeff Naughton. 2016. Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 1753–1764. https://doi.org/10.1145/2882903.2903728

[21] Francesco Marconi, Giovanni Quattrocchi, Luciano Baresi, Marcello M. Bersani, and Matteo Rossi. 2018. On the Timed Analysis of Big-Data Applications. In *NASA Formal Methods*. Springer, 315–332. https://doi.org/10.1007/978-3-319-77935-5_22

[22] Olga Shumsky Matlin, Ewing Lusk, and William McCune. 2002. SPINning Parallel Systems Software. In *Model Checking Software*. Springer, 213–220. https://doi.org/10.1007/3-540-46017-9_16

[23] Kristi Morton, Magdalena Balazinska, and Dan Grossman. 2010. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, 507–518. https://doi.org/10.1145/1807167.1807223

[24] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. 2010. Formal Methods Applied to High-performance Computing Software Design: A Case Study of MPI One-sided Communication-based Locking. *Software Prac. Experience* 40, 1 (2010), 23–43. https://doi.org/10.1002/spe.v40:1

[25] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley. 2010. Performance-driven Task Co-scheduling for MapReduce Environments. In *IEEE Network Operations and Management Symposium - NOMS 2010*. IEEE, 373–380. https://doi.org/10.1109/NOMS.2010.5488494

[26] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing User-defined Aggregations Using Symbolic Execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, 153–167. https://doi.org/10.1145/2815400.2815418

[27] K. G. Shin and P. Ramanathan. 1994. Real-time Computing: A New Discipline of Computer Science and Engineering. *Proc. IEEE* 82, 1 (1994), 6–24. https://doi.org/10.1109/5.259423

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[29] S. Sidhanta, W. Golab, and S. Mukhopadhyay. 2016. OptEx: A Deadline-Aware Cost Optimization Model for Spark. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 193–202. https://doi.org/10.1109/CCGrid.2016.10

[30] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. 2008. Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. *ACM Transactions on Software Engineering and Methodology* 17, 2 (2008), 10:1–10:34. https://doi.org/10.1145/1348250.1348256

[31] Stephen F. Siegel and Louis F. Rossi. 2008. Analyzing BlobFlow: A Case Study Using Model Checking to Verify Parallel Scientific Software. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 274–282. https://doi.org/10.1007/978-3-540-87475-1_37

[32] Stephen F. Siegel and Timothy K. Zirkel. 2012. Loop Invariant Symbolic Execution for Parallel Programs. In *Verification, Model Checking, and Abstract Interpretation*. Springer, 412–427. https://doi.org/10.1007/978-3-642-27940-9_27

[33] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, 275–287. https://doi.org/10.1145/1272996.1273025

[34] Claes Wohlin, Martin Höst, and Kennet Henningsson. 2006. *Empirical Research Methods in Web and Software Engineering*. Springer, 409–430. https://doi.org/10.1007/3-540-28218-1_13

[35] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*. USENIX Association. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets