POLITECNICO DI MILANO

Facoltà di Ingegneria

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in

Computer Science and Engineering

Using Symbolic Execution to Improve the Runtime Management of Spark Applications

Supervisor:

PROF. LUCIANO BARESI

Co-Supervisor:

DR. GIOVANNI QUATTROCCHI

Master Graduation Thesis by:

DAVIDE BERTOLOTTI
Student Id n. 787366

Academic Year 2017-2018

[March 25, 2019 at 15:26 – version 1.0]

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

The template has been adapted by Emanuele Mason, Andrea Cominola and Daniela Anghileri: *A template for master thesis at DEIB*, June 2015

Here you can put your dedication, like:

To time, that do not go backwards

— A & D & E

ACKNOWLEDGMENTS

Here you can put acknowledgements to people that helped you during the thesis. Remember that helping students to write thesis is part of the job of some of them, and they're also paid for that. Please make sure to thank them for what they weren't supposed to do.

Remember also that this page is part of your thesis. I know that your boyfriend/girlfriend is very important to you and you cannot live without her/him, as it is for me. But there's no need to put her/his name here unless she/he gave a proper contribution to this work. Same goes for friends, parents, drinking buddies and so on.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ABSTRACT

The need to crunch a steadily growing amount of data generated by the modern applications is driving an increasing demand of flexible computing power, that is more and more often satisfied by cloud computing solutions. Cloud computing has revolutionized the way computer infrastructures are abstracted and used. It is built on abstract hardware and software infrastructures accessible via the Internet and its usage is suitable for big data processing by enterprises of any size. Big data is a massive amount of structured and unstructured data whose size is so large that it makes it very difficult, in many practical cases impossible, the processing with traditional database and software approaches. Dealing with large datasets makes it difficult to create, manipulate and manage data, especially about search and analysis. In addition, big data applications pose a new challenge to the Quality of Service (QoS) provided to users. In particular, users may be interested in quantifying and constraining the execution time of every single run of an application. Cluster computing, an implementation of the Parallel Computing paradigm composed by a large number of networked multi-cpu computers in the cloud, is widely used to speed-up application execution time. One of the most frequently used cluster computing frameworks for big data analytics is Apache Spark, which provides a fast and general, fault-tolerant data processing platform that allows quick in memory computation. Spark computation is based on RDDs, a data abstraction, and DAGs, representing the data manipulation processes. xSpark, developed at Politecnico di Milano, is an extension of the Apache Spark framework that offers fine-grained dynamic resource allocation using lightweight containers. It allows users to constrain the duration of the execution of an application by specifying a deadline. This is possible thanks to the knowledge of the application Directed Acyclic Graph (DAG), generated by running the application in profiling mode, and the runtime allocation of resources to task executors by a specialized xSpark's control loop, composed by a centralized heuristic and a distributed local controller. All the above works under the assumption that the application execution flow is represented by a single DAG, which is true when the application code does not contain any conditional branch whose outcome depends on user input values or the result of previous calculations involving input data. When the application contains such conditional branches, a family of DAGs (or a tree of DAGs) is needed to describe all the possible execution flows corresponding to the combinations of all the different branch outcomes. Here is where xSpark-dagsymb, the project described in this

thesis, comes into play. xSpark-dagsymb extends xSpark capability to safely run multi-DAG applications, by exploiting symbolic execution (techniques, principles or theory?). At each decisional branch outcome in the application, xSpark-dagsymb determines which DAGs are still valid and prunes the DAG tree, removing the invalid DAGs, thus leaving only the valid ones in the DAG tree. A heuristic is used to select the DAG to execute among the valid ones, in order to minimize the risk of missing the deadline while maximizing the CPU usage efficiency.

SOMMARIO

Per abstract si intende il sommario di un documento, senza l'aggiunta di interpretazioni e valutazioni. L'abstract si limita a riassumere, in un determinato numero di parole, gli aspetti fondamentali del documento esaminato. Solitamente ha forma "indicativo-schematica"; presenta cioé notizie sulla struttura del testo e sul percorso elaborativo dell'autore.

Max 2200 caratteri compresi gli spazi.

ESTRATTO

"... il testo delle tesi redatte in lingua straniera dovrà essere introdotto da un ampio estratto in lingua italiana, che andrà collocato dopo l'abstract."

1 INTRODUCTION

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

— Verne *Journey to the Center of the Earth* 1957

Cloud computing has become a widely used form of service oriented computing, where infrastructure and solutions are offered as a service. The cloud has dramatically changed the way computing infrastructures are abstracted and used. Some of the most intriguing features of cloud computing are elasticity (e.g. on demand resource scaling), pay-per-use, no upfront capital investment, low time to market and transfer of risk.

The term "big data" is used to describe a large amount of data that can be structured, like in the traditional relational databases, semi-structured, like in the self-described XML or JSON documents, or unstructured, like in the logfiles collected mostly by web applications to monitor usage or other user's preferences. More properly, we call big data those that cannot be handled using traditional database and software technologies.

1.1 CONTEXT

Today, every second 8,411 Tweets are sent, 902 Instagram photos are uploaded, 1,502 Tumblr posts are created, 3,690 Skype calls are done, 73,116 Google searches are performed and 2,780,000 emails are sent [43]. These data are collected and analyzed.

Gartner defines big data as data that contains greater variety arriving in increasing volumes and with ever-higher velocity. This is known as the three V's characterizing big data: Volume, Velocity, Variety [72].

Volume is important because the amount of data drives both the size of memory infrastructure needed to hold them and the computation effort for their analysis. With big data, you'll have to process high volumes of low-density, unstructured data. This can be data of unknown value, such as Twitter data feeds, clickstreams on a webpage or a mobile app, or sensor-enabled equipment. In some cases, this might be tens of terabytes of data, sometimes hundreds of petabytes.

Velocity is the fast rate at which data is received and (perhaps) acted upon. Normally, highest velocity data streams are directly stored into memory versus being written to disk. Some internet-enabled smart products operate in real time, or near real time, requiring a very fast evaluation and action.

Variety refers to the many available data types. Traditional data types were structured and are suitable to be stored and managed in a relational database. With the rise of big data, data arrives in an unstructured form. Unstructured and semistructured data types, such as text, audio, and video require an additional preprocessing effort to transform, derive meaning and attach metadata to them.

Storing and processing big volumes of data requires scalability, fault tolerance and availability [62].

Scalability means the ability to maintain a near-linear progression between size of data to process and computational resources to perform the task. A big challenge to scalability is the overhead to keep the numerous chunks of intermediate results of the data processing steps in synch. Such overhead could drain so many resources to hinder scalability already above modest scale factors.

Fault tolerance is a technology challenge in big data, especially when processing involves many networked nodes and it becomes cumbersome to retain all the checkpoints/restarts to be enacted upon partial processing failures. Devising 100% reliable systems is not an easy task, however the systems can be architected so that the probability of failure falls within the allowed range.

By means of hardware virtualization, cloud computing services satisfies all the requested requisites needed to manipulate big data. Elasticity and redundancy provided by cloud computing also enable big data application high availability, scalability and fault tolerance. Big data also represent an unprecedented business opportunity for many companies which started to deliver big data applications as a service. According to SoftwareTestingHelp [15], these are the top 10 big data companies of 2019: IBM [42], HP Enterprise [41], Teradata [67], Oracle [58], SAP [63], Dell EMC [31], Amazon [10], Microsoft [55], Google [37], VMware [70].

Big data applications are used to transform, aggregate and analyze a large amount of data in an easy and efficient way. Specialized frameworks are used to transform these applications in atomic parts that can be executed in a distributed cluster of physical or virtual machines. The limit to the level of parallelism we can obtain is given by the number of machines and the amount of synchronization (e.g. aggregations, grouping) needed among the chunks of data representing the intermediate results. This paradigm has been historically represented by the map-reduce programming model firstly introduced by Google [38]. Nowadays, more advanced solutions are available, such as Apache Spark [4] and Apache Tez [6] that provide a greater flexibility and allow building large-scale data processing applications using a DAG based structure.

One of the most popular cluster computing framework for big data analytics is Apache Spark [59]. Spark provides a fast and general data processing platform, letting users execute programs 100x faster

in memory or 10x faster on disk than Hadoop, indeed in 2014 it won the Daytona GraySort contest as the fastest open source engine for sorting a petabyte [53]. Spark is fault-tolerant and is designed to run on commodity hardware. It generalizes the two stage Map-Reduce to support arbitrary DAG. The main advantage of Spark with respect to previous cluster computing frameworks is the fast data sharing between operations. For example, Apache Hadoop requires intermediate data to be written to disk in order to be accessible by the following operations, Spark instead allows to execute in-memory computing. Spark offers a quick way of writing code by means of high-level operators provided in the API: Spark Core, Spark SQL, Spark Streaming, MLlib (machine learning), GraphX (graph). Spark integrates well with various storage systems, including Amazon S3, Hadoop HDFS and any POSIX-compliant file system. Spark provides its own cluster manager, but it can also run on clusters managed by Hadoop Yarn or Apache Mesos. Spark is often used for in-memory computation, but is also capable of handling workloads whose size exceeds the aggregate cluster memory.

In order to execute big data applications, Spark [74] divides the computation into different phases and split the input dataset into partitions that are stored in a distributed fashion and processed in parallel. Spark exploits in-memory processing and storage as a means to reuse partial results. Spark applications can be written in Java, Python, or Scala and exploit two types of dedicated operations: *action*s and *transformation*s. Actions trigger (distributed) computations that return results to the application. Transformations carry out data transformation in parallel. Spark groups operations into *stage*s and then into *job*s. A stage is composed by 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 directed acyclic graphs of stages, while the complete *PEP* of an application is simply the sequence of the *PEPs* of its jobs.

The execution of Spark applications is based on the definition of the execution order and parallelism of the different jobs, given data and available resources. Spark keeps track of these dependencies in a graph that we will refer to as the (parallel) execution plan of the application, also called *PEP*.

A very important measure related to IT applications is the *Quality of Service* (*QoS* in the reminder of this document).

The notion of *QoS* in big data application differ by application type. Interactive applications are usually assessed according to response time or throughput, and their fulfillment depends on the intensity

and variety of the incoming requests. Big data applications might require a single batch computation on a very large dataset, thus *QoS* must consider the execution of a single run. In this domain *QoS* is often called deadline, or the desired duration of the computation. Many factors influence the duration of an application execution, surely resource allocation and scheduling greatly influence the duration.

A resource allocation problem arises when applications with different structures run in contexts with different amount of resources or size of input datasets. A scheduling problem arises when many applications run concurrenlty on the same hardware, so that each application cannot have the totality of the resources assigned to itself. Satisfying deadline-based *QoS* constraints is a problem related to resource allocation, since the amount of allocated resources determines the duration of the execution of Spark applications. The simplest option available on all cluster managers is static partitioning of the resources. This way, each application is given a maximum amount of resources it can use, and holds them for the whole execution time. Memory sharing across applications is currently not provided. Spark also provides a mechanism to dynamically adjust the resources assigned to a specific application according to the workload. Applications may give resources back to the cluster if they are no longer used and re-acquire them again when needed.

xSpark, developed at Politecnico di Milano, is an extension of Spark that offers fine-grained dynamic resource allocation using lightweight containers and enforces *QoS* constraints. xSpark estimates the execution times and allocate resource in order to meet user defined deadlines. A previous work on xSpark has addressed the scheduling problem and established a policy for managing the deadlines when multiple applications run simultaneously on the same hardware.

We have mentioned availability, fault tolerance and availability as fundamental requirements and *QoS* as a measure of the capability to meet a user-defined deadline when processing big data.

The work in this thesis will focus on *QoS*.

1.2 PROBLEM AND MOTIVATION

The growing importance of big-data applications has favoured the birth of many analysis techniques to estimate the execution time of Spark applications and perform a proper allocation of resources. For example, Islam et al. [44] propose a solution to statically allocate resources to deadline-constrained Spark applications while minimizing execution costs. Sidhanta et al. [66] estimate the duration of Spark applications using a closed-form model based on the size of the input dataset and the size of the available cluster. Alipourfard et. al [1] use Bayesian optimization to generate performance models of Spark applications and compute the best configuration for their execution.

Baresi et al. [13, 12] propose xSpark, an extension of Spark that exploits control theory and containers[1] to scale allocated resources elastically given the execution times of interest and the other applications on the same cluster.

All the previous works regarding the estimation of the execution times and dynamic provisioning of resources have always assumed the uniquess of the execution plan for an application, given the computing resources available. This assumption is a simplification of real-world applications, since the actual execution plan is generally different across different program paths when the application code includes conditional branches and loops. Hence, the assumption of a unique *PEP* limits the precision of the analysis and prediction techniques.

xSpark offers optimized and elastic provisioning of resources in order to meet user defined deadlines. This is obtained by using nested control loops. A centralized loop is implemented on the master node and controls the execution of different stages of an application. Multiple local loops, one per executor, focus on task execution. xSpark exploits metadata provided by an initial profiling to create an enriched annotated DAG of the application hat holds information about the execution of the stages. At runtime, the annotated DAG is used to understand how much work has already been done and how much work remains to complete the application. Since all executions of the same application use the same DAG, we infer that an xSpark implicit requirement is that applications cannot contain branches or loops, which might be resolved in different ways at runtime.

The xSpark centralized control loop is activated before the execution of each stage and uses a heuristic to assign a stage deadline and calculate the required CPU cores needed to satisfy that deadline, using the information contained in the enriched DAG and the user-provided application deadline. Many factors can influence the actual performance and invalidate the prediction. Local control loops have the objective to counteract those imprecisions, by dynamically modifying the amount of CPU cores assigned to the executors during the execution of a stage. A control theory algorithm determines the amount of CPU cores to be allocated to the executor for the next control period. Docker is used to tune the number of CPU cores allocate to the executors, which are run inside lightweight containers [29]. xSpark is able to use less resources than native Spark and can complete executions with less than 1% error in terms of set deadlines.

The capability to meet a particular deadline resorts to resource allocation problems (different resources required by different applications) or scheduling problems (more than one application running concurrently on the same hardware). Both the class of problems have been addressed by previous works on xSpark.

1 https://www.docker.com.

As mentioned earlier in this document, all these approaches implicitly assume that the *PEP* of an application is unique for any possible input dataset and input parameters, and this assumption is valid only for the most trivial applications. In fact, 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 *PEP*. For example, a Spark application can evaluate partial results through actions, and then use these partial results in conditional expressions of program branches.

Symbolic execution tecnhiques are exhaustively employed in testing of software programs to help identify unsafe inputs that cause programs' crash. The use of these techniques to deliver full-fledged cover of all prossible dangerous inputs is severely limited by the exponential growth of computational resources as a consequence of *path explosion*, due to the need to explore all the possible execution paths and solve all the corresponding *path conditions* of a complex program that has many iterative loops and/or conditional branches in its code. As a consequence, we can rapidly resort to unsolvability. The work of this thesis relies on a lightweight symbolic execution approach [8] that values the solvability of symbolic evaluation-based tools on an efficiency base instead of a *soundness and completeness* – often practically unfeasible – base.

In this thesis work we investigate the resource allocation problem when running big data multi-DAG applications with deadline-based *QoS* constraints. The outcome of the research should give an answer the following preliminary research questions:

Preliminary$\mathbf{RQ_1}$ **:** Does the solution provide an effective (w.r.t. obeying deadlines) runtime control of the Spark applications?

Preliminary$\mathbf{RQ_2}$ **:** Does the solution provide an efficient (w.r.t. resource usage) runtime management of the application?

The main reason behind this research and thesis work is to address the problem of executing multi-DAG deadline-constrained big data Spark applications, and give an answer to the above questions inherently associated to the identified problem.

1.3 SOLUTION AND CONTRIBUTION

To address the questions raised by the investigation of resource allocation problems related to running big data multi-DAG applications with deadline-based *QoS* constraints, we propose a solution that leverages *symbolic execution* principle.

The solution covered by this thesis proposes the use of an original combination of lightweight symbolic execution and search-based test generation to properly infer the actual *PEP*, for a given set of input

data and parameters [8]. Our approach is called *SEEPEP* (*Symbolic Execution-driven Extraction of Parallel Execution Plans*) and relies on a lightweight 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* and profile the application. The approach is supported by a prototype tool, also called *SEEPEP*, that identifies the PEP* of applications, that is, the set of *PEPs*, along with their respective path conditions and profiling data. It also incrementally evaluates the PEP* against the concrete values of the symbolic variables, to stay aware of the *PEPs* for which the path conditions continue to hold. This information can then be used to refine the actual *PEP* used to concretely execute the application.

Our proposed solution is built on a version of xSpark that does NOT support multiple concurrent applications.

To properly investigate the benefit of *SEEPEP*, we integrated the tool into a new version of xSpark called $xSpark_{\text{SEEPEP}}$ to understand how the dynamic selection of the best *PEP* can help in allocating resources more efficiently. A special-purpose component is now in charge of providing the worst-case *PEP*. At each execution step, the component feeds xSpark with the actual worst-case *PEP*, chosen among those that are still feasible (i.e. the ones for which their path condition still holds true).

In light of the proposed concrete solution, we can now refine the research questions as follows:

RQ$_1$: Can *SEEPEP* effectively control the execution of the Spark applications?

RQ$_2$: To what extend can *SEEPEP* improve the resource allocation capabilities of xSpark, given it used a single, constant *PEP*?

The aim of this work is to give a contribution in terms of knowledge about the application of symbolic execution to the theoretical solution of the identified problem, and a contribution in practical terms by providing:

i) a modified xSpark platform to enable the runtime management of multi-DAG deadline-constrained big data applications and

ii) a toolchain to identify the applications' execution paths, extract their associated path conditions and generate a path condition evaluator, submit the application and its metadata to the modified xSpark platfom and collect performance data for evaluating the QoS of the execution.

RESULTS AND FUTURE WORKS

The solution was tested with two applications: *Promocalls*[2], a paradigmatic example develop at Deib Labs of Politecnico di Milano, and Louvain, a Spark implementation of the *Louvain* algorithm [16], that we downloaded from a highly rated GitHub repository[3]. Louvain exploits *GraphX*, a Spark graph processing library to represent large networks of users and analyze communities in these networks.

We evaluated *SEEPEP* by integrating it with xSpark to control the parallel execution of two example Spark applications.

The results of the tests confirm the validity of our claim: $xSpark_{\text{SEEPEP}}$, being aware of the different PEPs generated by Spark applications, helps analyze and control their performance/execution time, and thus misses fewer deadlines and allocates resources more efficiently than xSpark.

Future works will address the performance improvement of the profiling phase, by using branch-based selection criteria instead of the simple path-based solution adopted in our presented solution.

2 https://github.com/seepep/promocalls

3 https://github.com/Sotera/spark-distributed-louvain-modularity

2 STATE OF THE ART

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

— Verne *Journey to the Center of the Earth* 1957

2.1 BIG DATA

The term "big data" is used to describe a large amount of data, that can be structured, like in the traditional relational databases, semi-structured, like in the self-described XML or JSON documents, or unstructured, like in the logfiles collected mostly by web applications to monitor usage or other user's preferences. More properly, we call big data those that cannot be handled using traditional database and software technologies. Today, every second 8,411 Tweets are sent, 902 Instagram photos are uploaded, 1,502 Tumblr posts are created, 3,690 Skype calls are done, 73,116 Google searches are performed and 2,780,000 emails are sent[1]. This data is collected and analyzed. Gartner's definition of big data is: "Big data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation"[2]. This defintion relates to something known as the three V's characterizing big data: Volume, Velocity,Variety [72]. Volume is important because the amount of data matters. With big data, you'll have to process high volumes of low-density, unstructured data. This can be data of unknown value, such as Twitter data feeds, clickstreams on a webpage or a mobile app, or sensor-enabled equipment. For some organizations, this might be tens of terabytes of data. For others, it may be hundreds of petabytes. Velocity is the fast rate at which data is received and (perhaps) acted on. Normally, the highest velocity of data streams directly into memory versus being written to disk. Some internet-enabled smart products operate in real time or near real time and will require real-time evaluation and action. Variety refers to the many types of data that are available. Traditional data types were structured and fit neatly in a relational database. With the rise of big data, data comes in new unstructured data types. Unstructured and semistructured data types, such as text, audio, and video require additional preprocessing to derive meaning and support metadata. By means of hardware virtualization, cloud computing services satisfies

1 Data source: http://www.internetlivestats.com/
2 https://www.gartner.com/it-glossary/big-data/

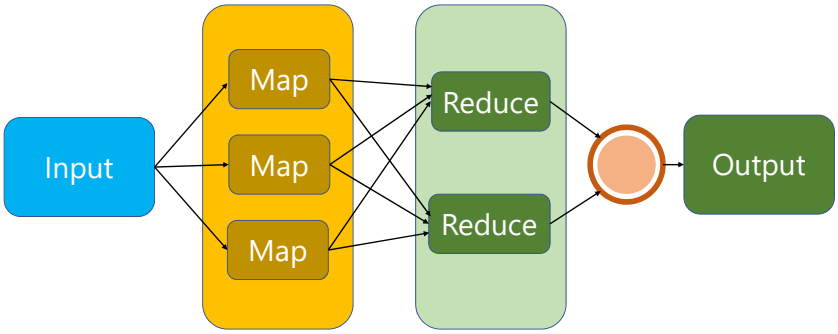

Figure 2.1: Abstract representation of map-reduce paradigm

all the requested requisites needed to manipulate big data. Elasticity and redundacncy provided by cloud computing also enable big data application high availability, scalability and fault tolerance. Big data also is an unprecedented business opportunity for many companies which deliver big data applications as a service. According to SoftwareTestingHelp [15], these are the top 10 big data companies of 2019: IBM [42], HP Enterprise [41], Teradata [67], Oracle [58], SAP [63], Dell EMC [31], Amazon [10], Microsoft [55], Google [37], VMware [70].

2.1.1 *Batch Processing: Hadoop*

MapReduce is a software framework introduced by Google in order to support the distributed computation of large dataset in cluster of computers. The framework is inspired by map and reduce functions used in functional programming, even though their purpose in the MapReduce framework is not the same as in the original form. There are available MapReduce libraries written in different programming languages.

MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.[3].

- MapReduce consists of two distinct tasks – Map and Reduce.
- As the name MapReduce suggests, reducer phase takes place after mapper phase has been completed.

3 Extracted from: https://www.edureka.co/blog/mapreduce-tutorial/

- So, the first is the map job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
- The output of a Mapper or map job (key-value pairs) is input to the Reducer.
- The reducer receives the key-value pair from multiple map jobs.
- Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

There are open source implementation of the MapReduce framework, for example Apache Hadoop. The MapReduce framework is composed by different functions for each step:

1. Input Reader
2. Map Function
3. Partition Function
4. Compare Function
5. Reduce Function
6. Output Writer

The Input Reader reads the data from mass memory and splits the input in S different splits, with a fixed dimensions (e.g., 64 MB) that are successively distributed to M machines of the cluster that have the Map Function. The Input Reader has also the goal of generating a pair (key, value). The N machine of the cluster are divided in 1 master, whose goal is to detect idling slaves and assign them a task, and N - 1 slaves that receive the tasks assigned by the master node. In total, M Map tasks and R Reduce tasks are assigned. A slave that has been assigned the M-th task reads the content of the input, extracts the (key, value) pairs and send them to the Map function defined by the user, that generates zero or more (key, value) pairs as output. These pairs are buffered in memory. Periodically the buffered pairs are cached on disk and partitioned in R sections by the partition function. The addresses of the partitioned sections are sent to the master node which is responsible of rotating the location of the slaves that will process the Reduce function. Between the slave with the Map function and the one with the Reduce one, all the pairs are reordered in order to find the ones that point at the same value, and thus also have the same key. The so called shuffling phase is the process that is used to transfer data from mappers to reducers. Once all the keys that point to the same value have been found using the compare function, a merge operation is performed. The sorting operation is useful because in this way the reducer can know when a new reduce task should start. For

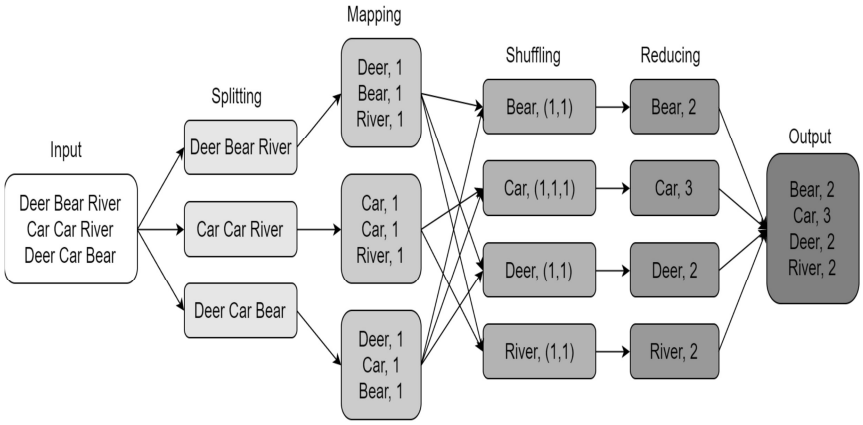

Figure 2.2: Map-Reduce word count job example. Counts the occurrence of each word in input

each of the keys, the associated slave iterates on all the keys, takes the values with the same key and then applies the Reduce function defined by the user, generating one or more element in output. The Output Writer has the goal of writing the results back to mass storage. A sample word count application can be seen in figure 2.2 The input is a document containing words, our goal is to compute the number of occurrence of each of the words in the document. Each Map task applies its function on a line of the document, emitting for each of the words in the line a pair ('word', 1). For example if the input line is "Dear Bear River", it is split into ["Dear", "Bear", "River"] and then mapped into [("Dear", 1), ("Bear", 1), ("River", 1)]. After shuffling the map results, the Reduce task receives a word and a list containing as many ones as the times the word appeared in the document, the reduce function will simply sum the ones in the list, emitting as a result the pair ('word', 'count'). For example, a reducer can receive the key "Bear" with list of values (1, 1), this is reduced into ("Bear", 2). Reducers results are then collected and stored in mass storage.

Apache Hadoop[4] is an open-source framework for distributed storage and processing of big datasets using MapReduce programming model.

Apache Hadoop MapReduce cluster have a centralized structure composed by a single master Job Tracker (JT) and multiple worker nodes running Task Tracker (TT), as shown in figure 2.3 JT main goal is organizing the job tasks on the slave nodes and continuously monitor the Task Trackers by means of heartbeats. Heartbeats provide

4 url: https://hadoop.apache.org/docs/

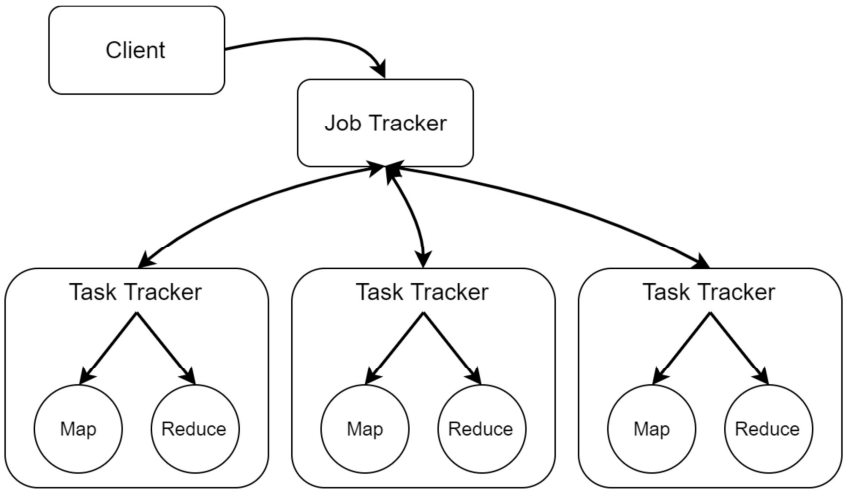

Figure 2.3: Hadoop Map-Reduce Architecture.

a way to retrieve information about the liveliness of the slaves and to inspect the progress of the executions of the different tasks. In order to be fault tolerant, if a task execution fails, it is re-executed possibly on a different slave. JT has the role of the cluster manager, so it needs also to check the admissibility of the submitted MapReduce jobs. TT have the objective of running the assigned task. They reply to heartbeats in order to affirm their liveliness and to update the master about the progress of the assigned tasks. They are configured with a fixed number of map and reduce task slots. As previously introduced, Apache Hadoop also offers a distributed file-system that stores data on different machine, providing an high aggregate bandwidth across the cluster. This functionality is called Hadoop Distributed File System (HDFS)[5]. It is highly fault tolerant and designed to be deployed on low cost hardware. HDFS exposes a filesystem namespace and allows user data to be stored in files and retrieved. Cluster structure is similar to the one of MapReduce cluster, with one master and multiple slaves, as we can see from figure 2.4. The master is composed by a single Name Node (NN), that manages the file system namespace and regulates access to the files by clients. NN executes filesystem operations such as opening, closing, renaming files and directory, but the most important operation performed is keeping track of the mapping between blocks and Data Node (DN). Indeed a file stored in HDFS is split into one or more blocks, and those blocks are stored in the Data Node (DN). Data Node (DN) represent the slaves, they are usually one per node, and manage the storage that is attached to the node they are running on.

5 *HDFS Architecture Guide.* url: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

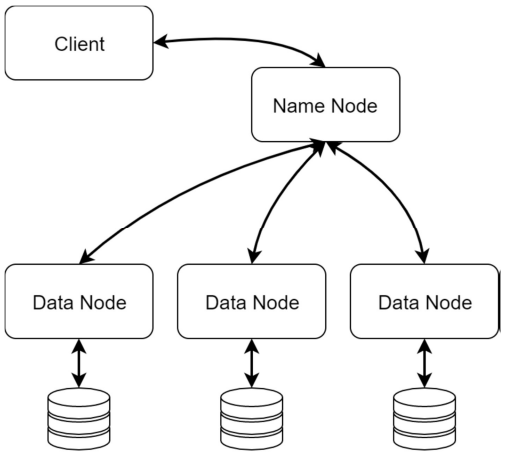

Figure 2.4: HDFS Node Strucure.

They are responsible for serving read and write operation requests from the clients, but also can perform block creation, deletion and replication. Block replication is a significant way to improve fault tolerance.

2.1.2 *Batch Processing: Spark*

Apache Spark is an open source framework for distributed computation [4], that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. With respect to the MapReduce paradigm, the in-memory multilevel primitives of Spark allow to have performances up to 100 times better in certain applications. Spark can work as standalone or on a cluster manager such as Apache Hadoop Yarn or Apache Mesos. It also needs a distributed storage and can natively use HDFS and other solutions.

Spark has been designed as a unified engine for distributed data processing. Its programming model resembles the one of MapReduce, but it is extended with a data sharing abstraction called Resilient Distributed Dataset (RDD). Using this abstraction, a wide range of processing workloads can be captured, including SQL, streaming, machine learning and graph processing. The generality of the Spark approach gives great benefits: firsty, applications are easier to develop since there is a unified API, secondly, it is a lot easier to combine processing tasks. With previous distributed computation framework we needed to write data to mass storage before using them in another engine. Spark, instead, can reuse the same data, often kept in memory. The programming abstraction at the foundation of Spark is the Re-

silient Distributed Dataset (RDD), that is a fault-tolerant collection of objects partitioned across the cluster and can be processed in parallel. The users create RDD's by applying operations called "transformations", such as map, filter and group-by, on the data. RDD's can be backed by a file obtained from an external storage. Spark evaluates RDD's in a lazy mode. This allows the construction of an efficient plan to execute the computation requested by the user. In particular, every transformation operation returns a new RDD, that is the representation of the result of the computation, but the computation is not executed immediately after the transformation request is encountered, but only when a Spark action is met. When an "action" is requested by the user code, Spark checks the entire graph of the transformation and uses it to create an efficient execution plan. For example, if there are many filters and maps in a row, Spark can merge them together and execute as a single operation. RDDs also offer an explicit support to data sharing among the computations that are ephemeral by default, but can be persisted to disk or memory for rapid reuse. This data sharing is one of the main differences between Spark and the previous computing models like MapReduce, because all the other operations that Spark can perform are similar to the ones of MapReduce. The data sharing capability allows huge speedups, up to 100 times, in particular when executing interactive queries and iterative algorithms. RDDs can also recover automatically from a failure. Traditionally, fault tolerance in distributed computing was achieved by means of data replication and checkpointing. Spark instead uses a different approach called lineage. Each RDD keeps track of its transformation graph used to generate the RDD and re-executes the transformation operations on the base data to recover every lost partition. The data recovery based on lineage is significantly more efficient than replication in case of data-intensive workload. In general, recovering lost partitions is faster than re-executing the entire program. Spark was designed to support different external systems for persistent storage, usually it is used in conjunction with a clustered file system like HDFS. Spark is designed as a storage-system-agnostic engine, to make it easy to run computations against data from different sources.

Different high-level libraries have been developed in order to simplify the creation of programs that can run in Spark framework.

- SQL and DataFrames: support for relational queries, that are the most common data processing paradigm
- Spark Streaming: implements incremental stream processing using a model called "discretized streams", input data is split into micro batches
- GraphX: graph computation interface
- MLlib: machine learning library, more than 50 common algorithms for distributed model training

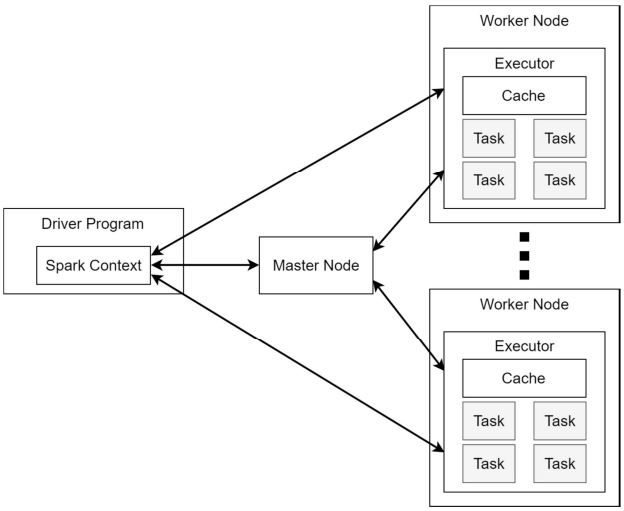

Figure 2.5: Spark Standalone Architecture.

Spark architecture follows the master/worker paradigm (figure 2.5). A master server accepts data and processing request, split them into smaller chunk of data and simpler actions that can be handled in parallel by the multiple workers. A Spark application is executed inside a driver program, that makes the user code executable on the computing cluster using a SparkContext. The driver program is responsible for managing the job flow and scheduling tasks that will run on the executors. The SparkContext will split the requested operations in tasks the can be scheduled for distributed execution on the workers. When a SparkContext is created, a new Executor process is created on each worker. An executor is a separate Java Virtual Machine (JVM) that runs for the entire lifetime of the Spark application, executes tasks using a thread pool and store data for its Spark application. Communication between the SparkContext and the other components is performed using a shared bus. When an application is submitted to Spark, it is divided in multiple jobs. Jobs are delimited by Spark actions in the application code. Spark actions are those operations that return a value to the driver program after running a computation on the dataset. For each job, a Directed Acyclic Graph (DAG) is created to keep track of the RDDs that are materialized inside the job. DAG nodes represent the RDDs, meanwhile arcs represent transformations, that are the operations that create new datasets from existing ones. The application steps inside a single job are further organized into stages, that are delimited by operations require data reshuffling, that will inevitably break locality. Spark distinguishes between narrow transformations, that do not reshuffle data (e.g., map, filter), and wide transformations, that require data reshuffling (e.g., reduceByKey). Stages are also used

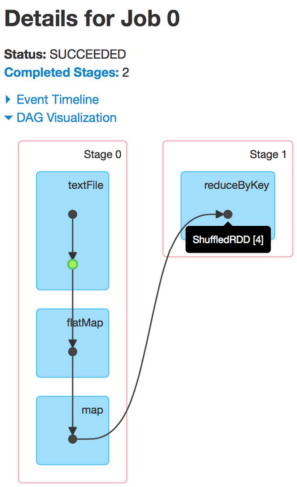

Figure 2.6: Spark DAG Example.

Listing 2.1: Spark word count application example.

```
sparkContext.textFile("hdfs ://. . . ")
.flatMap(line => line.split(" "))
.map(word => (word, 1)).reduceByKey(_ + _)
.saveAsTextFile("hdfs ://. . . ")
```

to produce intermediate result that can be persisted to memory or mass storage to avoid re-computation. When all stages inside a job have been identified, Spark can determine which parallel tasks need to be executed for each stage, and schedule them for operation on the executors. Spark creates one task for each partition of the RDD received in input by a stage.

In Figure 2.6, we can see a simple DAG representing the single job of the word count application presented in Listing 2.1 [57]. The image was obtained from SparkWeb UI. Through a textFile operation, the input file is read from HDFS. Then a flatMap operation is applied to split each of the lines of the document into words. Following, another map is used to create ('word', 1) pairs. Finally, a reduceByKey operation is performed to count the occurrences of each word. The blue boxes represent the Spark operations that the user calls in his code, while the dots represent the RDDs that are created as a result of these operations. Operations are grouped into stages, represented by the boxes with a red border. The job has been divided into two stages because the reduceByKey transformation requires the data to be shuffled. The green dot represents a cached RDD, in particular the data

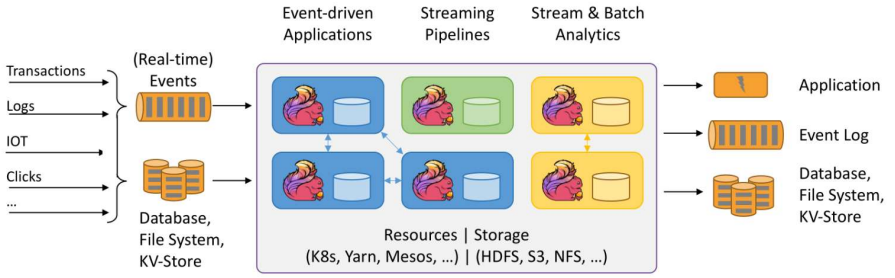

Figure 2.7: Apache Flink® - Stateful Computations over Data Streams.

read from HDFS has been cached, in this way future computations on this RDD can be done faster since data will be read from memory instead of HDFS. The default deployment of Spark is in standalone mode, that is using its embedded cluster manager.

2.1.3 *Streams Processing: Flink*

Apache Flink [2] is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. It can run in any commonly used cluster environments, computes at in-memory speed and manages data at any scale.

Flink's architecture is suitable for processing unbounded and bounded data. Any type of data is generated as a stream of events. Sensors data, server logs or user interactions on a website or mobile application, credit card transactions, all of these data are generated as streams of data.

Data can be processed as unbounded or bounded streams [3].

Unbounded streams have a start but no defined end. They do not terminate and provide data as it is generated. Unbounded streams must be continuously processed, i.e., events must be promptly handled after they have been ingested. It is not possible to wait for all input data to arrive because the input is unbounded and will not be complete at any point in time. Processing unbounded data often requires that events are ingested in a specific order, such as the order in which events occurred, to be able to reason about result completeness.

Bounded streams have a defined start and end. Bounded streams can be processed by ingesting all data before performing any computations.

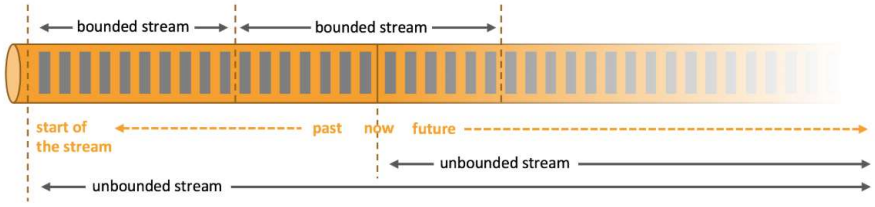

Figure 2.8: Apache Flink® - Bounded & Unbounded Data Streams.

Ordered ingestion is not required to process bounded streams because a bounded data set can always be sorted. Processing of bounded streams is also known as batch processing.

Apache Flink excels at processing unbounded and bounded data sets. Precise control of time and state enable Flink's runtime to run any kind of application on unbounded streams. Bounded streams are internally processed by algorithms and data structures that are specifically designed for fixed sized data sets, yielding excellent performance.

2.1.4 *Streams Processing: Storm*

Apache Storm [5] is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language, and is easy to use.

Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed.

The Apache Storm architecture is quite similar to that of Hadoop. However there are certain differences which can be better understood

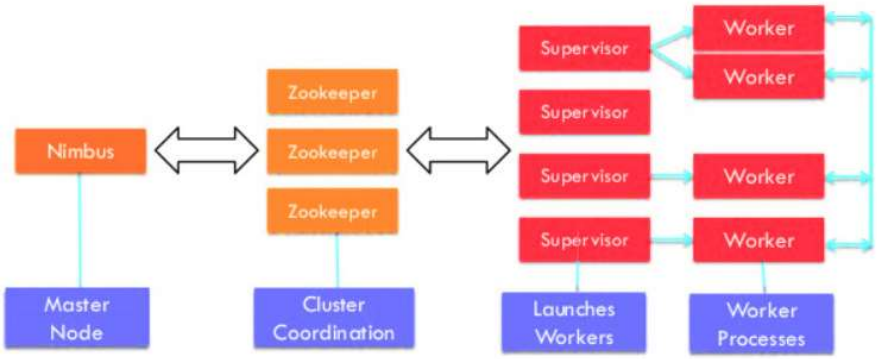

Figure 2.9: Apache Storm.

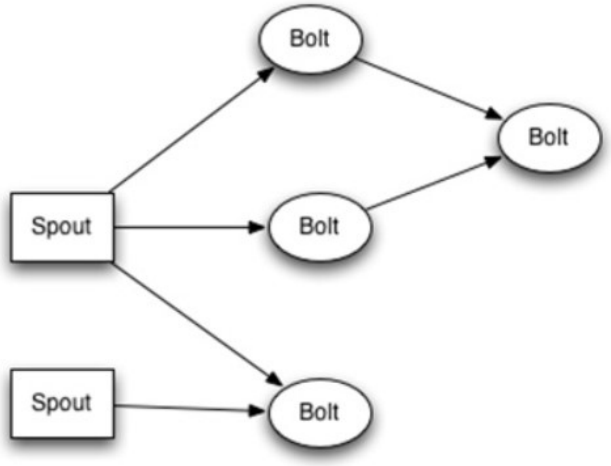

Figure 2.10: Apache Storm Components Abstraction.

by getting a closer look at its cluster in Figure 2.9: **Nodes** - There are two types of nodes in Storm cluster similar to Hadoop:

- Master node - The master node of Storm runs a daemon called 'Nimbus', which is similar to the 'Job Tracker' of Hadoop cluster. Nimbus is responsible for distributing codes, assigning tasks to machines and monitoring their performance.
- Worker node – Similar to the master node, the worker node also runs a daemon called 'Supervisor' which is able to run one or more worker processes on its node. Each supervisor works assigned by Nimbus and starts and stops the worker processes when required. Every worker process runs a specific set of topology which consists of worker processes working around machines. Since Apache Storm does not have the abilities to manage its cluster state, it depends on **Apache Zookeeper** for this purpose. Zookeeper facilitates communication between Nimbus and Supervisors with the help of message acknowledgements, processing status, etc.

Storm Components/Abstractions (see Figure 2.10) – There are basically four components which are responsible for performing the tasks:

- Topology – Storm Topology can be described as a network made of **spouts** and **bolts**. It can be compared to the Map and Reduce jobs of Hadoop. Spouts are the data stream source tasks and Bolts are the accrual processing tasks. Every node in the network consists of processing logic's and links to demonstrate the ways in which data will pass and the processes will be executed. Each time a topology is submitted to the storm cluster, Nimbus consults the supervisor nodes about the worker nodes.
- Stream – One of the basic abstractions of the storm architecture is stream which is an unbounded pipeline of tuples. A tuple can be defined as the fundamental component in the Storm cluster containing a named list of the values or elements.
- Spout – It is the entry point or the source of streams in the topology. It is responsible for getting in touch with the actual data source, receiving data continuously, transforming those data into actual stream of tuples and finally sending them to the bolts to be processed. two types of nodes in Storm cluster.
- Bolt - Bolts keep the logic required for processing. These are responsible for emitting the streams for processing by other bolts and saving or sending the data for storage. These are capable of running functions, filtering tuples, aggregating and joining streams, linking with database, etc.

2.2 RUNTIME MANAGEMENT OF BIG DATA APPLICATIONS

Big data applications requires system scalability, fault tolerance and availability [62].

Scalability means the ability to maintain an approximate linear relationship between the size of processed data and the amount of consumed resources.

Fault tolerance is a challenge, especially when systems are complex and involve many networked nodes. By means of hardware virtualization, cloud computing services satisfies all the requested requisites, and the *elasticity* and redundancy it provides also enable big data application high availability, scalability and fault tolerance.

Another very important feature of big data applications is the *Quality of Service* or *QoS*.

QoS definition for IT applications differ by application type. Interactive applications are usually assessed according to response time or throughput, and their fulfillment depends on the intensity and variety of the incoming requests.

Big data applications might require a single batch computation on a very large dataset, thus *QoS* must consider the execution of a single run. In this domain *QoS* is often called *deadline*, or the desired duration of the computation.

We have mentioned availability, fault tolerance and availability as fundamental requirements and *QoS* as a measure of the capability to meet a user-defined deadline when processing big data. Many factors influence the duration of an application execution, surely resource allocation greatly influences the duration.

The challenges introduced by big data require resilient, flexible and self-adapting software systems [49]. Hence, *autonomic system*s and *self-adaptation* has increasingly captured the attention of researchers [71]. These systems automatically react to changes in the environment, or in their own state, and change their behaviour to satisfy functional and non-functional requirements. Meeting requirements in complex and variable execution environments is a difficult task that can be tackled at design time or at runtime. At runtime the adaptation is very often obtained by using a well-known process called MAPE [45], a control loop composed of four phases: monitoring, analysis, planning and execution.

One of the challenges that modern software systems face is the provisioning and optimization of resources to meet a varying demand, generated by are increasingly common phenomena like fluctuating workloads, unpredictable peaks of traffic and unexpected changes. Service providers cannot disregards these factors if they want to cope with the challenge of satisfying functional and non-functional requirements, usually defined in SLAs (Service Level Agreements). Hence the need arises for an automatic adjustment of system resources

allocation to avoid resource saturation and unresponsiveness, users dissatisfaction and unnecessary costs. This paradigm is called elastic resource provisioning [30, 75, 64, 39].

Many approaches about elastic systems and dynamic resource allocation were proposed both in the industry and in academia. In modern technology elasticity is often enabled by cloud computing that gives to an application a theoretical infinite degree of scalability. However considering only resources is rather restrictive because of the many factors that impact application during their runtime life-cycle. In fact Dustdar et al. [30] argue that elastic computing should be designed by considering three dimensions: quality, resources and cost. Quality elasticity considers how quality is affected by a change in resource availability. Instead cost elasticity measures how resource provision is affected when a change in cost happens.

Cloud computing services provide the needed level of fault tolerance and availability required by big data applications. In the remainder of this chapter we present an overview of popular big data frameworks that can leverage cloud computing solutions and how they address elastic resource allocation to satisfy functional and non-functional application requirements. The resulting scenario represent the base for our work, where we will consider quality elasticity and not cost elasticity aspect.

This thesis shows how the application of lightweight symbolic execution tecniques to deadline-based *QoS* constrained multi-DAG big data applications helps reduce the number of deadline violations and allocate resources more efficiently.

2.3 ELASTIC RESOURCE PROVISIONING

The word *elasticity* comes from physics and is defined as the ability of an object or material to resume its normal shape after being stretched or compressed. In computing, elasticity has a similar meaning and is used to characterize *autonomic systems*. Herbst et al. [39] defines elastic provisioning as reported below.

> ***Elasticity*** *is the capability of a system to adapt to workload changes by provisioning or de-provisioning resources automatically such that at each point in time the available resources match the current demand as closely as possible.*

A system is in an *under provisioned* state if it allocates less resources than required by the current demand; it is in an *over provisioned* state if allocates more resources than required. Moreover, elasticity is determined by four attributes:

- *Autonomic Scaling*: the adaptation process used to control the system.

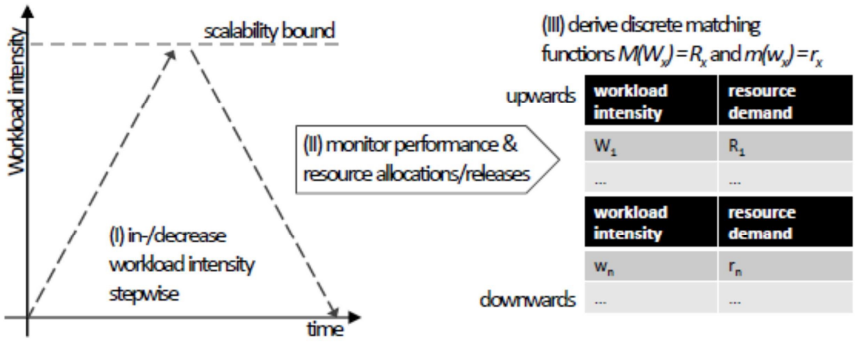

Figure 2.11: Elasticity Matching Function derivation.

- *Elasticity Dimensions*: the set of scaled resources in the adaptation process.
- *Resource Scaling Units*: the minimum amount of allocable resources to each dimension.
- *Scalability Bounds*: the lower and the upper bound on the amount of resources that can be allocated to each dimension.

Additionally, two aspects must be considered in evaluating the elasticity degree of a system: speed and precision.

Speed of scaling up/down: the time it takes to switch from an underprovisioned/overprovisioned state to an optimal or overprovisioned/underprovisioned state respectively.

Precision of scaling: the absolute deviation of the current amount of allocated resources from the actual resource demand.

Scalability and efficiency are terms related to elasticity, nevertheless they differ by the following aspects:

Scalability, although is a prerequisite for elasticity, it does not consider the temporal aspects (how fast and how often) and the granularity of the adaptation actions.

Efficiency, contrary to elasticity, it takes in account all the types of resources employed to accomplish a certain amount of work, not only the resources scaled by the adaptation actions.

Elasticity reflects the (theoretical) infinite upper bound on resource scalability in cloud computing and the frictionless resource renting model. Nevertheless elasticity is not just a synonym of resource management. Elasticity is also related to trade off between cost and qual-

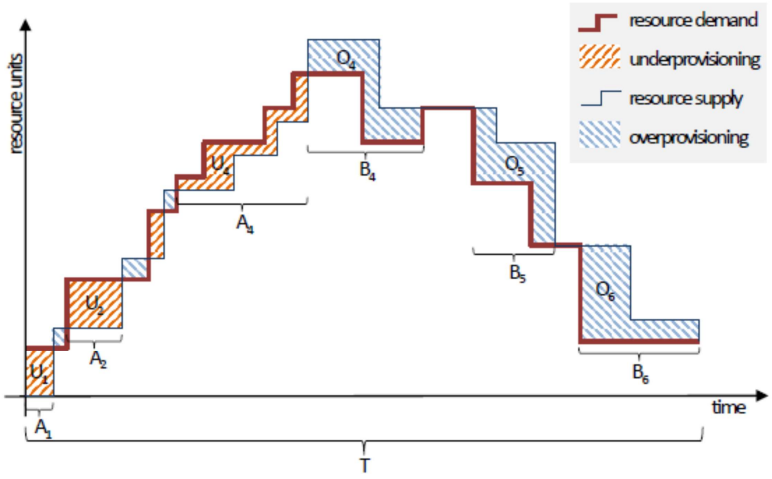

Figure 2.12: Resource Provisioning Chart.

ity [30]. Cost elasticity describes how resources are managed in response to cost changes, while quality elasticity measures how responsive is the quality to changes in resource utilization.

A *matching function* $m(w) - r$ is a system specific function that gives the minimum quantity of resources r for any given resource type needed to meet the system's performance requirements at a certain workload intensity. A matching function is required for both up and down scaling directions.

The matching functions can be determined based on measurements, as illustrated in Figure 2.11, by increasing the workload intensity w stepwise, and measuring the resource consumption r, while tracking resource allocation changes. The process is then repeated for decreasing w. After each change in the workload intensity, the system should be given enough time to adapt its resource allocations reaching a stable state for the respective workload intensity. As a rule of thumb, at least two times the technical resource provisioning time is recommended to use as a minimum. As a result of this step, a system specific table is derived that maps workload intensity levels to resource demands, and the other way round, for both scaling directions within the scaling bounds.

An example of how *speed* and *precision* affect resource provisioning is shown in Figure 2.12.

2.4 SPARK RESOURCE PROVISIONING

It's important to remember that the cluster manager is responsible for starting executor processes and determine where and when they will

run. Using Spark's embedded cluster manager might be a problem in terms of resource utilization when we want to execute different distributed applications at the same time. Using a single cluster manager for different distributed applications has the advantage of providing a global view on which applications are running and which we want to execute inside the cluster. Without a single cluster manager, we can have two main approaches in order to perform resource sharing and allocation:

- allowing every application to allocate all the resources in the cluster at the same time, this leads to an unfair situation of resource contention
- splitting the resource pool into smaller pools, one per application.

In this way we will avoid resource contention but we will have a less efficient utilization of the resources, because some of the applications might request more resources than the ones in the pool, meanwhile some others are using less resources than the allocable ones in order to execute. A more dynamic way of allocating resources will led to a better resource utilization. Spark natively support executing on top of Apache Hadoop YARN and Apache Mesos cluster managers. Spark supports the dynamic allocation of executors, also known as elastic scaling, this feature allows to add and remove Spark executors in a dynamic way in order to match the workload. In traditional static allocation, a Spark application would allocate CPU and memory upon starting the execution, disregarding how much resources will effectively use later on. With dynamic allocation instead it is possible to allocate as much resources as they are necessary, in order to avoid wasting them. The number of running executors is scaled up and down according to the workload, in particular idling executors are removed and when there are tasks waiting to be executed and new executors are launched. Dynamic allocation can be activate in Spark settings and should be used together with the External Shuffle Service, in this way data that have been manipulated from the executor is still available after the removal of the executor. Dynamic allocation has two different policy for scaling the executors:

- Scale Up Policy: new executor are requested when there are pending tasks, the number of executors is increased exponentially because they start slow and so the application might need a slightly higher number of them
- Scale Down Policy: idling executors are removed after a certain amount of time, this amount of time can be configured

In order for dynamic allocation to work, we must configure it, by setting the initial number of executors that are created when application starts and the minimum and maximum number of executors

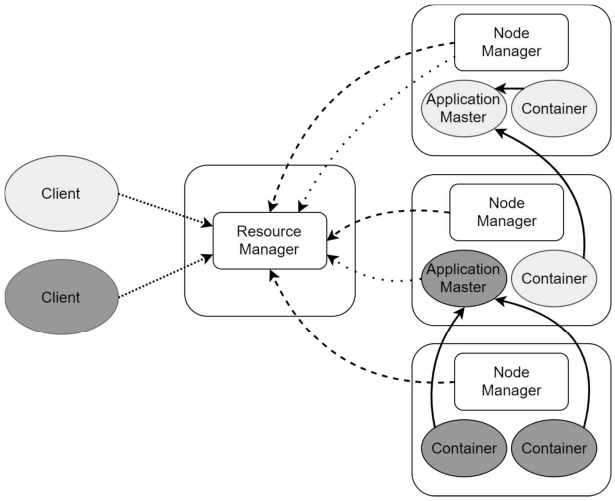

Figure 2.13: Apache Hadoop YARN Architecture.

that can be reached when scaling down and up respectively. Dynamic allocation is available on all cluster managers currently supported by Spark, even in Standalone mode.

2.4.1 *Apache Hadoop Yarn*

Apache Hadoop YARN (Yet Another Resource Negotiator) is the next generation of Hadoop's compute platform[68]. The idea is to split the functionality of resource management and job scheduling and monitoring. This is achieved by having two different kind of daemons running, one global Resource Manager (RM) and a per-application Application Master (AM). Resource Manager (RM) and Node Manager (NM) form the data computation framework (Figure 1.4). RM is the authority that manages resources among all the applications that are running in the system, meanwhile NM is the per-machine daemon who is responsible for managing containers, monitoring and reporting. The perapplication AM has the goal of negotiating resources with RM and working with NM in order to execute and monitor tasks. Resource Manager (RM) is composed by two components: Scheduler and Applications Manager. The Scheduler is responsible for allocating resources to the various applications that are running, by taking into account constraints about capacity, queues, etc. It is a real scheduler in the sense that it does not perform monitoring or tracking of the application state. Moreover, it does not offer any guarantee that a failed application will be reexecuted after an application or hardware failure. The Scheduler performs the allocation according to the resources that are requested by an application; this is based on the abstract notion

of container which has elements as memory, CPU cores, disk and network bandwidth. There are pluggable policy that determine the repartition of resources among the different applications, for example we have the Capacity Scheduler, designed for multi-tenant clusters, and the Fair Scheduler, that shares cluster resources fairly. The Applications Manager is responsible of accepting the submission of a job, it negotiates the first container that will execute the AM and it offers a service that can be used to restart the AM in case of failure. The per-application AM has the goal of negotiating the needed containers from the Scheduler, track their status and monitor their progress. The RM keeps a global model of the cluster state and thanks to the resource requirements reported by the running applications, it makes possible to enforce a global scheduling, but it is required to have an accurate understanding of the applications' resource requirements. In response to AM requests, the RM generates containers together with tokens that grant access to resources. An extension of the protocol allows the RM to ask back resources from applications, for example when cluster resources become scarce. Application Master (AM) is the process that coordinates the execution of an application inside the cluster. It is important to remember that AM itself is run in the cluster, just like any other container. Periodically, an heartbeat is sent to the RM in order to confirm its liveliness and to update the Scheduler about its resource requests. After having modeled the application requirements, the AM encodes its preferences and constraints inside the heartbeat message. This information are stored in the form of Resource Request, containing the desired number of containers (e.g., 100 container), the resources of each container (e.g., <2 CPU, 2 GB>), the locality preferences and the priority of this resource request with respect to the other ones of this application. When a container lease is received, the AM can choose to modify its execution plan in order to take into account the abundance or scarcity of resources. Node Manager (NM) is the worker daemon in YARN, its purpose is to authenticate container lease, manage dependencies, monitoring the execution of containers and offer them a set of services. After having registered with the RM, the NM sends heartbeats in order to communicate its status and receives instructions from the RM. All the containers are described by a container launch context (CLC), that keeps track of all the environment variables, the dependencies, the security tokens, but also of the payloads needed by NM services and the commands that are needed to launch the process inside the container. After having validated the authenticity of the container lease, the NM configures the container with the specified resource constraints and initializes a monitoring subsystem. In order to launch the container, dependencies are copied into local storage. NM also has the duty of killing container upon a request from RM or AM, for example when a tenant is being evicted or when an application completes. Whenever a container exits,

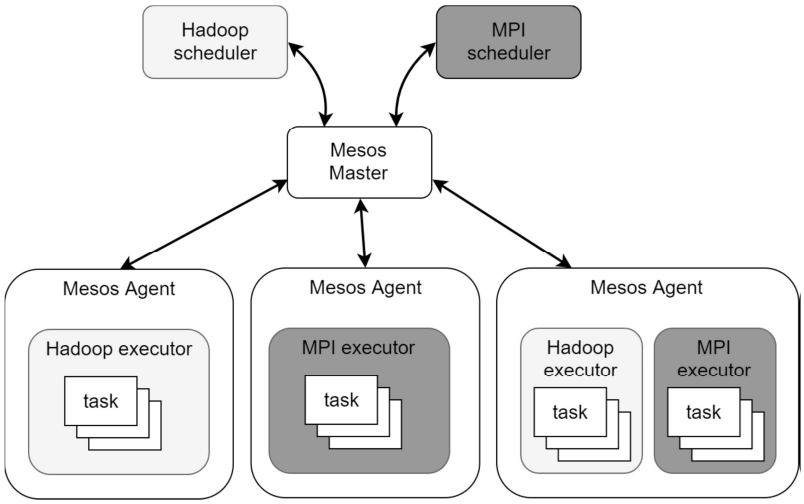

Figure 2.14: Apache Mesos Architecture.

NM needs to clean the working directory. When an application ends, all the resources held by its container on all nodes are released. NM periodically checks the state of the physical machine and informs the RM of a possible unhealthy state.

2.4.2 *Apache Mesos*

Apache Mesos is an open-source project used to manage computer clusters. The purpose of Mesos is to share cluster between different computing frameworks, such as Apache Hadoop or Message Passing Interface (MPI). The sharing increments the utilization of the cluster and prevents per-framework data replication. Mesos shares resources in a fine-grained way, allowing to achieve data locality. It presents a scheduling mechanism on two layer called resource offers. Mesos decides how many resources to offer to each of the running frameworks, meanwhile they decide how many resources to accept and which computation to execute on the granted resources. New cluster computing frameworks continue to emerge, it is clear that finding a framework that is optimal for all type of application is almost impossible. We expect that organization would like to use different frameworks inside the same cluster, picking the best one according to the kind of application that they are going to execute. Two classic solution are: i) statically partitioning the cluster and executing one framework per partition; ii) allocate a set of VMs to each of the frameworks. Unluckily these solution do not achieve high utilization and efficient data sharing. The main problem is the different allocation granularity of these solutions and the one of the existing frameworks, for example Hadoop employs

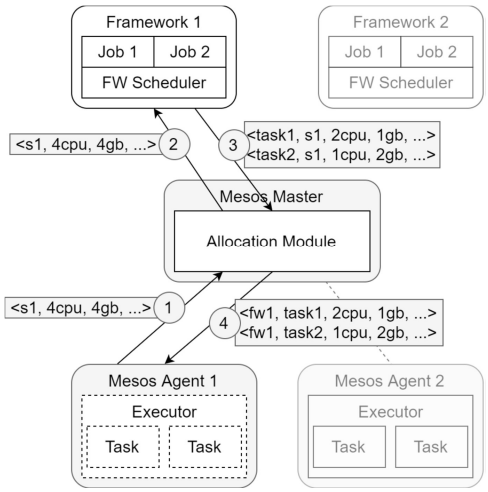

Figure 2.15: Apache Mesos Resource Offer example. 1) Mesos Agent 1 reports free resources to the Allocation Module; 2) Allocation Module offers resources to Framework 1 scheduler; 3) Framework 1 scheduler accepts resources and assign tasks; 4) Allocation Module launches tasks on the executor running in Mesos Agent 1.

a fine grained resource sharing model, where nodes are divided into slots and each job is composed by short tasks that match the slots. The presence of short tasks allows us to achieve high utilization, as jobs can rapidly scale when new nodes are available. But it is not possible to achieve fine grained sharing across frameworks, because they have been developed in an independent way, and thus it is difficult to efficiently share the cluster among different frameworks. Mesos delegates the control over the scheduling to the different frameworks. In this way it is possible to have the abstraction of the resource offers, that encapsulate a bundle of resources that the framework can allocate on a node in order to execute a task. Mesos decides how many resources to offer to each framework, this is based on policies, and the framework decides which resources to accept and which tasks to execute on them. Even though this approach does not lead to a globally optimum scheduling, it has been proved that it performs particularly well in practice, allowing the frameworks to obtain near perfect data locality. Mesos provides other benefits to its users, for example the possibility of running different instances of the same framework or even different versions. Mesos is composed by a master process that manages slave daemons running on each cluster node and frameworks that run tasks on these slaves, as we can see from figure 2.14 . Master implements fine-grained sharing across frameworks using resource offers. Every

resource offer is a list of free resources on the different slave nodes. The master decides how many resources to offer to each framework, according to some policy such as fairness or priority.

Apache Mesos Resource Offer example. 1) Mesos Agent 1 reports free resources to the Allocation Module; 2) Allocation Module offers resources to Framework 1 scheduler; 3) Framework 1 scheduler accepts resources and assign tasks; 4) Allocation Module launches tasks on the executor running in Mesos Agent 1.

Every framework that is running on Mesos is composed by two components: a scheduler, that registers with the master in order to obtain the resource offers, and an executor process that is launched on the slave node in order to execute framework's tasks. While the master chooses how many resources to offer, the scheduler chooses which resources to use among those offered. When an offer is accepted, the scheduler sends to the master the description of the tasks that should be executed. The resource offer process is repeated every time tasks are finished and when there are new free resources. In order to maintain a light interface, Mesos does not ask the frameworks to specify their resource requirements or constraint, instead it gives them the possibility of refusing offered resources. Mesos allows frameworks to set up a set of filters, in the form of boolean predicates, specifying the conditions on which the framework will always refuse a proposal (e.g., providing a whitelist of nodes it can run on). In Figure 1.6 we have an example of resource offer process. 1. Agent 1 reports to master that it has 4 CPUs and 4 GB of memory free. Master invokes its allocation module policy, which tells that framework 1 should be offered all the resources. 2. Master sends a resource offer describing the resources available on agent 1 to framework 1.

2.4.3 *Spark on Yarn*

subsec:sparkOnYarn Support for running Spark on YARN was added to Spark in version 0.6.0 and has been improved in subsequent releases [61]. When running on YARN, each Spark executor is run inside a YARN container. Spark supports two different modes to run on YARN, the Yarn-cluster and Yarn-client mode. In client mode, as shown in figure 2.16, the driver program is run inside the client process. In this way, the Application Master (AM) that is run in a YARN container is used only to request resources to the Resource Manager (RM). This mode is useful for interactive applications and for debugging purposes, since you can see applications' output immediately on the client side process. If the client disconnects from the cluster, the Spark application will terminate, this is due to the fact that the driver process resides on the client. In cluster mode instead, as shown in Figure 1.10, Spark driver program is run inside the AM process managed by YARN. After initializing the application, client can disconnect from

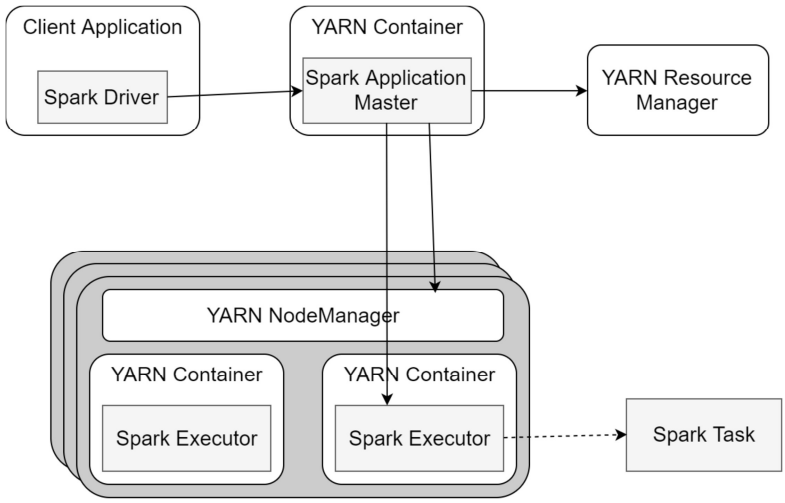

Figure 2.16: Spark on YARN Client Mode.

the cluster and reconnect later on. This mode makes sense when using Spark on YARN in production jobs. Running on top of YARN cluster manager has some benefits. First of all YARN allows to dynamically share the cluster resources between the different frameworks that are running together. For example we can run MapReduce jobs after running Spark jobs without the need of changing YARN configurations. Moreover, YARN supports for categorizing, isolating and prioritizing workloads and employs security policies, in this way Spark can use secure authentication between its processes.

When running on YARN, Spark executors and driver program use about 6-10% more memory with respect to the standalone execution, this is due to the fact that this extra amount of off-heap memory is allocated in order to take into account YARN overheads.

2.4.4 *Spark on Mesos*

subsec:sparkOnMesos Support for running Spark on Mesos was added to Spark in version 1.5. Spark on Mesos can be executed in two different modes: coarse-grained and fine-grained [60]. In coarse-grained mode, as shown in figure 2.18, each Spark application is submitted to Mesos master as a framework and Mesos slaves will run tasks for the Spark framework that are Spark executors. Mesos tasks are launched for each Spark executor and those Mesos tasks stay alive during the lifetime of the application unless we are using dynamic allocation or the executor is killed for various reasons. The advantage of coarse-grained mode is in a much lower task startup overhead, with respect to the other mode, and so it is good for interactive sessions.

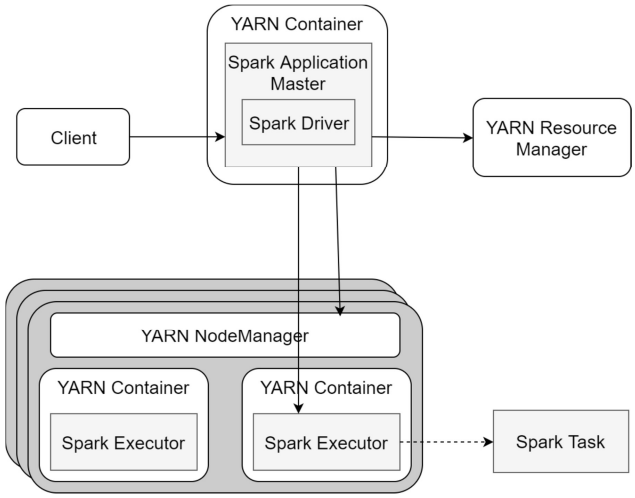

Figure 2.17: Spark on YARN Cluster Mode.

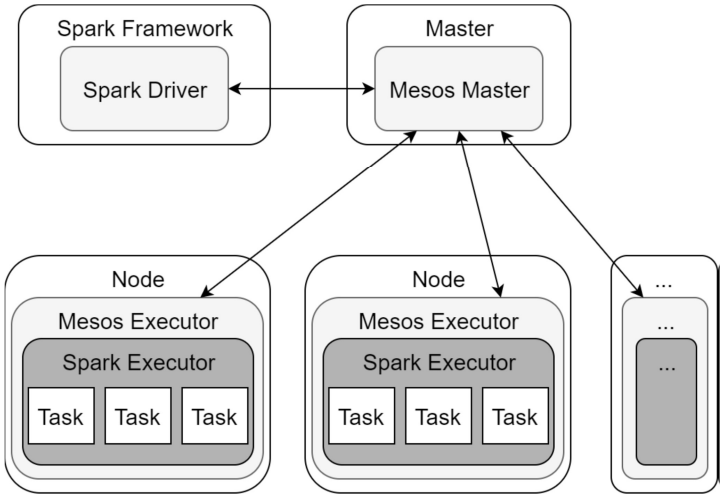

Figure 2.18: Spark on Mesos Coarse Grained Mode.

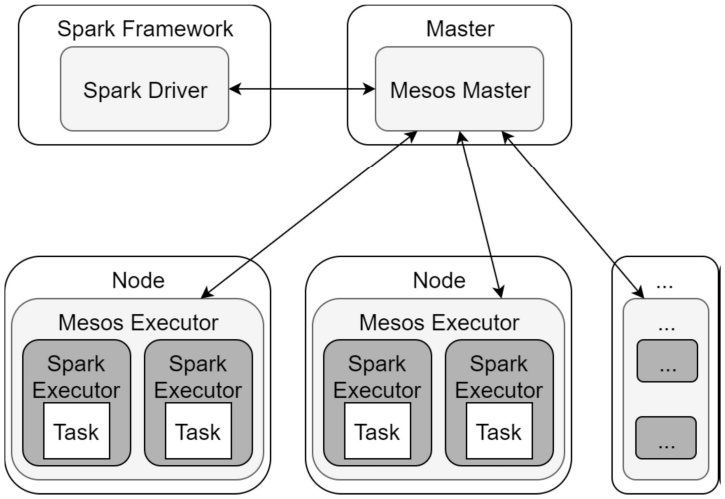

Figure 2.19: Spark on Mesos Fine Grained Mode.

The drawback is that we are reserving Mesos resources for the complete duration of the application, unless dynamic allocation is active. Dynamic allocation allows to add and remove executors based on load: i) kill executor when they are idle, ii) add executors when tasks queue up in the scheduler. To use dynamic allocation it is required that the external shuffle service is running on each node. In fine-grained mode, shown in figure 2.19, Mesos tasks are launched for each Spark task, and those tasks die as soon as Spark tasks are done. This mode has too much overhead in case that Spark has too many tasks, for example if Spark application has 10,000 tasks, then Spark needs to be installed 10,000 times on Mesos agents. Because of this huge overhead, fine-grained mode has been deprecated since Spark version 2.0.0. This mode allows multiple instances of Spark to share cores at a very fine granularity, but it comes with an additional overhead in launching each task. Thus this mode is inappropriate for low-latency requirements like interactive queries or serving web requests, instead it is fine for batch and relatively static streaming. Similarly to what happens on YARN, it is possible to run spark in Mesos-client or Mesos-cluster mode. In client mode, the driver process is executed in the client machine that submits the job, so it is required that it stays connected to the cluster for the entire time of the application execution. In cluster mode instead, the driver program is run on a machine of the cluster.

2.5 VIRTUALIZATION AND CONTAINERIZATION

Virtualization refers to creating the virtual version of something, included hardware components, storage devices and computer networks.

Virtualization is born in 1960's, as a way to logically partition the system resources offered by a mainframe computer between many different applications. From this point, the meaning of the word has been widely extended. Virtualization is a technology that allows creating multiple simulated environments or dedicated resources from a single unique physical hardware system. An hypervisor is a software that can directly connect to the hardware, with the purpose of splitting the unique physical system into separated environments, different from each other and secure, known as virtual machines (VM's). These VM's rely on the hypervisor ability to separate the hardware resources and distribute them in a proper way. The original physical machine equipped with the hypervisor is called host, meanwhile the VM's are called guests. These guests use the computation resources, such as CPU, memory and storage, as a set of resources that are easily re-allocatable. The operators can control the virtual instances of CPU, memory, storage and other resources, so that the guests can receive all the resources they need to execute their task. The words host and guest are used to distinguish the software that runs on the physical hardware from the software that is running on the virtual machines. Hardware virtualization or platform virtualization refer to the creation of a VM that acts like a real computer with an OS. The software that is run in this VM is separated from the underlying hardware. This allows us to run particular configurations, for example we run a computer with a Windows OS that hosts a VM with Linux as guest OS. There are at least two different hardware virtualization types:

- full virtualization: it completely simulates the hardware in order to allow the software, typically a guest OS, to be run without the need of modifications
- paravirtualization: the hardware environment is not simulated, anyway guest programs are run in isolated domains, as if they were run in completely separated systems. Guest programs need to be modified in a specified way in order to run in this kind of environment.

In figure 2.20, we can see the differences between the two kind of virtualization. In paravirtualization, the VM presents a different interface compared to the one of a physical machine. This requires modification in the guest OS in order to allow its execution inside the VM. The hypervisor exposes a set of APIs that the guest OS must use to execute privileged instructions. Calls to these particular functions are often defined as Hypercalls. In full virtualization instead, VM have the same interface as the physical ones. Ideally, the guest OS would not be able to determine if it is being run on a physical or virtual machine. The great advantage of full virtualization is that we do not need to modify the OS. In this way the hypervisor can adopt a trap system to execute privileged instructions. We can improve the efficiency of

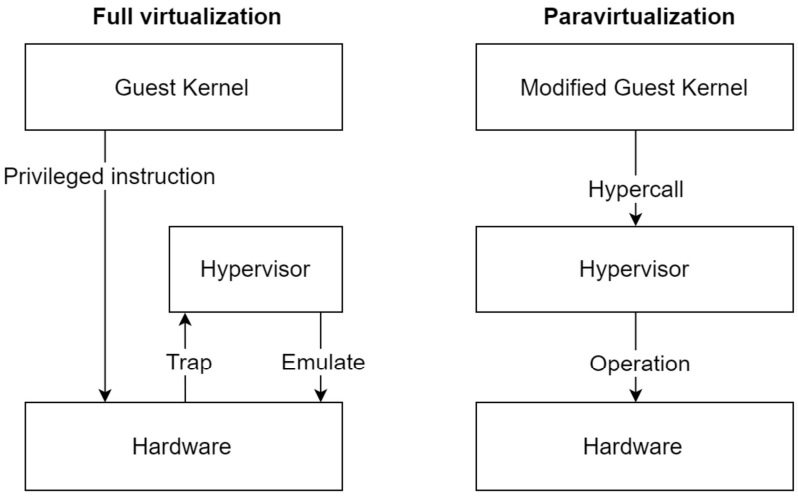

Figure 2.20: Full virtualization and paravirtualization.

the virtualization by using hardware assisted virtualization, in particular we can decide to use CPU's that provide efficient support for virtualizing on hardware, but also other kind of hardware components that can improve the performance of the guest environments. Hardware virtualization can be seen as a trend of the enterprise IT that includes autonomic computing, a scenario in which the different environments are able to manage themselves based on the detected level of activity, and utility computing, where the processing power is seen as a utility that users pay only when needed. The purpose of virtualization is to centralize the administrative tasks, offering scalability and good resource utilization. With virtualization, different OS can run in parallel on a single CPU. This parallelism reduces overhead cost in a way different from multitasking, where different programs are executed in parallel on the same OS. Thanks to virtualization, an enterprise IT can better handle updates and rapid changes in OS and applications, with little impact on users. Virtualization allows organizations to have better efficiency and availability of resources and applications. It is important to remember that hardware emulation is a complete different thing from hardware virtualization, in particular with emulation we have a piece of hardware that imitates another piece of hardware. In virtualization instead a hypervisor, which is a piece of software, mimics a piece of hardware or even an entire computer. Moreover, a hypervisor is not an emulator, even though both are software programs that mimic hardware, their domain of use is different. Containerization is a OS-level virtualization technique that allows deploying and executing distributed applications without the need of launching an entire VM for each of the applications (fig-

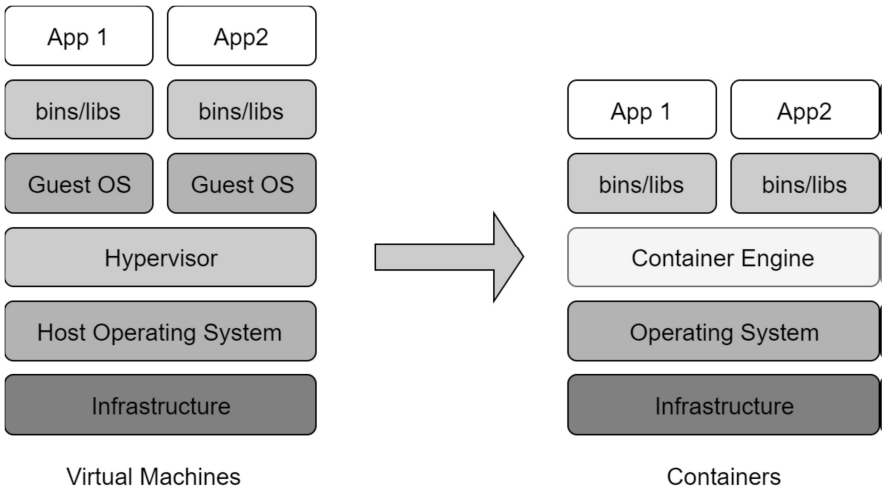

Figure 2.21: The difference in architecture between virtual machines and containers.

ure 2.21). These multiple isolated systems are called containers. They are executed on top of a single host controller and access a single kernel. Since containers share the same OS kernel of the host, they can be a lot more efficient than a VM, that instead needs a separate instance of the OS. Containers own all the different components that are needed in order to execute the desired software, such as files, environment variables and libraries. The host OS controls the access of the container to the physical resources, such as CPU and memory, in order to prevent a single container from occupying the entire resources offered by the host. The main advantages of containerization come from efficiency in terms of memory, CPU and storage, when compared to traditional hardware virtualization. Since containers do not have the same overhead of VM, in particular we do not need different instances of the OS and is possible to support more containers on the same infrastructure. Containerization offers better performances since there is a single OS that takes care of all the hardware calls. A special advantage of the containers is the fact that they can be created much faster than instances based on an hypervisor. This allows to have a more agile environment and allows the creation of new approaches to virtualization, such as microservices and continuous integration and delivery of services. Potential disadvantages of containerization are the absence of isolation from the host OS. Since containers share the same host OS, a potential security threat might easily gain access to the entire system. This does not happen when using virtualization based on a hypervisor, since in this case the only compromised component would be the VM. A way to circumvent this problem, is the creation

of containers inside an OS run from a VM. This prevents the security breach at container level from letting the attacker gain access to the OS of the physical host. Another little disadvantage of containerization is that containers must execute the same OS as the base OS, meanwhile instances based on an hypervisor are allowed to execute different OSs. Because of this, a container that is running on a Linux host, can neither execute an instance of Windows OS nor a Windows designed application. Containerization has gained more and more relevance thanks to the wide spread of the open source software Docker, that has developed aenhanced portability to the containers, allowing them to be moved from different systems that share the same kind of host OS without the need to change even a single line of code. In particular, with Docker container there are no environment variables that must be set on the guest OS or library dependencies that need to be managed.

2.5.1 *Docker*

subsec:docker Docker is an open source project that automatizes the deployment of applications inside software containers, giving a further abstraction thanks to OS level virtualization provided by Linux OS. Docker uses isolation functionalities provided by Linux kernel, such as cgroups and namespaces [29] in order to allow the coexistence of independent containers on the same Linux instance, avoiding the installation and maintenance of a VM. Linux kernel namespaces isolate what the application ca see of the operating environment, including process tree, network, user ids and mounted file system. Cgroups instead provide resource isolation, including CPU, memory, I/O devices and network. Docker implements a high level API in order to manage containers that execute in isolated environments. Since it uses Linux kernel functionalities, a Docker container, compared to a VM, does not includes a separated OS. Instead, it uses the kernel functionalities and exploits resource isolation and separated namespace in order to separate isolate what each application can see of the underlying OS. Docker can access Linux kernel virtualization functionalities using different ways, for example directly using libcontainer or indirectly using libvirt, Linux Containers (LXC) or systemd-nspawn (figure 2.22). Using containers, resources can be isolated, services can be limited and processes can be started in a way that each of them has a private perspective of the OS, with their own identifier, file system and network interface. More container can share the same kernel, but each of them can be forced to use a different amount of resources such as CPU, memory and I/O. By using Docker we can create and manage container in a way that simplifies the creation of distributed systems, allowing different applications and processes to work in an autonomous way on the same physical machine or on different virtual machines. This allows us to deploy new nodes only when necessary,

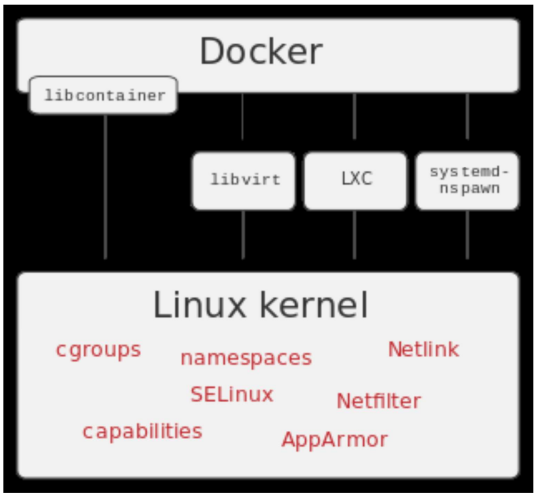

Figure 2.22: Docker can use different interfaces in order to access Linux kernel virtualization functionalities.

in order to follow an evolution style that is similar to the platform as a service one.

2.6 SYMBOLIC EXECUTION

In computer science, the term symbolic execution refers to a software program analysis technique used to determine which data inputs cause the execution of each part of a program. It was introduced in the mid '70s [46, 17, 47, 40] mainly to test whether a software program could violate certain properties, e.g. that no divisions by zero are performed, no NULL pointers get dereferenced, no access to protected data can happen by unathorized users, etc. In general, it's not possible to decide every possible program property by means of automated checks, for example we cannot predict the target of an indirect jump. In practice, approximate and heuristic-based analyses are used in many cases, even in the field of mission-critical and security applications.

Software testing is performed to check that certain program properties hold for any possible usage scenario. A viable approach would be to test the program using a wide range of different, possibly random inputs. As the problem may occur only for very specific input values, we need to automate the exploration of the domain of the possible input data.

With symbolic execution many possible execution paths are explored in parallel, without necessarily requiring concrete inputs. The idea is to replace the fully specified input data with symbols, that

are their abstract representation, devolving to constraint solvers the construction of actual instances that would cause property violations.

The symbolic execution intepreter walks through all the steps of the program, associating symbolic values to inputs rather than obtaining their actual values, building expressions in terms of those symbols and program variables, and constraints in terms of the symbols corresponding to the possible outcomes of each conditional branch.

When a program is run with a specific set of input data (a concrete execution), a single control flow path is explored. Hence, concrete executions can only under-approximate the analysis of the property of interest. With symbolic execution, multiple paths that a program could take under different inputs can be simultaneously explored. This means that a sound analyses can be done, giving stronger guarantees about the checked property [11]. When a program runs with *symbolic* – rather than concrete – input values, the execution is performed by a *symbolic execution engine*, which builds and updates a structure to hold (i) a first-order Boolean *formula* describing the conditions satisfied by all the traversed branches along that path, and (ii) a *symbolic memory store* mapping variables to symbolic expressions or values, for each path traversed by the control flow. Execution of a branch updates the formula, while assignments update the symbolic store. Finally, a *model checker*, commonly based on a *satisfiability modulo theories* (SMT) solver [14], is used to verify if the property is violated somewhere along each explored path and if the path itself is concretely feasible, i.e., if any assignment of concrete values to the program's symbolic arguments exists that satisfies its formula [11].

Symbolic execution techniques have been emphasized to a wide audience following the DARPA announcement in 2013 at the Cyber Grand Challenge, a two-year competition pursuing the creation of automatic systems for vulnerability detection, exploitation, and patching in near real-time [65]. More important, symbolic execution based engines have been running 24/7 in the testing process of many Microsoft applications since 2008, discovering nearly 30% of all the flaws discovered by file fuzz testing during the development of Windows 7, which other program analyses and blackbox testing techniques missed [36].

AN INTRODUCTORY EXAMPLE With reference to the C code in Figure 2.23, let's say we want to discover which of the 2^{32} possible `4-byte` inputs make the `assert` at line 8 of function `foo` fail. If we address the problem by running concretely the function `foo` on randomly generated inputs, we will unlikely pick up exactly the assert-failing inputs. Symbolic execution go beyond this limitation by reasoning on *classes of inputs*, rather than single input values, thanks to the evaluation of the code using symbols for its inputs, instead of concrete values,

```
1.  void foo(int a, int b) {
2.     int x = 1, y = 0;
3.     if (a != 0) {
4.        y = 3+x;
5.        if (b == 0)
6.           x = 2*(a+b);
7.     }
8.     assert(x-y != 0);
9.  }
```

Figure 2.23: Example: which values of `a` and `b` make the `assert` fail?

Going further into details, a symbol α_i is associated to each value that cannot be resolved by a static analysis of the code, e.g. an actual parameter of a function or data read from an input stream. At any time, the symbolic execution engine maintains a state $(stmt,\ \sigma,\ \pi)$ where:

- $stmt$ is the next statement to evaluate. At this time being, we assume that $stmt$ can be an assignment, a conditional branch, or a jump.
- σ is a *symbolic store* that associates program variables with either expressions over concrete values or symbolic values α_i.
- π denotes the *path constraints*, i.e., it is a formula expressing a set of assumptions on the symbols α_i as a result of branches taken in the execution to reach $stmt$. At the start of the analysis, $\pi = true$.

Depending on $stmt$, the symbolic engine changes the state as follows:

- The evaluation of an assignment $x = e$ updates the symbolic store σ by associating x with a new symbolic expression e_s. We express this association with $x \mapsto e_s$, where e_s is obtained by evaluating e in the context of the current execution state and can be any expression involving unary or binary operators over symbols and concrete values.
- The evaluation of a conditional branch $\texttt{if}\ e\ \texttt{then}\ s_{true}\ \texttt{else}\ s_{false}$ affects the path constraints π. The symbolic execution is forked by creating two execution states with path constraints π_{true} and π_{false}, respectively, which correspond to the two branches: $\pi_{true} = \pi \wedge e_s$ and $\pi_{false} = \pi \wedge \neg e_s$, where e_s is a symbolic expression obtained by evaluating e. Symbolic execution independently proceeds on both states.
- The evaluation of a jump goto s updates the execution state by advancing the symbolic execution to statement s.

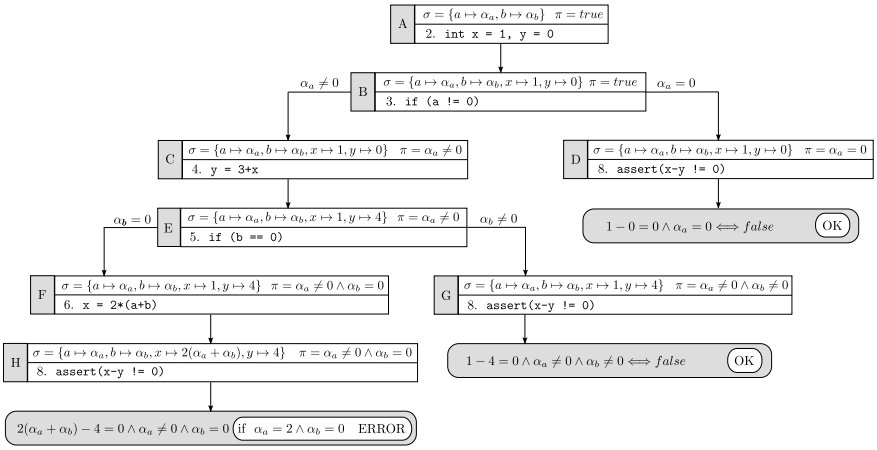

Figure 2.24: Symbolic execution tree of function `foo` given in Figure 2.23. Each execution state, labeled with an upper case letter, shows the statement to be executed, the symbolic store σ, and the path constraints π. Leaves are evaluated against the condition in the assert statement.

Figure 2.24 shows a symbolic execution of function `foo`, that can be adequately represented as a tree. In the initial state (execution state A) the path constraints are `true` and input arguments `a` and `b` are associated with symbolic values. After local variables initialization `x` and `y` at line 2, the symbolic store is updated by associating `x` and `y` with concrete values 1 and 0, respectively (execution state B). A conditional branch is met in line 3 and the execution is forked:according to which branch is taken, a different statement is evaluated and different assumptions are made on symbol α_a (execution states C and D). In the branch corresponding to $\alpha_a \neq 0$, variable `y` is assigned to `x` + 3, obtaining $y \mapsto 4$ in state E because $x \mapsto 1$ in state C. Arithmetic expression evaluation, generally, change only the symbolic values. When the assert at line 8 is reached by fully expanding every execution state on all branches, we can check which input values for parameters `a` and `b` can make the assert fail. By analyzing execution states $\{D, G, H\}$, we can conclude that only H can make x-y = 0 true. The path constraints for H at this point implicitly define the set of inputs that are unsafe for `foo`. In particular, any input values such that:

$$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$$

will make assert fail. An instance of unsafe input parameters can be eventually determined by invoking an *SMT solver* [14] to solve the path constraints, which in this example would yield $a = 2$ and $b = 0$.

2.6.0.1 *Challenges of Symbolic Execution*

The example shown in Section 2.6 represents a case where symbolic execution can derive *all* the inputs that make the assert fail, by exploring all the possible execution states. With regards to the underpinning

theory, exhaustive symbolic execution represents a *sound* and *complete* methodology for any decidable analysis. Soundness (no false negatives) means that all possible unsafe inputs are guaranteed to be found, while completeness (no false positives) means that input values deemed unsafe are actually unsafe. Exhaustive symbolic execution cannot be easily scalable beyond small-sized applications. In many practical cases, a trade-off between soundness and performance approach is used.

Symbolic execution applied to real world applications faces many challenges, deriving from the increased complexity of those applications compared to the simplicity of our example. These challenges are pointed out by the following questions and observations:

- *Pointers, arrays, or other complex objects*: how are they handled by the symbolic engine? Code manipulating pointers and data structures may identify memory addresses that are described by symbolic expressions, in addition to symbolic stored data.
- *Interactions across the software stack*: how are they handled by the engine? System code/library calls can have side effects, e.g. file creations, calls back to user code or callback functions that could affect the execution at a later stage of the computation and must be taken in account. Evaluating all possible interaction outcomes may be unfeasible.
- *State space explosion*: how does symbolic execution deal with path explosion? The presence of iterative constructs in the language (iteration loops) can dramatically increase the number of execution states, up to the point where it becomes unfeasible for a symbolic execution engine to exhaustively explore all the possible states within a reasonable amount of time.
- *Constraint solving*: what can a constraint solver do in practice? SMT solvers can scale to complex combinations of constraints over hundreds of variables. However, constructs such as non-linear arithmetic pose a major obstacle to efficiency.
- *Binary code*: what issues can arise when symbolically executing binary code? While our example of Section 2.6 is written in C, in many cases programs are only available as binary code. The availability of the source code of an application usually simplifies symbolic execution, as it can exploit high-level properties (e.g., object morphology) that can be inferred statically by analyzing the source code.

The above questions and assumptions are addressed in different ways and lead to different choices according to the specific contexts where symbolic execution is used. Typically these choices affect soundness or completeness, however this does not represent a serious limitation in

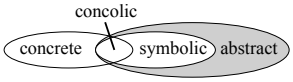

Figure 2.25: Concrete and abstract execution machine models.

many real cases where a partial exploration of the domain of possible execution states may provide enough information to reach the goal (e.g., determining which input values provokes an application failure) within specified time limits.

2.6.1 *Symbolic Execution Engines*

In this section we describe some important principles for the design of symbolic executors and crucial tradeoffs that arise in their implementation. Moving from the concepts of concrete and symbolic runs, we also introduce the idea of *concolic* execution.

2.6.1.1 *Mixing Symbolic and Concrete Execution*

As shown in the previous example, modeling all runs of concrete executions of real-world software programs is desirable but very often impossible in practice.

Symbolic execution, as we have described it so far, cannot explore feasible executions that would result in path constraints that cannot be dealt with [22]. Loss of soundness originates from many sources, e.g. untraceable external code, complex constraints involving non-linear arithmetic or transcendental functions. As constraint solving is a major performance barrier for an engine, we can assess the solvability in an absolute sense but also in terms of efficiency. Considering that practical programs are typically not self-contained, it is very challenging to statically analyze the whole software stack, especially considering the evaluation of every possible side effects.

One way to overcome this issue in practice is to mix concrete and symbolic execution, aka *concolic execution*, where the term concolic is a portmanteau of the words "concrete" and "symbolic" (Figure 2.25). Some applications of this general principle are briefly discussed in the remainder of this section.

DYNAMIC SYMBOLIC EXECUTION A common approach to concolic execution, known as *dynamic symbolic execution* (DSE) or *dynamic test generation* [34], is to have concrete execution drive the symbolic one. A concrete store σ_c is maintained, in addition to the symbolic and the path constraints stores, by the execution engine. It chooses an initial arbitrary input and executes the program both concretely and symbolically, by simultaneously updating the two stores and the path constraints. The symbolic execution wlaks through the same branch taken by the concrete execution, and the constraints extracted from

the branch condition are added to the current set of path constraints. The symbolic engine does not need to invoke the constraint solver to decide if a branch condition is satisfiable, as this is already tested by the concrete execution. Since not all the paths might be explored by concrete execution, the path conditions associated to a branch can be negated and passed to the SMT solver to determine a satisfying assignment for the new constraints, i.e., to generate a new input. This strategy can be repeated until the desired path coverage is reached.

Consider the C function in Figure 2.23 and suppose to choose $a = 1$ and $b = 1$ as input parameters. Under these conditions, the concrete execution takes path $A \rightsquigarrow B \rightsquigarrow C \rightsquigarrow E \rightsquigarrow G$ in the symbolic tree of Figure 2.24. Besides the symbolic stores shown in Figure 2.24, the concrete stores maintained in the traversed states are the following:

- $\sigma_c = \{a \mapsto 1,\ b \mapsto 1\}$ in state A;
- $\sigma_c = \{a \mapsto 1,\ b \mapsto 1,\ x \mapsto 1,\ y \mapsto 0\}$ in states B and C;
- $\sigma_c = \{a \mapsto 1,\ b \mapsto 1,\ x \mapsto 1,\ y \mapsto 4\}$ in states E and G.

After checking that the `assert` conditions at line 8 succeed, we can generate a new control flow path by negating the last path constraint, i.e., $\alpha_b \neq 0$. The solver at this point would generate a new input that satisfies the constraints $\alpha_a \neq 0 \wedge \alpha_b = 0$ (for instance $a = 1$ and $b = 0$) and the execution would continue in a similar way along the path $A \rightsquigarrow B \rightsquigarrow C \rightsquigarrow E \rightsquigarrow F$.

Although DSE uses concrete inputs to drive the symbolic execution toward a specific path, it still needs to pick a branch to negate whenever a new path has to be explored. Notice also that each concrete execution may add new branches that will have to be visited. Since the set of non-taken branches across all performed concrete executions can be very large, adopting effective search heuristics (Section 2.6.2) can play a crucial role. For instance, DART [34] chooses the next branch to negate using a depth-first strategy. Additional strategies for picking the next branch to negate have been presented in literature. For instance, the *generational search* of SAGE [35] systematically yet partially explores the state space, maximizing the number of new tests generated while also avoiding redundancies in the search. This is achieved by negating constraints following a specific order and by limiting the backtracking of the search algorithm. Since the state space is only partially explored, the initial input plays a crucial role in the effectiveness of the overall approach. The importance of the first input is similar to what happens in traditional *black-box fuzzing*; hence, symbolic engines such as SAGE are often referred to as *white-box fuzzers*.

The symbolic information maintained during a concrete run can be exploited by the engine to obtain new inputs and explore new paths.

The next example shows how DSE can handle invocations to external code that is not symbolically tracked by the concolic engine.

Consider function `foo` in Figure 2.26a and suppose that `bar` is not symbolically tracked by the concolic engine (e.g., it could be provided by a third-party component, written in a different language, or analyzed following a black-box approach). Assuming that $x = 1$ and $y = 2$ are randomly chosen as the initial input parameters, the concolic engine executes `bar` (which returns $a = 0$) and skips the branch that would trigger the error statement. At the same time, the symbolic execution tracks the path constraint $\alpha_y \geqslant 0$ inside function `foo`. Notice that branch conditions in function `bar` are not known to the engine. To explore the alternative path, the engine negates the path constraint of the branch in `foo`, generating inputs, such as $x = 1$ and $y = -4$, that actually drive the concrete execution to the alternative path. With this approach, the engine can explore both paths in `foo` even if `bar` is not symbolically tracked.
A variant of the previous code is shown in Figure 2.26b, where function `qux` – differently from `foo` – takes a single input parameter but checks the result of `bar` in the branch condition. Although the engine can track the path constraint in the branch condition tested inside `qux`, there is no guarantee that an input able to drive the execution toward the alternative path is generated: the relationship between a and x is not known to the concolic engine, as `bar` is not symbolically tracked. In this case, the engine could re-run the code using a different random input, but in the end it could fail to explore one interesting path in `qux`.
A related issue is presented by Figure 2.26c. We observe a *path divergence* when inputs generated for a predicted path lead execution to a different path. In general, this can be due to symbol propagation not being tracked, resulting in inaccurate path constraints, or to imprecision in modeling certain (e.g., bitwise, floating-point) operations in the engine. In the example, function `baz` invokes the external function `abs`, which performs a side effect on x by assigning it with its absolute value. Choosing $x = 1$ as the initial concrete value, the concrete execution does not trigger the error statement, but the concolic engine tracks the path constraint $\alpha_x \geqslant 0$ due to the branch in `baz`, trying to generate a new input by negating it. However the new input, e.g., $x = -1$, does not trigger the error statement due to the (untracked) side effects of `abs`. Interestingly, the engine has no way of detecting that no input can actually trigger the error.

As shown by the example, false negatives (i.e., missed paths) and path divergences are notable downsides of dynamic symbolic execution. DSE trades soundness for performance and implementation effort: false negatives are possible, because some program executions

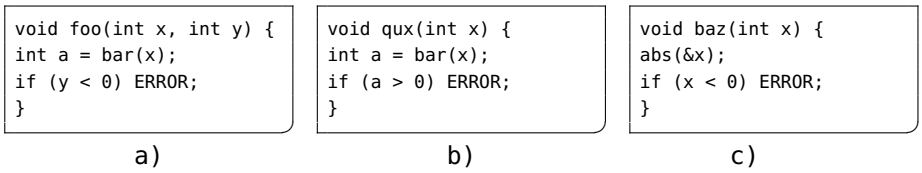

Figure 2.26: Concolic execution: (a) testing of function `foo` even when `bar` cannot be symbolically tracked by an engine, (b) example of false negative, and (c) example of a path divergence, where `abs` drops the sign of the integer at `&x`.

– and therefore possible erroneous behaviors – may be missed, leading to a *complete*, but *under-approximate* form of program analysis. Path divergences have been frequently observed in literature: for instance, [35] reports rates over 60%. [26] presents an empirical study of path divergences, analyzing the main patterns that contribute to this phenomenon. External calls, exceptions, type casts, and symbolic pointers are pinpointed as critical aspects of concolic execution that must be carefully handled by an engine to reduce the number of path divergences.

SELECTIVE SYMBOLIC EXECUTION S²E [27] takes a different approach to mix symbolic and concrete execution based on the observation that one might want to explore only some components of a software stack in full, not caring about others. *Selective symbolic execution* carefully interleaves concrete and symbolic execution, while keeping the overall exploration meaningful.

Suppose a function A calls a function B and the execution mode changes at the call site. Two scenarios arise: (1) *From concrete to symbolic and back*: the arguments of B are made symbolic and B is explored symbolically in full. B is also executed concretely and its concrete result is returned to A. After that, A resumes concretely. (2) *From symbolic to concrete and back*: the arguments of B are concretized, B is executed concretely, and execution resumes symbolically in A. This may impact both soundness and completeness of the analysis: (i) *Completeness*: to make sure that symbolic execution skips any paths that would not be realizable due to the performed concretization (possibly leading to false positives), S²E collects path constraints that keep track of how arguments are concretized, what side effects are made by B, and what return value it produces. (ii) *Soundness*: concretization may cause missed branches after A is resumed (possibly leading to false negatives). To remedy this, the collected constraints are marked as *soft*: whenever a branch after returning to A is made inoperative by a soft constraint, the execution backtracks and a different choice of arguments for B is attempted. To guide re-concretization of B's arguments, S²E also collects the branch conditions during the concrete

execution of B, and chooses the concrete values so that they enable a different concrete execution path in B.

2.6.2 *Path Selection*

Since enumerating all paths of a program can be prohibitively expensive, in many software engineering activities related to testing and debugging the search is prioritized by looking at the most promising paths first. Among several strategies for selecting the next path to be explored, we now briefly overview some of the most effective ones. We remark that path selection heuristics are often tailored to help the symbolic engine achieve specific goals (e.g., overflow detection). Finding a universally optimal strategy remains an open problem.

Depth-first search (DFS), which expands a path as much as possible before backtracking to the deepest unexplored branch, and *breadth-first search* (BFS), which expands all paths in parallel, are the most common strategies. DFS is often adopted when memory usage is at a premium, but is hampered by paths containing loops and recursive calls. Hence, in spite of the higher memory pressure and of the long time required to complete the exploration of specific paths, some tools resort to BFS, which allows the engine to quickly explore diverse paths detecting interesting behaviors early. Another popular strategy is *random path selection*, that has been refined in several variants. For instance, KLEE [21] assigns probabilities to paths based on their length and on the branch arity: it favors paths that have been explored fewer times, preventing starvation caused by loops and other path explosion factors.

Several works, such as EXE [23], KLEE [21], Mayhem [24], and S^2E [27], have discussed heuristics aimed at maximizing code coverage. For instance, the *coverage optimize search* discussed in KLEE [21] computes for each state a weight, which is later used to randomly select states. The weight is obtained by considering how far the nearest uncovered instruction is, whether new code was recently covered by the state, and the state's call stack. Of a similar flavor is the heuristic proposed in [50], called *subpath-guided search*, which attempts to explore *l*ess traveled parts of a program by selecting the subpath of the control flow graph that has been explored fewer times. This is achieved by maintaining a frequency distribution of explored subpaths, where a subpath is defined as a consecutive subsequence of length n from a complete path. Interestingly, the value n plays a crucial role with respect to the code coverage achieved by a symbolic engine using this heuristic and no specific value has been shown to be universally optimal. *Shortest-distance symbolic execution* [51] does not target coverage, but aims at identifying program inputs that trigger the execution of a specific point in a program. The heuristic is based however, as in coverage-based strategies, on a metric for evaluating the shortest

distance to the target point. This is computed as the length of the shortest path in the inter-procedural control flow graph, and paths with the shortest distance are prioritized by the engine.

Other search heuristics try to prioritize paths likely leading to states that are *interesting* according to some goal. For instance, AEG [9] introduces two such strategies. The *buggy-path first* strategy picks paths whose past states have contained small but unexploitable bugs. The intuition is that if a path contains some small errors, it is likely that it has not been properly tested. There is thus a good chance that future states may contain interesting, and hopefully exploitable, bugs. Similarly, the *loop exhaustion* strategy explores paths that visit loops. This approach is inspired by the practical observation that common programming mistakes in loops may lead to buffer overflows or other memory-related errors. In order to find exploitable bugs, Mayhem [24] instead gives priority to paths where memory accesses to symbolic addresses are identified or symbolic instruction pointers are detected.

[76] proposes a novel method of dynamic symbolic execution to automatically find a program path satisfying a regular property, i.e., a property (such as file usage or memory safety) that can be represented by a Finite State Machine (FSM). Dynamic symbolic execution is guided by the FSM so that branches of an execution path that are most likely to satisfy the property are explored first. The approach exploits both static and dynamic analysis to compute the priority of a path to be selected for exploration: the states of the FSM that the current execution path has already reached are computed dynamically during the symbolic execution, while backward data-flow analysis is used to compute the future states statically. If the intersection of these two sets is non-empty, there is likely a path satisfying the property.

Fitness functions have been largely used in the context of search-based test generation [54]. A fitness function measures how close an explored path is to achieve the target test coverage. Several works, e.g., [73, 22], have applied this idea in the context of symbolic execution. As an example, [73] introduces *fitnex*, a strategy for flipping branches in concolic execution that prioritizes paths likely *closer* to take a specific branch. In more detail, given a target branch with an associated condition of the form $|a - c| == 0$, the closeness of a path is computed as $|a - c|$ by leveraging the concrete values of variables a and c in that path. Similar fitness values can be computed for other kinds of branch conditions. The path with the lowest fitness value for a branch is selected by the symbolic engine. Paths that have not reached the branch yet get the worst-case fitness value.

2.6.3 *Symbolic Backward Execution*

Symbolic backward execution (SBE) [25, 28] is a variant of symbolic execution in which the exploration proceeds from a target point to

an entry point of a program. The analysis is thus performed in the reverse direction than in canonical (forward) symbolic execution. The main purpose of this approach is typically to identify a test input instance that can trigger the execution of a specific line of code (e.g., an assert or throw statement). This can be very useful for a developer when performing debugging or regression testing over a program. As the exploration starts from the target, path constraints are collected along the branches met during the traversal. Multiple paths can be explored at a time by an SBE engine and, akin to forward symbolic execution, paths are periodically checked for feasibility. When a path condition is proved unsatisfiable, the engine discards the path and backtracks.

[51] discusses a variant of SBE dubbed *call-chain backward symbolic execution* (CCBSE). The technique starts by determining a valid path in the function where the target line is located. When a path is found, the engine moves to one of the callers of the function that contains the target point and tries to reconstruct a valid path from the entry point of the caller to the target point. The process is recursively repeated until a valid path from the main function of the program has been reconstructed. The main difference with respect to the traditional SBE is that, although CCBSE follows the call-chain backwards from the target point, inside each function the exploration is done as in traditional symbolic execution.

A crucial requirement for the reversed exploration in SBE, as well as in CCBSE, is the availability of the inter-procedural control flow graph which provides a whole-program control flow and makes it possible to determine the call sites for the functions that are involved in the exploration. Unfortunately, constructing such a graph can be quite challenging in practice. Moreover, a function may have many possible call sites, making the exploration performed by a SBE still very expensive. On the other hand, some practical advantages can arise when the constraints are collected in the reverse direction.

2.6.4 *Design Principles of Symbolic Executors*

A number of performance-related design principles that a symbolic execution engine should follow are summarized in [24]. Most notably:

1. *Progress*: the executor should be able to proceed for an arbitrarily long time without exceeding the given resources. Memory consumption can be especially critical, due to the potentially gargantuan number of distinct control flow paths.

2. *Work repetition*: no execution work should be repeated, avoiding to restart a program several times from its very beginning in order to analyze different paths that might have a common prefix.

3. *Analysis reuse*: analysis results from previous runs should be reused as much as possible. In particular, costly invocations to the SMT solver on previously solved path constraints should be avoided.

Due to the large size of the execution state space to be analyzed, different symbolic engines have explored different trade-offs between, e.g., running time and memory consumption, or performance and soundness/completeness of the analysis.

Symbolic executors that attempt to execute multiple paths simultaneously in a single run – also called *online* – clone the execution state at each input-dependent branch. Examples are given in KLEE [21], AEG [9], S²E [27]. These engines never re-execute previous instructions, thus avoiding work repetition. However, many active states need to be kept in memory and memory consumption can be large, possibly hindering progress. Effective techniques for reducing the memory footprint include *copy-on-write*, which tries to share as much as possible between different states [21]. As another issue, executing multiple paths in parallel requires to ensure isolation between execution states, e.g., keeping different states of the OS by emulating the effects of system calls.

Reasoning about a single path at a time, as in concolic execution, is the approach taken by so-called *offline executors*, such as SAGE [35]. Running each path independently of the others results in low memory consumption with respect to online executors and in the capability of reusing immediately analysis results from previous runs. On the other side, work can be largely repeated, since each run usually restarts the execution of the program from the very beginning. In a typical implementation of offline executors, runs are concrete and require an input seed: the program is first executed concretely, a trace of instructions is recorded, and the recorded trace is then executed symbolically. *Hybrid executors* such as Mayhem [24] attempt at balancing between speed and memory requirements: they start in online mode and generate checkpoints, rather than forking new executors, when memory usage or the number of concurrently active states reaches a threshold. Checkpoints maintain the symbolic execution state and replay information. When a checkpoint is picked for restoration, online exploration is resumed from a restored concrete state.

3

XSPARK

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

— Verne *Journey to the Center of the Earth* 1957

This chapter illustrates the previous works done on xSpark, that is the base for the developments made and contribution given by this thesis. xSpark is a Spark extension that offers optimized and elastic provisioning of resources in order to meet execution deadlines. This is obtained by using nested control loops. A centralized control loop, implemented on the master node, controls the execution of the different stages of an application; at the same time multiple local loops, one per executor, focus on task execution. In Figure 3.1 we can see a high level representation of xSpark execution flow. A preliminary Profiling Phase, that is executed once per application, is obtained by executing the application once to obtain information about its runtime execution. The log generated by the application are used to generate its Profiling Data. In the Execution Phase, we control the application by means of xSpark's control loops. The centralized control loop is represented as a Heuristic Based Planner, which exploits the Profiling Data. The profiling data is used to understand the amount of work that is needed to execute the application. This component uses the provided profiling data to determine the amount of resources to assign to executors for each of the application stages, in order to complete the execution within the given deadline. The local loops instead are represented as Control Theory Controllers. They perform fine-grained tuning of the resources assigned to each of the executors using a control theory based controller. This component is used to counteract the possible imprecision of the estimated needed resources, which may be caused by different factors, such as the available memory, etc. Usually Spark applications are run multiple times, as they are reusable and longlasting assets. xSpark exploits an initial profiling phase to create an enriched DAG describing the entire application execution flow, by collecting information about all the stages of the application. For each stage, xSpark annotates the DAG with the execution time (stage duration), the number of task processed, the number of input records read, the number of output record written and the nominal rate, defined as the number of records that a single core processes during one second of execution.

In Listing 3.1 we can see a portion, relative to stage number three, of the profiling data related to a PageRank application executed in xSpark. For example, the duration field contains the serialized total

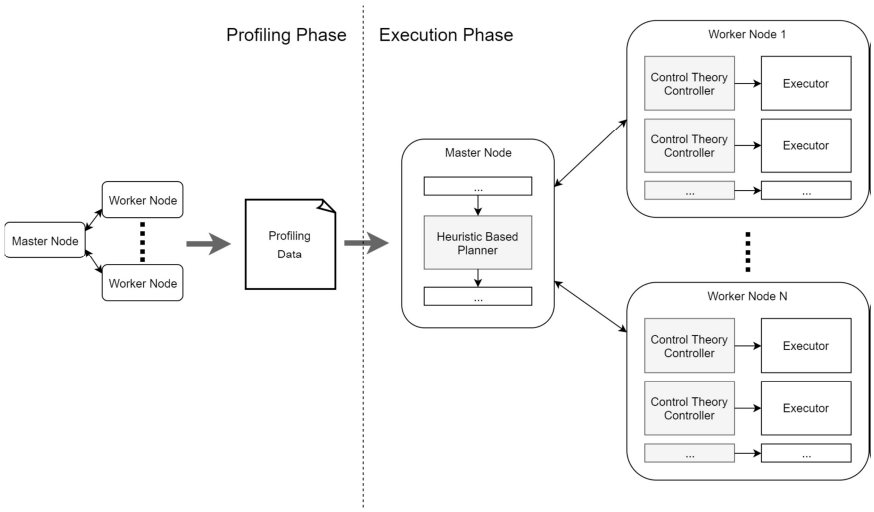

Figure 3.1: High level setup and execution flow of xSpark.

duration of the tasks in milliseconds. At runtime, the annotated DAG allows us to comprehend how much work has already been completed and how much work still needs to be done. This means that xSpark can only optimize the allocation of the resources if all the executions of the same application use the same DAG. This might not always be the case, for example when the code contains branches or loops, because these might need to be resolved in different ways at runtime. If the application DAG at runtime differs from the one obtained during the profiling phase, xSpark is not able to estimate the amount of remaining work. The centralized control loop is activated before the execution of every stage, it uses an heuristic, explained in Section **??**, in order to assign a deadline to the stage, calculate the amount of CPU cores that are needed to satisfy it, and assign cores to the allocated executors. The per-stage deadline takes into account the amount of work already completed, the consumed time, and the overall deadline. All the computations done by the heuristic are based on the information stored in the DAG and obtained during the profiling phase. Unfortunately many factors can influence the actual performance and invalidate the prediction, such as the amount of records that have been filtered-out, the available memory, the number of used nodes, the storage layer dimension, and so on. It is important to remember that Spark mostly uses in-memory data, but there are operations like textFile, saveAsTextFile and saveAsSequenceFile that impose restricting constraints on the storage layer. If not correctly dimensioned, the storage layer might become a bottleneck, causing the throughput to degrade and thus making the provisioning predicted by xSpark incorrect. Local control loops, explained in Section 3.3, counteract this imprecision

Listing 3.1: Example of profiling data from a PageRank application.

```
{ ...
  "3": {
      ...
      "cachedRDDs": [16, 9],
      "duration": 504086,
      "genstage": false,
      "name": "mapPartitions at . . . ",
      "nominalrate": 413525.2675138766,
      "numtask": 1000,
      "parentsIds ": [1, 2],
      "recordsread": 1000,
      " recordswrite": 0.0,
      " shufflerecordsread": 208452298,
      " shufflerecordswrite ": 1000000,
      "skipped": false,
      "weight": 4.97875957673889
      ...
    },
... }
```

by dynamically modifying the amount of CPU cores assigned to the executors during the execution of a stage. This can lead the executor to use more or less resources than the ones previously assigned by the centralized control loop. The local loop controls the progress of a specific executor with respect to the tasks it has assigned. A control theory algorithm determines the amount of CPU cores that must be allocated to the executor for the next control period, typically one second, and assigns them. xSpark uses Docker in order to dynamically allocate CPU cores and memory, as explained in Section 2.1.1. Memory allocation simply sets an upper bound to the memory that each docker container (executor) can use. CPU cores instead are allocated in a more sophisticated way. Using Linux cgroups, Docker can support CPU shares, reservations and quotas. In particular, xSpark uses CPU quotas. CPU shares are not able to limit the number of CPU cores used by a container in a deterministic way, in particular it is not independent of other processes running on the same machine. CPU reservation instead does not have the fine granularity we are looking for, indeed the allocation would be limited to entire cores. By using CPU quotas instead, we have a reliable and tunable mechanism that provides also fine granularity allocation, in particular it allows xSpark to allocate fractions of cores to the containers (executors), with a precision up to 0.05 cores.

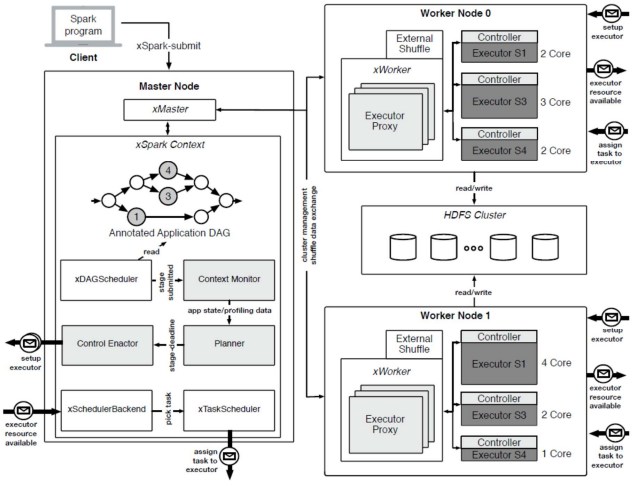

Figure 3.2: Architecture of xSpark. New components are represented in light grey boxes, meanwhile those that start with an x are the modified ones. In dark grey are represented the containerized components (the executors).

3.1 ARCHITECTURE

To achieve the objectives of xSpark, the architecture and processing model of Spark have been modified. In Figure 2.2 we can se how xSpark architecture differs from Spark. The principal architectural change introduced by xSpark is an increased focus on stages. Instead of considering entire applications, xSpark reasons on per-stage deadlines. xSpark instantiates an executor per stage per worker, instead of a single executor per worker for all the stages that will be executed. This way the resources that are allocated to a single executor only impact the performance of the stages that are associated with it, this leads to a fine grained control over the different stage, and thus on the entire application. When multiple stages are run in parallel, multiple executors can be running on the same worker node. When a stage is submitted for execution, one executor per worker is created and bound to that stage. This way computation and data are equally spread across the entire cluster. Thanks to containers, xSpark can isolate the execution of the different executors that are running on the same worker node and achieve quick, fine-grained resource provisioning. On average, containers can be modified in less than one second, allowing a more precise way of allocating cores. Users submit applications and their deadlines to the master node using the submit command. This creates an *xSparkContext* on the master node, containing a*xMaster* object that is used to manage the cluster and that knows all the

resources that are available on each worker node. A *xSparkContext* is composed by six components:

- *xDagScheduler* schedules the stages according to the application's DAG. The submitted stage is enriched with the information obtained during the profiling phase
- *ContextMonitor* monitors the progress of the application, taking into account stage scheduling and completion. It also stores information about the performance of the execution, that will be later used to calculate the deadlines and the resources needed by upcoming stages
- *Planner* is heuristics based and is used to calculate the deadlines and resources associated with a stage
- *ControlEnactor* determines when a stage is ready to be executed, meaning there are enough resources (cores) and sufficient executors become available. It also has the duty of initializing the different executors.
- *xSchedulerBackend* controls the stage execution. In particular it launches new tasks by taking into account the resources availability and registers their completion
- *xTaskScheduler* also controls stage execution, with the goal of allocating tasks to the available cores to optimize data locality. In general, the closer a data partition required by a task is to the task's executor, the better.

xSpark also modified the worker nodes. Each node contains a *xWorker* that connects to a *xMaster*, generates local controllers for the executors, and controls the evolution of its executor by dynamically scaling their resources. xWorker creates an *Executor Proxy* for each of the executors that are running on its node. These proxies are placed between the executors and the *xSchedulerBackend*, and are used to monitor the execution progress of the stage assigned to the executors. It is important to remember that each executor focuses on a single stage at a time. The heuristic calculates how many tasks must be executed by each of the executors of the same stage, the strategy is to have en equal number of tasks assigned to each of the executors. This way the deadline assigned to each of the executors coincides with the deadline of their stage. This allows the different executors of the same stage to not be synchronized. Native Spark instead requires that executors with free resources spontaneously require new tasks to execute from the master node.

Every xWorker uses an External Shuffle Service. Native Spark moves data across the cluster in different ways. If the data is stored on executor's memory, then the executor itself manages the data exchange.

If the data is stored on an external storage system (e.g., HDFS cluster), they can be retrieved by using different communication protocols. If the data is stored in the internal storage of a worker node, then the data is managed by the External Shuffle Service. Notice that this is not the default, but xSpark adopts this technique to be able to assign zero CPU cores to an executor, without loosing the ability to read data, since it is effectively not performed by the executor but by the external service.

3.2 heuristic

xSpark uses a heuristic to compute per-stage deadlines and to estimate how many cores must be allocated for a stage to successfully fulfill the deadline. In order to do this, at submission time the user is asked to specify three parameters: i) the application deadline, ii) the cluster size, and iii) the number of cores per worker node. Before executing the application, xSpark performs a feasibility check given the available resources.

When a stage is submitted for execution, its deadline is computed

$$deadline(sk) = \frac{\alpha \cdot ApplicationDeadline - SpentTime}{weight(sk)}$$

where $SpentTime$ is the time already spent for execution and α a value between 0 and 1 that xSpark uses to be more conservative with respect to the provided ApplicationDeadline. The weight is computed

$$\begin{cases} w1(sk) = \#(RemainingStages + 1) \\ w2(sk) = \frac{\Sigma_{a_{i=k}}^{k+w_1} duration(s_i)}{duration(sk)} \\ weight(sk) = \beta \cdot w1(sk) + (1 - \beta) \cdot w2(sk) \end{cases}$$

where w1 is the number of stages still to be scheduled (s included) and w2 is the rate between the duration of s and the duration of the remaining stages (s included). xSpark then proceeds to estimate how many cores are needed to execute the stage:

$$estimatedCores(sk) = \lceil \frac{inputRecords(sk)}{deadline(sk) \cdot nominalRate(sk)} \rceil$$

where inputRecords is the number of records that will be processed by sk and nominalRate is the number of records processed by a single core per second in stage sk.

Since xSpark controls the resource allocation of a stage before and during the execution, the maximum amount of allocable cores needs to be greater than the estimated one, in order to be able to accelerate when progressing slower than expected

$$maxAllocableCores(sk) = overscale \cdot estimatedCores(sk)$$

The final step is to determine the initial number of cores that should be assigned to the different executors, xSpark distributes the cores equally amongst the available workers by creating one executor per stage per worker. In this way, it is guaranteed that executor performances will be equal, and that xSpark can compute the same deadline for all the executors. The initial number of cores per executor is computed as

$$\mathtt{initCorePerExec}(sk) = \lceil \frac{\mathtt{maxAllocableCores}(sk)}{\mathtt{overscale} \cdot \mathtt{cq} \cdot \mathtt{numExecutors}} \rceil \cdot \mathtt{cq}$$

where numExecutors is the number of executors and cq is the core quantum, a constant that defines the quantization applied to resource allocation, the smaller this value is, the more precise the allocation.

3.3 CONTROLLER

Each containerized executor has an associated local controller, whose goal is to fulfill the per-stage deadline taking into account external disturbances by dynamically allocating CPU cores. The controllers use control theory, with no heuristic involved.

The centralized control loop determines the desired stage duration, the maximum and the initial number of cores that should be assigned to the executors and the number of tasks that must be processed. Local controllers adjust the number of allocated cores, according to the work that has already been accomplished.

Executors that are dedicated to different stages are implicitly independent, and thus their controllers are also independent. The executors that are running in parallel on the same stage must complete the same amount of work (number of tasks) in the same desired time. This means that local controllers are independent and do not need to communicate amongst themselves. Moreover, the heuristic is relegated outside the local controller, so that it cannot compromise the controller's stability. In the controller, the progress set point is chosen based on the desired completion time. Its value is received from the centralized control loop. In Figure 3.3 we see the prescribed completion percentage, in particular t_{co} is the desired completion time and $\alpha \in (0, 1]$ is a configuration parameter used to determine how much earlier we are willing to complete the execution with respect to requested deadline,In order to track the set point ramp, we need to use a Proportional plus Integral (PI) controller. As a result, the discrete-time controller in state space form reads:

$$\begin{cases} x_C(k) = x_C(k-1) + (1-\alpha)(a^o_{\%}(k-1) - a_{\%}(k-1)) \\ c(k) = K x_C(k) + K(a^o_{\%} - a_{\%}(k)) \end{cases}$$

where $(a^o_{\%}$ is the prescribed progress percentage at each k control step and $a_{\%}$ is the accomplished completion percent at each k control step.

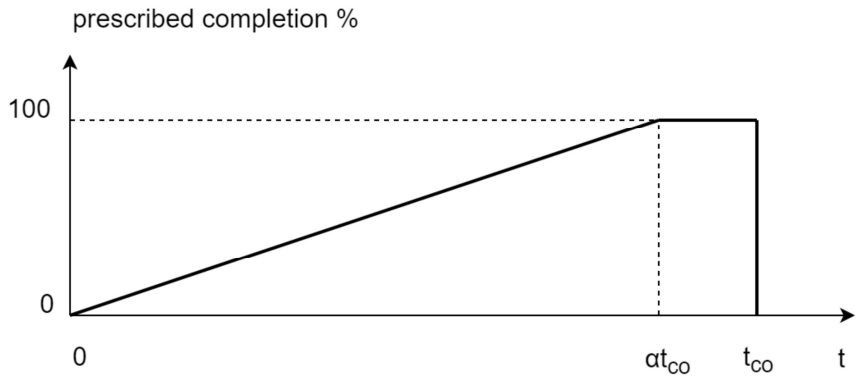

Figure 3.3: Set point generation for an executor controller.

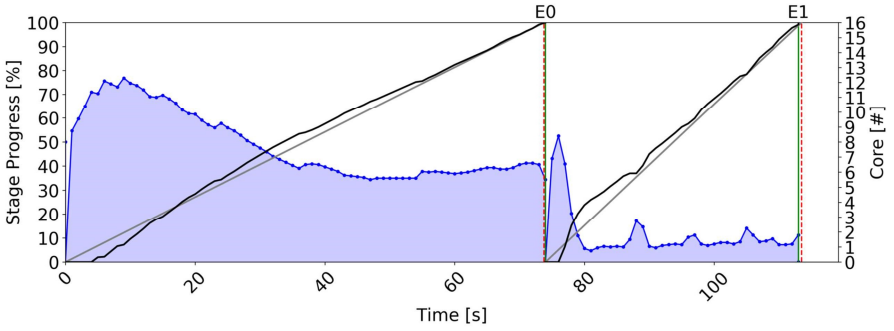

Figure 3.4: CPU cores allocated to the application aggregate-by-key running on a xSpark worker node. The blue line represents the allocated cores over the time, gray and black line are desired and actual progress rate respectively. Green line represents the obtained stage ending, while dashed red line is the desired one.

Notice that it is possible that the controller computes a negative value for $c(k)$, the CPU cores that need to be allocated. To fix this problem $c(k)$ must be clamped between a minimum c_{min} and a maximum c_{max}. To maintain consistency, we need to recompute the state $x_C(k)$ as

$$x_C(k) = \frac{c(k)}{K} - a^o_{\%}(k) + a_{\%}(k)$$

In Figure 3.4 the cores allocated to an application executor running on a worker node are shown. The running application is *aggregateby- key* and is composed by two stages. The black line represents the progress of the stage, meanwhile the gray one is the desired progress rate. The goal of the controller is to reduce the error, i.e., the distance between the two lines. We want the black line to follow as much as possible the gray line.

4

METHODOLOGY

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

— Verne *Journey to the Center of the Earth* 1957

Big data applications are widely used in the industry and research fileds. Specialized distributed frameworks are used to execute these applications on clusters of computers, very often made by cloud computing resources, like virtual machines and virtual storage. Apache Spark is probably the most popular big data processing framework. Spark organizes computations in directed acyclic graphs (DAG's) and its declarative API offers the capability to make transformations to datasets and return results to the client program, usually a standard Java, Python or Scala application.

Spark starts executing a program by identifying jobs, delimited by the presence of actions in the code, and stages (within jobs), delimited by operations that require data to be shuffled (i.e., moved among executors), thus breaking locality. Indeed, Spark distinguishes between narrow and wide transformations specifically for this purpose; the former do not shuffle data (e.g., map, filter, etc.), while the latter do (e.g., reduceByKey, etc.). Spark "identifies" all the operations that are to be executed, up to the first action, and materializes them as a directed acyclic graph (DAG). The DAG is not the control-flow graph of the job's code. It does not contain branches and loops since they were already resolved during the execution of the driver program. The DAG defines the execution order among stages, and defines the extent to which stages can be executed in parallel. For each job, Spark computes a DAG also called *Parallel Execution Plan* (*PEP*) to maximize the parallelism while executing an application. In fact, a stage is, by definition, executed in parallel, and also different stages can be executed at the same time. For this reason, Spark materializes *PEPs* as directed acyclic graphs of stages while the complete PEP* of an application is simply the sequence of the *PEPs* of its jobs .A Spark application is usually composed by several jobs executed using a LIFO queue, therefore the PEP* of a Spark application is composed by the sequence of the *PEPs* generated by the application jobs (an action corresponds to a job). In fact, Spark does not "compile" the code of the driver program to generate the application PEP* but it incrementally generates it as soon as an action is reached.

4.1 PROBLEM STATEMENT

Our goal is to support efficient execution of deadline-based QoS constrained multi-PEP* Spark applications, i.e. applications whose execution flow cannot be represented with a single *PEP*, and whose actual execution flow is only known at application execution runtime. In addition, the execution time is constrained by a user-defined deadline, that is the expected duration of the application.

The literature contains several works exploring adaptation capabilities, formal guarantees or response time estimation for Spark applications based on their *PEP*-based structure [44, 13, 12], assuming that the *PEP* of the application does not change with respect to different data input or parameters. However, the *PEP* uniqueness assumption does not hold if conditional branches or loops are present in the control flow of the client program.

This is particularly relevant in Spark because of the possibility of evaluating partial results through actions. In fact, these values can be used in the code as part of conditional expressions that can create branches in the control flow graph. In this cases the conditional branches govern the final structure of the *PEP* and also the operations that form a stage while loops influence the number of repetition of either transformations, stages or actions.

Our solution is based on xSpark, a modified version of Apache Spark, developed at Politecnico di Milano [13, 12], that has demonstrated the capability to execute deadline-constrained single-*PEP* applications by using resources more efficiently than Spark would do in running the same applications. xSpark is able to use less resources than native Spark and can complete executions with less than 1% error in terms of set deadlines.

Given all the above stated, in order to reach our objective of efficiently running deadline-based QoS constrained multi-*PEP* Spark applications, we need to positively answer to the following Research Questions:

RQ$_1$ - [Effectiveness]: Does our solution effectively control the execution of the Spark applications?

RQ$_2$ - [Efficiency]: To what extent can our solution improve the resource allocation capabilities of xSpark, given it used a single, constant *PEP*?

Most of the approaches in the literature (e.g., [33, 66, 44, 52]) use the execution graph to reason on the work to do, the 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 hidden 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 them 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 execution graph would then comprise two jobs if $c \geqslant 0$; it would only comprise the first job otherwise. This simple example shows how Spark can return partial results (c) through actions to the driver program and use these results to evaluate conditional (loop) expressions, and thus produce different execution graphs.

To overcome this problem, xSpark and many solutions [66, 44] exploit an initial profiling phase to retrieve the execution graph and collect some performance metrics. Back to the simple example sketched above, the initial profiling would simply return the execution graph 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 execution graph 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 time would be foreseen.

```
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()
```

Figure 4.1: Example Spark application with conditional branches and loops.

The execution graph can also depend on user parameters or local variables and they must be considered in a sound analysis. For example, Figure 4.1 shows the code of an example application that takes two input parameters *numIterations* and *threshold*; its execution graph depends on both user parameters and input dataset. The first two *aggregate* jobs[1] 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)

1 In Spark *aggregateByKey* is a transformation while *aggregate* is an action.

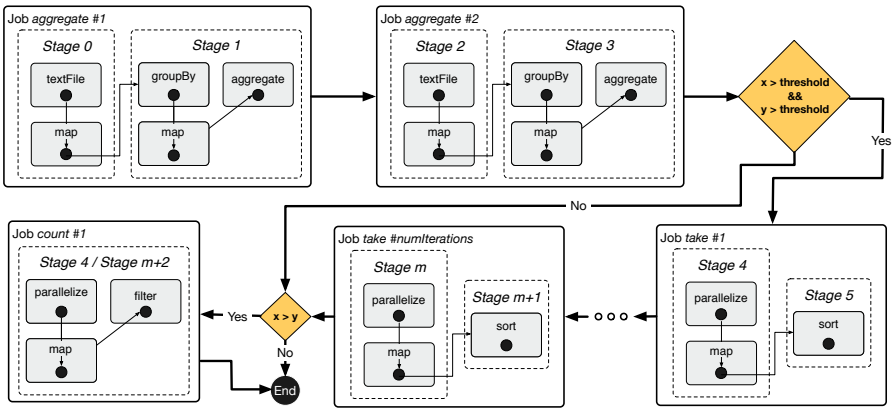

Figure 4.2: The four *PEPs* of the application of Figure 4.1.

is repeated *numIterations* times (`for` loop). Finally, if $x > y$ (line 11) *count* (line 12) is executed.

This simple code corresponds to four possible execution graphs (Figure 4.2): i) the sequence of the two *aggregate*s (if the two conditional statements are both 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).

4.2 SOLUTION OVERVIEW

This section contains a functional level description of the proposed solution, of its components and their interactions to make the solution work.

This thesis work presents *xSpark*$_{\text{SEEPEP}}$, a toolchain providing the capability to manage the efficient execution of deadline-based QoS constrained multi-*PEP* Spark applications.

xSpark$_{\text{SEEPEP}}$ is the result of the integration of *SEEPEP*, a tool exploiting symbolic execution techniques to generate the path condition associated to each possible *PEP*s produced by different inputs and parameters, with (a modified version of) xSpark. Moreover, *xSpark*$_{\text{SEEPEP}}$ generates a launcher with a synthesized dataset for each *PEP* and an artifact to retrieve the feasible *PEP*s given a set of symbolic variables resolved to a value. Finally, we integrated this approach with xSpark, an extension of Spark that can control the duration of Spark applications according to specified deadlines through dynamic resource allocation.

The evaluation shows that SEEPEP is able to effectively extract all the DAGs generated by Spark applications and that xSpark reduces the number of deadline violation thanks to the presented integration.

In the remainder of this section we will go through a more detailed explanation of the solution's components and how they cooperate to provide the final result.

4.2.1 *SEEPEP*

In this section we describe SEEPEP, *Symbolic Execution-driven Extraction of Parallel Execution Plans*, 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 xSpark (see Section **??**) with the ability of tuning its adaptation strategy according to the worst-case behaviour of the application. At runtime, our extended version of xSpark exploits the *GuardEvaluator* to refine the control policy by recomputing the worst-case estimation every time the current worst-case refers to a program path for which the execution condition stored in the PEP* becomes unsatisfiable. Below, we describe each phase of SEEPEP in detail.

4.2.2 *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 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 for 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 lightweight 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 fashion.

Figure 4.3 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 4.3, where s' is the state reached from s when executing the transition t.

Figure 4.3 specifies 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 the variable `x` to the value of expression `e` in the current state (ii) the assume operations `assume(c)` by conjoining the current path condition with the value of condition `c` in the current state ($pc \wedge [\![c]\!]_{vv}$). The last two cases in Figure 4.3 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 model the result of the aggregation operator called at the program

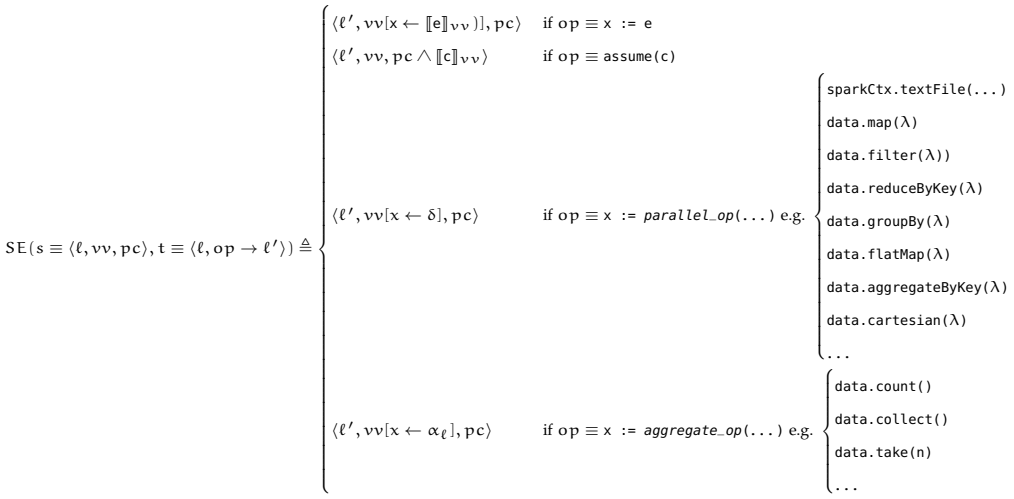

Figure 4.3: Symbolic execution algorithm of SEEPEP.

location ℓ. (For simplicity the figure omits the further 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 4.3 exemplifies a set of both `parallel_op` and `aggregation_op` operations. These examples include Spark operations that appear in any listing in this paper. Beyond these examples, with reference to the RDD Programming Guide [7], the `parallel_op` operations of Figure 4.3 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(...)` operation. 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, which is the reason why we can embrace 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 Figure 4.3 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 formula 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 [18] that relies on the constraint solver Z3 [56].

For example, for the Spark application in Figure 4.1, 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:

- $\alpha_5 \leqslant \mathit{thresh} \wedge \alpha_5 > \alpha_7$;
- $\alpha_5 \leqslant \mathit{thresh} \wedge \alpha_5 \leqslant \alpha_7$;
- $\alpha_5 > \mathit{thresh} \wedge \alpha_7 \leqslant \mathit{thresh} \wedge \alpha_5 > \alpha_7$;
- $\alpha_5 > \mathit{thresh} \wedge \alpha_7 > \mathit{thresh} \wedge \mathit{iters} \leqslant 0 \wedge \alpha_5 > \alpha_7$;
- $\alpha_5 > \mathit{thresh} \wedge \alpha_7 > \mathit{thresh} \wedge \mathit{iters} \leqslant 0 \wedge \alpha_5 \leqslant \alpha_7$;

while it identifies the unsatisfiable path condition $\alpha_5 > \mathit{thresh} \wedge \alpha_7 \leqslant \mathit{thresh} \wedge \alpha_5 \leqslant \alpha_7$.

For programs with loops, like the one in the figure, SEEPEP bounds the iterations of the loops to an user-defined maximum value, thus guaranteeing to have to symbolically analyze a finite amount of paths.

4.2.3 *Search-Based Test Generation*

SEEPEP exploits the path conditions identified with symbolic execution as above, to generate test cases (a test case for each path condition) comprised of 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 in the next phase of SEEPEP, to profile the behavior of the *PEP* generated by the execution each path of the driver program.

To generate a test case for a given path condition, SEEPEP incrementally explores the space of the possible test cases in 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.

4.2.3.1 *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 and concrete datasets). The algorithm samples the possible values of the inputs in the style of *genetic algorithms*. It starts with generating a *population* of test cases comprised of randomly picked 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-generation population with *mutation* and *crossover* operators. Mutation operators generate new test cases by randomly modifying some inputs of a test case of the previous-generation population. The crossover operators generate new test cases as the children of some pair of test cases of the previous-generation 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 path condition, one of those identified in the previous phase, 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; Otherwise, the fitness function yields a value greater than zero that the search algorithm exploits to comparatively order the test cases of the current population. The search algorithm proceeds with probabilistically favouring the application of mutation and crossover operators to test cases with lower distance from the goal, thus increasing the chances to eventually converge 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 results of aggregation actions (the α_ℓ symbols of Figure 4.3) to the corresponding results collected while executing the test case. If the test case does not execute an aggregation action referred 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 $\mathtt{true}$ for the concrete assignment induced by the test case, then the fitness is zero: indeed the test case satisfies the path condition. Otherwise, the fitness is the positive value yielded by the following formula (let t be the test case, and pc be the path condition):

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

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):

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

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

We make the following remarks about the SEEPEP fitness function. Function $distance$ yields always a value in the interval $[0, 1]$, and thus 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 are missing the satisfaction of the conditional for increasingly smaller amounts. Thus, the overall $fitness$ is increasingly closer to zero, the higher the number of satisfied or close-to-be-satisfied conditionals, as expected.

4.2.3.2 *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 in consideration of 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*.

A SDD dataset synthetically represents a RDD object with many data points that hold the same value. In SDD format, a 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 the SDD 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 a 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 for the RDD objects (e.g., operations like `map`, `filter`, `reduceByKey`, etc.) to sequencial operations executed against the data blocks in the SDD objects. For example, a `map(λ)` transformation executed on a 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 data blocks of D that satisfy the condition $\lambda(b_i)$. Yet, a `count()` action on D yields the value $\sum^i b_i.size$ as result. Our SDD objects handle all transformations and actions defined in the RDD Programming Guide [7].

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 implements the search algorithm described in this section based on the SUSHI test generation framework [19, 20]. 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 the tool EvoSuite [32] to use these fitness functions.

For example, with reference to a path condition computed for the Spark application in Figure 4.1, e.g., one of those reported in the previous section, SEEPEP may compute a test case resembling like 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);
```

that 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 that the application will read as input (lines 5–6), and executes the application with these inputs.

$$a = \lceil b + c \rceil$$

4.2.4 *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 comprised of the same amount of data. It executes the test cases against the target application in fully parallel fashion. While executing each test case, SEEPEP stores the *PEP* that the Spark engine produces before starting each parallel execution 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

$$\langle pc \rightarrow \mathit{PEPs}, times \rangle$$

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 an an artifact that we call *GuardEvaluator* that takes as input partial set of concrete values of the symbolic variables, evaluates the path conditions of triples in the PEP* against these values, identifies which path conditions evaluate to $false$ for these values, and returns as output the subset of the PEP* with only the triples with non-falsified path conditions. In the control policy that we define in the next section, we invoke the guard evaluator at runtime, feeding it with the concrete values of the input parameters and incrementally with the concrete values of the executed actions, to stay tuned on the program paths that are possibly reachable at every intermediate execution state.

SEEPEP addresses the possible incompleteness of either its symbolic execution and search phase as follows. 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 it indeed happens to dismiss some program path (if any) that iterates any loop more than that amount. In this case, SEEPEP tracks the path conditions $\hat{pc}$ that correspond 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 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 take decisions to mitigate the impact of these unforeseen situations.

4.2.5 *xSpark*$_{SEEPEP}$

This section describes how *SEEPEP* integrates within xSpark: the resulting tool-chain is called *xSpark*$_{\text{SEEPEP}}$. *SEEPEP* produces the path conditions associated with the different *PEPs* of the application, a set of test cases for each *PEP* for profiling, and a *GuardEvaluator* to allow xSpark to select the most appropriate *PEP* at runtime. At each execution step, *GuardEvaluator* always returns the *PEPs* whose associated path conditions still hold true.

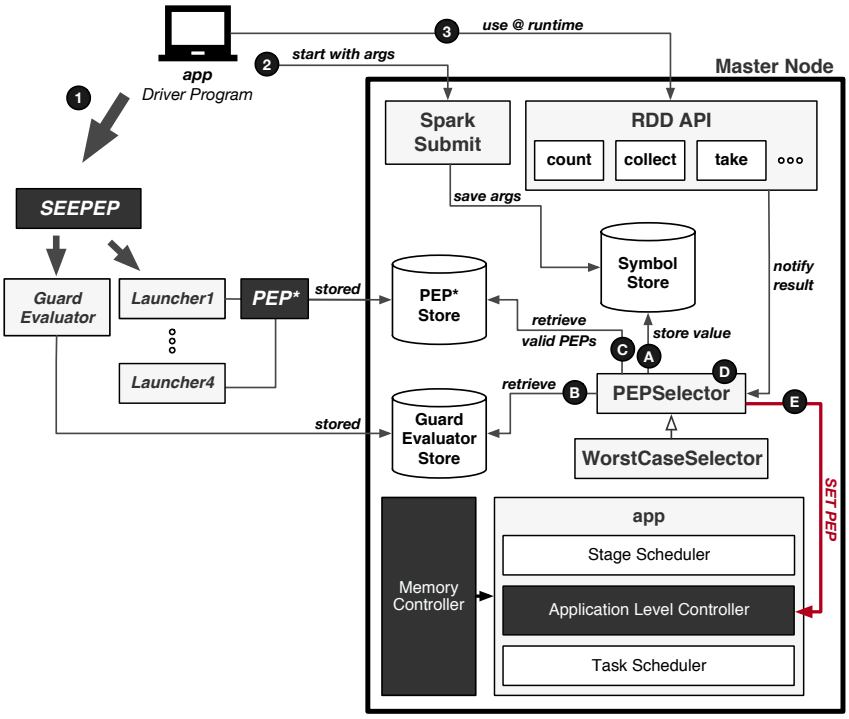

Figure 4.4: *xSpark*$_{\text{SEEPEP}}$

Figure 4.4 shows the main elements of the tool-chain and exemplifies it on profiling and controlling the example application of Figure 4.2 —*app*, hereafter. As first step, *SEEPEP* generates the 4 (n, in general) launchers, which activate the four (n) *PEPs* of the program, and an application specific *GuardEvaluator*.

The new toolchain associates each *PEP* to its path condition, it uses the generated launchers to obtain the profiling data of each *PEP* and synthesize the PEP* for the application that is then stored in the master

node in component *PEP* Store*. Finally the generated *GuardEvaluator*, which implements a common interface to be dynamically instantiated and used without a static import in the source code of xSpark, is also stored in the master node (component *GuardEvaluator Store*).

After this phase, `app` can be executed and controlled by xSpark. Since the application parameters can be part of a path condition as symbols, we modified component *SparkSubmit* to store their values. A symbol is identified by application name, action name, code line in the driver program, and a counter to take multiple executions into account (i.e., loops or recursive functions).

At runtime, everytime an action is executed, that is, a result is computed and returned to the driver program, our modified version of the *RDD API* notifies component *PEPSelector* that a new result is available. This component 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 to provision resources. *PEPSelector* saves computed results into component *Symbol Store*, retrieves an instance of the dedicated *GuardEvaluator*, and feeds it with all the symbols resolved by the aforementioned results. *GuardEvaluator* returns the list of *PEP* whose path conditions still hold.

This means, for example, that 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 only two *PEPs* whose path conditions still hold: it excludes all the path conditions that depends on the expression $x > \texttt{threshold}$). This way, since the *PEP* is updated constantly, xSpark becomes aware of what has been actually done, and can use this information to refine resource provisioning.

Note that *PEPSelector* receives all the valid *PEPs* and computes the next *PEP* to use. This selection can be customized by the user. 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 could privilege a *PEP* that corresponds to an average case instead of the worst one.

5 IMPLEMENTATION

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

— Verne *Journey to the Center of the Earth* 1957

In this chapter we show the implementation details of *xSpark*$_{\text{SEEPEP}}$, which consists of the modifications to existing xSpark component, new components added to xSpark, *SEEPEP* concrete application *launchers* and the xSpark-dagsymb python tool that was used to launch the experiments to generate the data for the evaluation of the solution.

5.1 OVERVIEW

Figure 5.1 shows a simplified overview of the components of the solution. New components are highlighted with a yellow dotted-pattern background, modified components are are highlighted with a grey background.

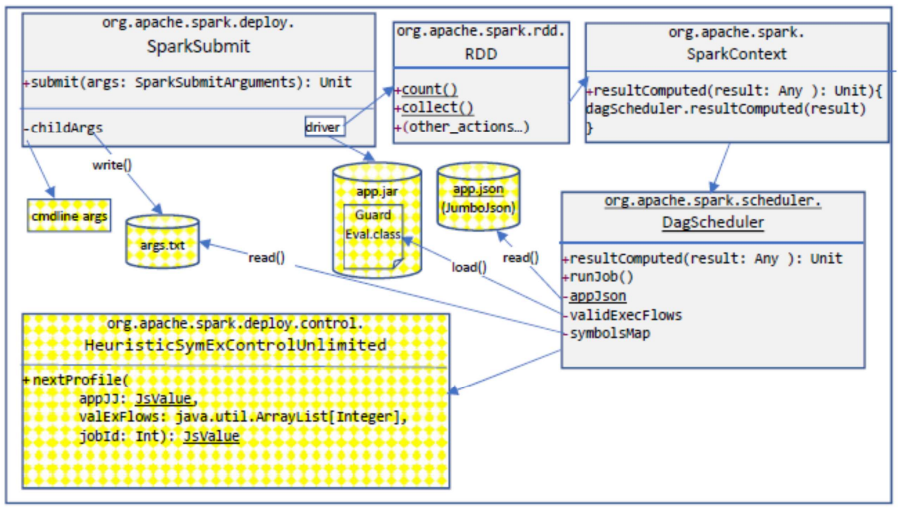

Figure 5.1: Simplified solution components overview.

5.1.1 *Backgound: Current xSpark Heuristic*

We recap here the information given in Section 3.2.

xSpark uses a heuristic to compute per-stage deadlines and to estimate how many cores must be allocated for a stage to successfully fulfill the deadline. At submission time three parameters: are collected: i) the application deadline, ii) the cluster size, and iii) the number of cores per worker node. Before executing the application, xSpark performs a feasibility check given the available resources. When a stage is submitted for execution, its deadline is computed

$$deadline(sk) = \frac{\alpha \cdot ApplicationDeadline - SpentTime}{weight(sk)}$$

where $SpentTime$ is the time already spent for execution and α a value between 0 and 1 that xSpark uses to be more conservative with respect to the provided ApplicationDeadline. The weight is computed

$$\begin{cases} w1(sk) = \#(RemainingStages + 1) \\ w2(sk) = \frac{\Sigma_{a_{i=k}}^{k+w_1} duration(s_i)}{duration(sk)} \\ weight(sk) = \beta \cdot w1(sk) + (1 - \beta) \cdot w2(sk) \end{cases}$$

where w1 is the number of stages still to be scheduled (s included) and w2 is the rate between the duration of s and the duration of the remaining stages (s included). xSpark then proceeds to estimate how many cores are needed to execute the stage:

$$estimatedCores(sk) = \lceil \frac{inputRecords(sk)}{deadline(sk) \cdot nominalRate(sk)} \rceil$$

where inputRecords is the number of records that will be processed by sk and nominalRate is the number of records processed by a single core per second in stage sk.

Since xSpark controls the resource allocation of a stage before and during the execution, the maximum amount of allocable cores needs to be greater than the estimated one, in order to be able to accelerate when progressing slower than expected

$$maxAllocableCores(sk) = overscale \cdot estimatedCores(sk)$$

The final step determines the initial number of cores that should be assigned to the different executors. xSpark distributes the cores equally among the available workers by creating one executor per stage per worker. In this way, it is guaranteed that executor performances will be the same for each of them, and that xSpark can compute the same deadline for all the executors. The initial number of cores per executor is computed as

$$initCorePerExec(sk) = \lceil \frac{maxAllocableCores(sk)}{overscale \cdot cq \cdot numExecutors} \rceil \cdot cq$$

where numExecutors is the number of executors and cq is the core quantum, a constant that defines the quantization applied to resource allocation, the smaller this value is, the more precise the allocation.

5.1.2 *Current xSpark Scheduling Limitation*

At runtime, an annotated DAG allows us to comprehend how much work has already been completed and how much work still needs to be done. This means that xSpark can only optimize the allocation of the resources if the execution of all jobs of the application use the same DAG. This might not always be the case, for example when the code contains branches or loops, because these might need to be resolved in different ways at runtime. This is a severe limitation of the xSpark capability to manage real-world applications.

5.2 SCOPE AND OBJECTIVE OF THE IMPLEMENTATION WORK

The current work, by addressing the xSpark limitation explained above, aims at extending the scope of applicability of xSpark enhancing it with the capability to manage the case of applications that can potentially generate, at runtime, a different DAG at each execution. The code in this kind of applications includes conditional branches or iterative loops whose outcomes can only be resolved at runtime because they depend on user input values or results from previous computations that cannot be predicted or folded to constant values by the compiler.

5.2.1 *Symbolic Execution*

Executing a program symbolically means to simultaneously explore multiple paths that a program could take under different inputs. The key idea is to allow a program to take on symbolic – rather than concrete – input values. Execution is performed by a symbolic execution engine, which maintains for each explored control flow path: (i) a first-order Boolean formula that describes the conditions satisfied by the branches taken along that path, and (ii) a symbolic memory store that maps variables to symbolic expressions or values.

When a conditional branch is met, both sides of the branch are executed. Branch execution updates the formula, while assignments update the symbolic store. A symbolic execution tree is generated with an execution state associated with each node, containing the statement to be executed, the symbolic store, and the path conditions (a formula that expresses a set of assumptions on the symbols). The leaves of the tree identify the end of the computations, and tracing back from each leaf up to the root of the tree allows us to reconstruct, in reverse order, all the possible execution paths of the program.

5.2.2 *xSpark*$_{\text{SEEPEP}}$ *vs. xSpark*

The first important difference between *xSpark*$_{\text{SEEPEP}}$ and xSpark is in the profiling of the applications. xSpark requires the generation of a single DAG profile per application, while *xSpark*$_{\text{SEEPEP}}$ requires a family of DAG profiles, one for each possible execution path, each of them associated to a unique set of "Path Conditions". Profile information is collected in special JSON files, called "JSON profiles". Listing 5.1 shows an example of a JSON profile.

Listing 5.1: Example of JSON profile.

```
{
    "0": {
        "RDDIds": {
            "0": {
                "callsite": "textFile at PromoCalls.java:34",
                "name": "hdfs://10.0.0.4:9000//user/ubuntu/
                    last24HoursLocalCalls.txt"
            },
            "1": {
                "callsite": "textFile at PromoCalls.java:34",
                "name": "hdfs://10.0.0.4:9000//user/ubuntu/
                    last24HoursLocalCalls.txt"
            }
        },
        "actual_records_read": 597900000.0,
        "actual_records_write": 597900000.0,
        "actualtotalduration": 43000.0,
        "bytesread": 55618499408.0,
        "byteswrite": 0.0,
        "cachedRDDs": [],
        "duration": 17000.0,
        "genstage": false,
        "id": 0.0,
        "io_factor": 1.0,
        "jobs": {
            "0": {
                "id-symb": "count_PromoCalls.java:34_0",
                "stages": [
                    0
                ]
            },
            "1": {
                "id-symb": "count_PromoCalls.java:42_0",
                "stages": [
                    1
                ]
            },
            "2": {
                "id-symb": "count_PromoCalls.java:45_0",
                "stages": [
```

```
                    2
                ]
            }
        },
        "monocoreduration": 705582.0,
        "monocoretotalduration": 2170942.0,
        "name": "count at PromoCalls.java:34",
        "nominalrate": 847385.562556868,
        "nominalrate_bytes": 78826414.8008311,
        "numtask": 500,
        "parentsIds": [],
        "recordsread": 597900000.0,
        "recordswrite": 0.0,
        "shufflebytesread": 0.0,
        "shufflebyteswrite": 0.0,
        "shufflerecordsread": 0.0,
        "shufflerecordswrite": 0.0,
        "skipped": false,
        "t_record_ta_executor": 0.0011801003512293025,
        "totalduration": 43000.0,
        "weight": 3.0384051747351832
    },
    "1": {
        "RDDIds": {
            "2": {
                "callsite": "textFile at PromoCalls.java:35",
                "name": "hdfs://10.0.0.4:9000//user/ubuntu/
                    last24HoursLocalCalls.txt"
            },
            "3": {
                "callsite": "textFile at PromoCalls.java:35",
                "name": "hdfs://10.0.0.4:9000//user/ubuntu/
                    last24HoursLocalCalls.txt"
            },
            "4": {
                "callsite": "filter at PromoCalls.java:36",
                "name": "MapPartitionsRDD"
            }
        },
        "actual_records_read": 597900000.0,
        "actual_records_write": 597900000.0,
        "bytesread": 55618499408.0,
        "byteswrite": 0.0,
        "cachedRDDs": [],
        "duration": 13000.0,
        "genstage": false,
        "id": 1.0,
        "io_factor": 1.0,
        "monocoreduration": 737425.0,
        "name": "count at PromoCalls.java:42",
        "nominalrate": 810794.3180662441,
        "nominalrate_bytes": 75422584.54486898,
```

```
        "numtask": 500,
        "parentsIds": [],
        "recordsread": 597900000.0,
        "recordswrite": 0.0,
        "shufflebytesread": 0.0,
        "shufflebyteswrite": 0.0,
        "shufflerecordsread": 0.0,
        "shufflerecordswrite": 0.0,
        "skipped": false,
        "t_record_ta_executor": 0.001233358421140659,
        "weight": 1.9935654473336273
    },
    "2": {
        "RDDIds": {
            "5": {
                "callsite": "textFile at PromoCalls.java:44",
                "name": "hdfs://10.0.0.4:9000//user/ubuntu/
                    last24HoursAbroadCalls.txt"
            },
            "6": {
                "callsite": "textFile at PromoCalls.java:44",
                "name": "hdfs://10.0.0.4:9000//user/ubuntu/
                    last24HoursAbroadCalls.txt"
            },
            "7": {
                "callsite": "filter at PromoCalls.java:45",
                "name": "MapPartitionsRDD"
            }
        },
        "actual_records_read": 597900000.0,
        "actual_records_write": 597900000.0,
        "bytesread": 55618499408.0,
        "byteswrite": 0.0,
        "cachedRDDs": [],
        "duration": 13000.0,
        "genstage": false,
        "id": 2.0,
        "io_factor": 1.0,
        "monocoreduration": 727935.0,
        "name": "count at PromoCalls.java:45",
        "nominalrate": 821364.5449112902,
        "nominalrate_bytes": 76405859.60010166,
        "numtask": 500,
        "parentsIds": [],
        "recordsread": 597900000.0,
        "recordswrite": 0.0,
        "shufflebytesread": 0.0,
        "shufflebyteswrite": 0.0,
        "shufflerecordsread": 0.0,
        "shufflerecordswrite": 0.0,
        "skipped": false,
        "t_record_ta_executor": 0.001217486201705971,
```

```
        "weight": 1.0
    }
}
```

The profiling information for a *xSpark*$_{\text{SEEPEP}}$ application is obtained by combining the JSON profiles, obtained by driving the application with different sets of input data so to drive the execution of all the possible execution paths, into a JSON file that we will call with a jargon "JumboJSON", as shown in Figure 5.2. Futhermore, each single

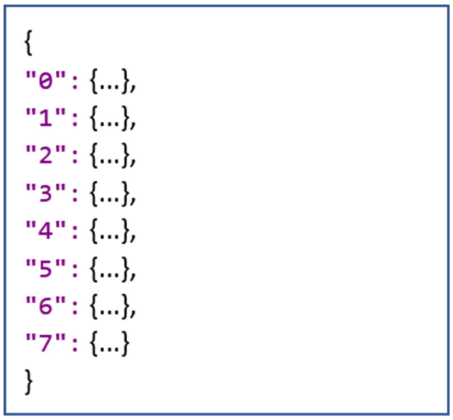

Figure 5.2: Structure of profile JumboJSON.

json profile is enhanced with information about the jobs composing the application, as shown in Figure 5.3. Inside *xSpark*$_{\text{SEEPEP}}$ is kept a symbolic memory store that maps symbolic values (or symbols) to actual values. To this structure, initially empty, a new entry is added every time a symbolic value gets assigned a concrete value. Each entry is a key-value pair containing the symbol as key and the assigned value as value. The convention adopted for naming the symbols is the following:

- **Commandline arguments**: prefix "arg_" followed by an integer reflecting the position of the argument on the commandline. For example: "arg_0", "arg_1" etc...
- **Program variables**: Spark action name followed by "_", followed by program name followed by ":", followed by the program line number where the action is called, followed by "_", followed by an integer representing the number of times the action in the same line of code is being repeated. For example: "count_PromoCalls.java:34_2"

Table 5.1 shows an example of symbolic memory store contents during the execution of the application, run with three commandline arguments having value "100", "200", "300" and two spark actions already

```
{
    "0": {
        "0": {
            "jobs": {
                "0": {
                    "id-symb": "count_PromoCalls.java:34_0",
                    "stages": [
                        0
                    ]
                },
                "1": {
                    "id-symb": "count_PromoCalls.java:42_0",
                    "stages": [
                        1
                    ]
                },
                "2": {
                    "id-symb": "count_PromoCalls.java:45_0",
                    "stages": [
                        2
                    ]
                },
                "3": {
                    "id-symb": "collect_PromoCalls.java:51_0",
                    "stages": [
                        3
                    ]
                },
                "4": {
                    "id-symb": "collect_PromoCalls.java:52_0",
                    "stages": [
                        4
                    ]
                },
                "5": {
                    "id-symb": "collect_PromoCalls.java:68_0",
                    "stages": [
                        5
                    ]
                }
            },
            …
        },
    …
}
```


Figure 5.3: Information about jobs in json DAG profile.

Table 5.1: Example of Symbolic Memory Store contents.

Entry#	Key	Value
0	arg_0	100
1	arg_1	200
2	arg_2	300
3	count_PromoCalls.java:42_0	2350
4	count_PromoCalls.java:45_0	1920
5	count_PromoCalls.java:45_1	3800

executed, of which the second was executed twice. The application is also required to provide a class implementing a method called `evaluateGuards` that receives in input a Map of a symbolic memory store (as the one described above) and returns a list of the profile id's whose DAG's are still executable (i.e. they contain execution paths whose Path Conditions are satisfiable).

5.2.3 *A new Heuristic*

xSpark$_{\text{SEEPEP}}$ implements `HeuristicSymExControlUnlimited`, a new heuristic that extends `HeuristicControlUnlimited`. Figure 6.1 below shows the simplified class diagram showing the relationships between the heuristic classes in the spark.deploy.control package. A new method, `nextProfile`, is implemented by the new heuristic, that takes as input parameters: 1) a json containing the application profiles obtained by the concrete execution of every possible execution path of the application; 2) a list of the profile id's that are still satisfiable; 3) the id of the job being submitted; and returns the json of the application profile to be used during the execution of the next job. Keeping in

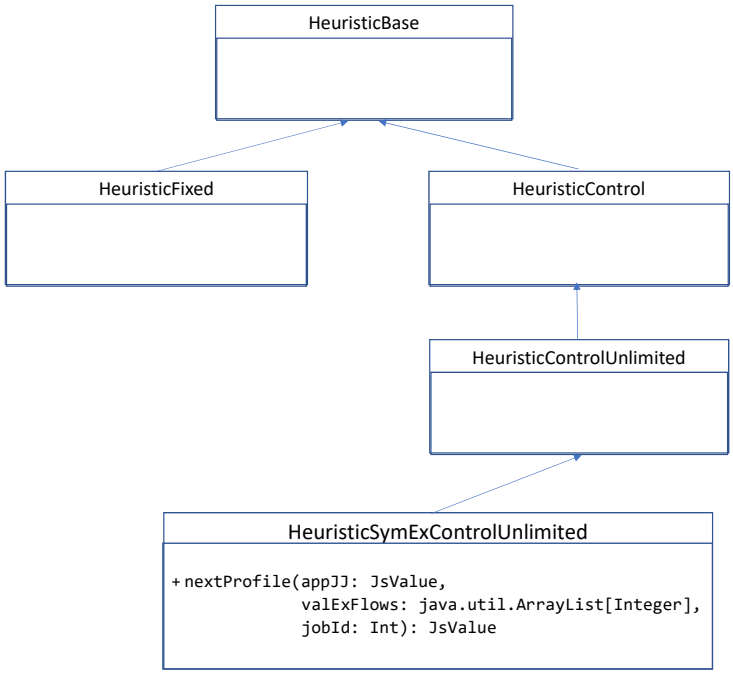

Figure 5.4: *xSpark*$_{\text{SEEPEP}}$ Heuristic related simplified class diagram.

mind that the scheduler uses the `HeuristicSymExControlUnlimited` to estimate how many cores are needed to execute the stage, given the application deadline and the parameters in the application profile, we expect the new heuristic to choose the profile so as not to jeopardize the controller's ability to meet the deadline. This can be achieved by

choosing a profile that will lead to not underestimate the cores needed to execute the remaining stages. All the following parameters seem to be good proxies for estimating the remaining computing effort:

```
1) Number of remaining stages to be executed
2) Sum of duration of remaining stages to be executed
3) Weighted combination of 1 and 2 above
```

The above parameters can be calculated using the data inside the application json profile. Current implementation of the heuristic uses proxy #1 (number of remaining stages to be executed). It calculates the value for each of the satisfiable profiles, and then selects the profile associated to the maximum value of the proxy. This way, we supply the "worst case" profile to the heuristic, so that the stage deadline is not overestimated and consequently the number of cores to be assigned for the next stage execution is not underestimated.

5.3 APPLICATION PARAMETERS

As explained in Chapter 4, the application parameters can be part of a path condition as they could have been associated to a symbol by the symbolic executor part of *SEEPEP*. Hence, we have introduced in xSpark a mechanism to intercept and store these application parameters in the *xSpark*$_{\text{SEEPEP}}$ *Symbol Store*, as shown in Figure 4.4. Listing 5.2 shows part of the code of method *submit* of xSpark class *SparkSubmit*, that was modified in order to read the values of the application's runtime arguments passed via the Spark *submit* command and write them as separate lines to textfile *args.txt*. This file is a component of the *PEP* Store*. Records from this file are read by xSpark at a later stage, when lazily executing the application by means of job scheduling.

Listing 5.2: Changes to SparkSubmit method "submit".

```
1  private def submit(args: SparkSubmitArguments): Unit = {
2      val (childArgs, childClasspath, sysProps, childMainClass) =
           prepareSubmitEnvironment(args)
3      val argsFile = sys.env.getOrElse("SPARK_HOME", ".") + "/conf/
           args.txt"
4      val bw = new BufferedWriter(new FileWriter(argsFile))
5      if (childArgs.size > 0) {
6        bw.write(childArgs(0) + "\n")
7        bw.write(args.primaryResource + "\n")
8        for ( i <- 1 to childArgs.size - 1 ) {
9          bw.write(childArgs(i)+"\n")
10       }
11     }
12     bw.close()
13     ...
```

5.4 APPLICATION PROFILING

As shown in Figure 4.4, for each set of input parameters identified by *SEEPEP* a *Launcher* is generated. A *Launcher* is a Java class which contains the command to run the application with a specific set of arguments, which is in a 1:1 relationship with the application parameters of the corresponding *PEP*. An example of Launcher class is shown in Listing 5.3.

A *Profiling* (see Figure 3.1) of the application is done by running it with each *Launcher*'s set of arguments. Each profiling run generates the corresponding *PEP* in a specialized JSON file. At the end of this process, all the generated *PEPs* are packaged into another JSON file, in jargon called *JumboJSON*, to form the PEP*. All the generated JSON files are then stored along with the PEP* *Store* on the *Spark Master* server.

Listing 5.3: Example of Launcher Code .

```
1  package it.polimi.deepse.dagsymb.launchers;/*
2   * This file was automatically generated by EvoSuite
3   * Wed May 16 13:17:45 GMT 2018
4   */
5
6  import it.polimi.deepse.dagsymb.examples.PromoCalls;
7  import it.polimi.deepse.dagsymb.examples.UserCallDB;
8
9  public class Launcher0 {
10
11   //Test case number: 0
12   /*
13    * 1 covered goal:
14    * Goal 1. com.xspark.varyingdag.examples.calls.PromoCalls.
          run_driver(IJJI)V: path condition EvoSuiteWrapper_0_0 (id
          = 0)
15    */
16
17     public static void main(String[] args) {
18       int threshold = 2772;
19       long minLocalLongCalls = 2772;
20       long minAbroadLongCalls = 1397;
21       int pastMonths = 0;
22       int last24HLocalCallsLength = 0;
23       int last24HLocalCallsSize = 0;
24       int last24HAbroadCallsLength = 3361;
25       int last24HAbroadCallsSize = 2794; // 1397 * 2
26       int MonthCallsLength = 2990;
27       int MonthCallsSize = 3000;
28       int num_partitions = 500;
29       PromoCalls promoCalls0 = new PromoCalls();
30       boolean genData = false;
31       String appName = "";
```

```
32      if (args[12] != null && args[12].startsWith("-g")) genData
            = true;
33      if (args[13] != null && !args[13].startsWith("-")) appName
            = args[12];
34      //UserCallDB.addCallsToLast24HoursAbroadCalls(3361, 1397);
35      //UserCallDB.addCallsToLast24HoursAbroadCalls(3361, 1397);
36      //promoCalls0.run(2772, 2772, 1397, 0);
37      promoCalls0.run(threshold, minLocalLongCalls,
            minAbroadLongCalls, pastMonths,
38                  last24HLocalCallsLength, last24HLocalCallsSize,
39                  last24HAbroadCallsLength, last24HAbroadCallsSize,
40                  MonthCallsLength, MonthCallsSize, num_partitions,
                        genData, appName);
41  }
42 }
```

5.5 PEP*

The previous version of xSpark required a single *PEP* to be present in the *PEP Store*, so we had to modify the xSpark class *DAGScheduler* to account for the data in the *JumboJSON file* that now contains all the *PEPs*. To maintain backwards compatibility, a new variable was introduced, `JumboJson`, to hold the contents of the PEP* Store. The modified code is in charge of checking the *heuristic type* in the *SparkContext* instance `sc`, to understand if it should expect the contents of the PEP* Store to be a single *PEP* or a whole set of *PEPs* representing a PEP*. The heuristic type is initialized with the value of the key *spark.control.heuristic* specified in the xSpark configuration file *spark-defaults.conf*.

Listing 5.4: Changes to class DAGScheduler.scala - reading PEPs.

```
1   val jsonFile = sys.env.getOrElse("SPARK_HOME", ".") +
2     "/conf/" + sc.appName + ".json"
3
4   val appJumboJson = if (Files.exists(Paths.get(jsonFile))) {
5     io.Source.fromFile(jsonFile).mkString.parseJson
6   } else null
7
8   var appJson = if (appJumboJson != null && heuristicType > 2)
9     heuristic.nextProfile(appJumboJson)
10    else appJumboJson
```

5.6 GUARDEVALUATOR

With the term *GuardEvaluator* we collectively refer to the interface class `IGuardEvaluator` and its implementation class defining the `evaluateGuards` method, that is in charge of returning the list of

valid profiles (*PEPs*) when it is called with the HashMap of the known symbols and their values. The code of the `IGuardEvaluator` interface is shown in Listing 5.5, while Listing 5.6 shows an example of an implementation class and its method `evaluateGuards` for a specific application.

Listing 5.5: Interface class IGuardEvaluator.

```
package it.polimi.deepse.dagsymb.examples;
import java.util.List;
import java.util.Map;

public interface IGuardEvaluator {

  public List<Integer> evaluateGuards(Map<String, Object>
      knownValues);

}
```

Listing 5.6: Class GuardEvaluatorPromoCallsFile, implementing the IGuardEvaluator interface.

```
package it.polimi.deepse.dagsymb.examples;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class GuardEvaluatorPromoCallsFile implements
    IGuardEvaluator {

    private List<Integer> satisfiableGuards;

    @Override
    public List<Integer> evaluateGuards(Map<String, Object>
        knownValues) {
        satisfiableGuards = new ArrayList<>();

        extractValues(knownValues);

        evaluateActualGuards();

        return satisfiableGuards;
    }

    private void evaluateActualGuards() {
        //path condition evaluation
        if (
           ( !arg0_known ||   arg0 > 100 ) &&
           ( !arg3_known ||   arg3 >= 0 ) &&
           ( !arg3_known ||   arg3 <= 2 ) &&
           ( !count_PromoCalls_java_42_0_known || !arg1_known ||
                count_PromoCalls_java_42_0 - arg1 <= 0 ) &&
```

```
28            ( !arg2_known || !count_PromoCalls_java_45_0_known ||
                  count_PromoCalls_java_45_0 - arg2 > 0 ) &&
29        true) {
30            satisfiableGuards.add(0);
31        }
32
33        if (
34           ( !arg0_known ||   arg0 > 100 ) &&
35           ( !arg3_known ||   arg3 >= 0 ) &&
36           ( !arg3_known ||   arg3 <= 2 ) &&
37           ( !count_PromoCalls_java_42_0_known || !arg1_known ||
                  count_PromoCalls_java_42_0 - arg1 <= 0 ) &&
38           ( !arg2_known || !count_PromoCalls_java_45_0_known ||
                  count_PromoCalls_java_45_0 - arg2 <= 0) &&
39        true) {
40            satisfiableGuards.add(1);
41        }
42
43        if (
44           ( !arg0_known ||   arg0 > 100 ) &&
45           ( !arg3_known ||   arg3 >= 0 ) &&
46           ( !arg3_known ||   arg3 <= 2 ) &&
47           ( !count_PromoCalls_java_42_0_known || !arg1_known ||
                  count_PromoCalls_java_42_0 - arg1 > 0 ) &&
48           ( !arg3_known ||   1 > arg3 ) &&
49           ( !arg2_known || !count_PromoCalls_java_45_0_known ||
                  count_PromoCalls_java_45_0 - arg2 > 0 ) &&
50        true) {
51            satisfiableGuards.add(2);
52        }
53
54        if (
55           ( !arg0_known ||   arg0 > 100 ) &&
56           ( !arg3_known ||   arg3 >= 0 ) &&
57           ( !arg3_known ||   arg3 <= 2 ) &&
58           ( !count_PromoCalls_java_42_0_known || !arg1_known ||
                  count_PromoCalls_java_42_0 - arg1 > 0 ) &&
59           ( !arg3_known ||   1 > arg3 ) &&
60           ( !arg2_known || !count_PromoCalls_java_45_0_known ||
                  count_PromoCalls_java_45_0 - arg2 <= 0) &&
61        true) {
62            satisfiableGuards.add(3);
63        }
64
65        if (
66           ( !arg0_known ||   arg0 > 100 ) &&
67           ( !arg3_known ||   arg3 >= 0 ) &&
68           ( !arg3_known ||   arg3 <= 2 ) &&
69           ( !count_PromoCalls_java_42_0_known || !arg1_known ||
                  count_PromoCalls_java_42_0 - arg1 > 0 ) &&
70           ( !arg3_known ||   1 <= arg3 ) &&
71           ( !arg3_known ||   2 > arg3 ) &&
```

```
72            ( !arg2_known || !count_PromoCalls_java_45_0_known ||
                  count_PromoCalls_java_45_0 - arg2 > 0 ) &&
73        true) {
74            satisfiableGuards.add(4);
75        }
76
77        if (
78           ( !arg0_known ||   arg0 > 100 ) &&
79           ( !arg3_known ||   arg3 >= 0 ) &&
80           ( !arg3_known ||   arg3 <= 2 ) &&
81           ( !count_PromoCalls_java_42_0_known || !arg1_known ||
                  count_PromoCalls_java_42_0 - arg1 > 0 ) &&
82           ( !arg3_known ||   1 <= arg3 ) &&
83           ( !arg3_known ||   2 > arg3 ) &&
84           ( !arg2_known || !count_PromoCalls_java_45_0_known ||
                  count_PromoCalls_java_45_0 - arg2 <= 0) &&
85        true) {
86            satisfiableGuards.add(5);
87        }
88
89        if (
90           ( !arg0_known ||   arg0 > 100 ) &&
91           ( !arg3_known ||   arg3 >= 0 ) &&
92           ( !arg3_known ||   arg3 <= 2 ) &&
93           ( !count_PromoCalls_java_42_0_known || !arg1_known ||
                  count_PromoCalls_java_42_0 - arg1 > 0 ) &&
94           ( !arg3_known ||   1 <= arg3 ) &&
95           ( !arg3_known ||   2 <= arg3 ) &&
96           ( !arg2_known || !count_PromoCalls_java_45_0_known ||
                  count_PromoCalls_java_45_0 - arg2 > 0 ) &&
97        true) {
98            satisfiableGuards.add(6);
99        }
100
101       if (
102          ( !arg0_known ||   arg0 > 100 ) &&
103          ( !arg3_known ||   arg3 >= 0 ) &&
104          ( !arg3_known ||   arg3 <= 2 ) &&
105          ( !count_PromoCalls_java_42_0_known || !arg1_known ||
                  count_PromoCalls_java_42_0 - arg1 > 0 ) &&
106          ( !arg3_known ||   1 <= arg3 ) &&
107          ( !arg3_known ||   2 <= arg3 ) &&
108          ( !arg2_known || !count_PromoCalls_java_45_0_known ||
                  count_PromoCalls_java_45_0 - arg2 <= 0) &&
109       true) {
110           satisfiableGuards.add(7);
111       }
112
113   }
114
115   private boolean arg0_known; private Integer arg0;
```

```
116     private boolean count_PromoCalls_java_42_0_known; private
            Long count_PromoCalls_java_42_0;
117     private boolean arg2_known; private Long arg2;
118     private boolean arg1_known; private Long arg1;
119     private boolean count_PromoCalls_java_45_0_known; private
            Long count_PromoCalls_java_45_0;
120     private boolean arg3_known; private Integer arg3;
121
122     private void extractValues(Map<String, Object> knownValues) {
123         arg0_known = (knownValues.get("arg0") != null);
124         arg0 = arg0_known ? Integer.parseInt((String) knownValues
                .get("arg0")) : null;
125
126         count_PromoCalls_java_42_0_known = (knownValues.get("
                count_PromoCalls.java:42_0") != null);
127         count_PromoCalls_java_42_0 =
                count_PromoCalls_java_42_0_known ? (Long) knownValues
                .get("count_PromoCalls.java:42_0") : null;
128
129         arg2_known = (knownValues.get("arg2") != null);
130         arg2 = arg2_known ? Long.parseLong((String) knownValues.
                get("arg2")) : null;
131
132         arg1_known = (knownValues.get("arg1") != null);
133         arg1 = arg1_known ? Long.parseLong((String) knownValues.
                get("arg1")) : null;
134
135         count_PromoCalls_java_45_0_known = (knownValues.get("
                count_PromoCalls.java:45_0") != null);
136         count_PromoCalls_java_45_0 =
                count_PromoCalls_java_45_0_known ? (Long) knownValues
                .get("count_PromoCalls.java:45_0") : null;
137
138         arg3_known = (knownValues.get("arg3") != null);
139         arg3 = arg3_known ? Integer.parseInt((String) knownValues
                .get("arg3")) : null;
140
141     }
142 }
```

The application is in charge of providing the *GuardEvaluator* as a java class, implementing the interface `IGuardEvaluator`, and packaged inside the jar of the application. This class is loaded dynamically at runtime by the new code added for this purpose to the xSpark class `DAGScheduler` and shown in Listing 5.7.

Listing 5.7: Changes to class DAGScheduler.scala - Loading GuardEvaluator.

```
1 var guardEvalObj:Any = null
2 var guardEvalMethod: java.lang.reflect.Method = null
3 if (heuristicType > 2) {
4   /*
5    * DB - DagSymb enhancements
```

```
6     * The following variables are needed to load the
          GuardEvaluator class from the application jar
7     */
8    val jarfile = new File(appJar)
9    val classLoader = new URLClassLoader(Array(jarfile.toURI.toURL)
         )
10   val guardEvalClass = classLoader.loadClass(guardEvalClassname)
11   val guardEvalConstructor = guardEvalClass.getConstructor()
12   guardEvalObj = guardEvalConstructor.newInstance()
13   val methods = guardEvalClass.getDeclaredMethods
14   for (m <- methods) {
15     m.getName match {
16       case "evaluateGuards" => guardEvalMethod = m
17       case _ =>
18     }
19   }
20 } else {
21   guardEvalObj = new core.src.main.scala.org.apache.spark.
         scheduler.GuardEvaluator
22 }
```

5.7 SYMBOL STORE

In order to take advantage of the symbolic execution, we need to maintain an updated *Symbol Store* containing all the symbols that can be part of a path condition and their associated determinations (assigned values). We added the code into the xSpark class `DAGScheduler` to abstractly represent the *Symbol Store* as a *HashMap[String, Any]*. Each entry of this HashMap stores a *known symbol* name and its value. By *known symbol* we mean a symbol that has been associated to a value during the concrete execution of program code. Given this defintion, at the very beginning of the computation the only known symbols are the runtime arguments passed to the application. We added to the xSpark class `DAGScheduler` the code to read the arguments and their values and create the corresponding *Symbol Store* entries. The code is shown in Listing 5.8, where we can notice that the first two arguments loaded when var *iter* is set to negative values (-2 and -1) are respectively the *GuardEvaluator* class name and the application jar name, that are not symbols. They are not kept in the Symbol Store, instead they are used to initialize the variables `guardEvalClassname` and `appJar` which will be needed in a later step of the execution to identify the location and dynamically load the class implementing the *GuardEvaluator* function.

Listing 5.8: Changes to class DAGScheduler.scala - Initializing Symbol Store.

```
1 var symbolsMap = new java.util.HashMap[String, Any]()
2 var symbolName: String = ""
3 var guardEvalClassname: String = ""
```

```
var appJar: String = ""
val argsFile = sys.env.getOrElse("SPARK_HOME", ".") +
    "/conf/args.txt"
var iter: Int = -2
if (Files.exists(Paths.get(argsFile))) {
for (line <- Source.fromFile(argsFile).getLines) {
    iter match {
      case -2 => guardEvalClassname = line
      case -1 => appJar = line.split(":")(1)
      case _  => symbolsMap.put("arg" + iter, line)
    }
    iter += 1
}
...
```

5.8 HEURISTIC

The heuristic used by xSpark is determined by the value of configuration parameter spark.control.heuristic and is an implementation of the class `HeuristicBase`, whose class diagram is shown in Figure 5.5. In Listing 5.9, we can see that the heuristic *HeuristicControl* is used by default, but other heuristics can be selected. `HeuristicFixed` and `HeuristicControlUnlimited` were already available in xSpark, while we implemented a new heuristic **`HeuristicSymExControlUnlimited`** to exploit *Symbolic Execution*.

Listing 5.9: Changes to class ControlEventListener.scala - selecting the heuristic.

```
val heuristicType = conf.getInt("spark.control.heuristic", 0)
val heuristic: HeuristicBase =
  if (heuristicType == 1 &&
      conf.contains("spark.control.stagecores") &&
      conf.contains("spark.control.stagedeadlines") &&
      conf.contains("spark.control.stage"))
    new HeuristicFixed(conf)
  else if (heuristicType == 2)
    new HeuristicControlUnlimited(conf)
  else if (heuristicType == 3)
    new HeuristicSymExControlUnlimited(conf)
  else
    new HeuristicControl(conf)
```

`HeuristicSymExControlUnlimited` extends `HeuristicControlUnlimited` by adding the implementation of a new method, `nextProfile`, taking parameters `appJson`, the `JumboJSON` containing the PEP* representation and `valExFlows`, a list containing the id's of the valid application profiles,(i.e. the list of the *PEPs* whose path conditions still hold true), and returns the *PEP* of the profile to be used during the executing of the next scheduled job.

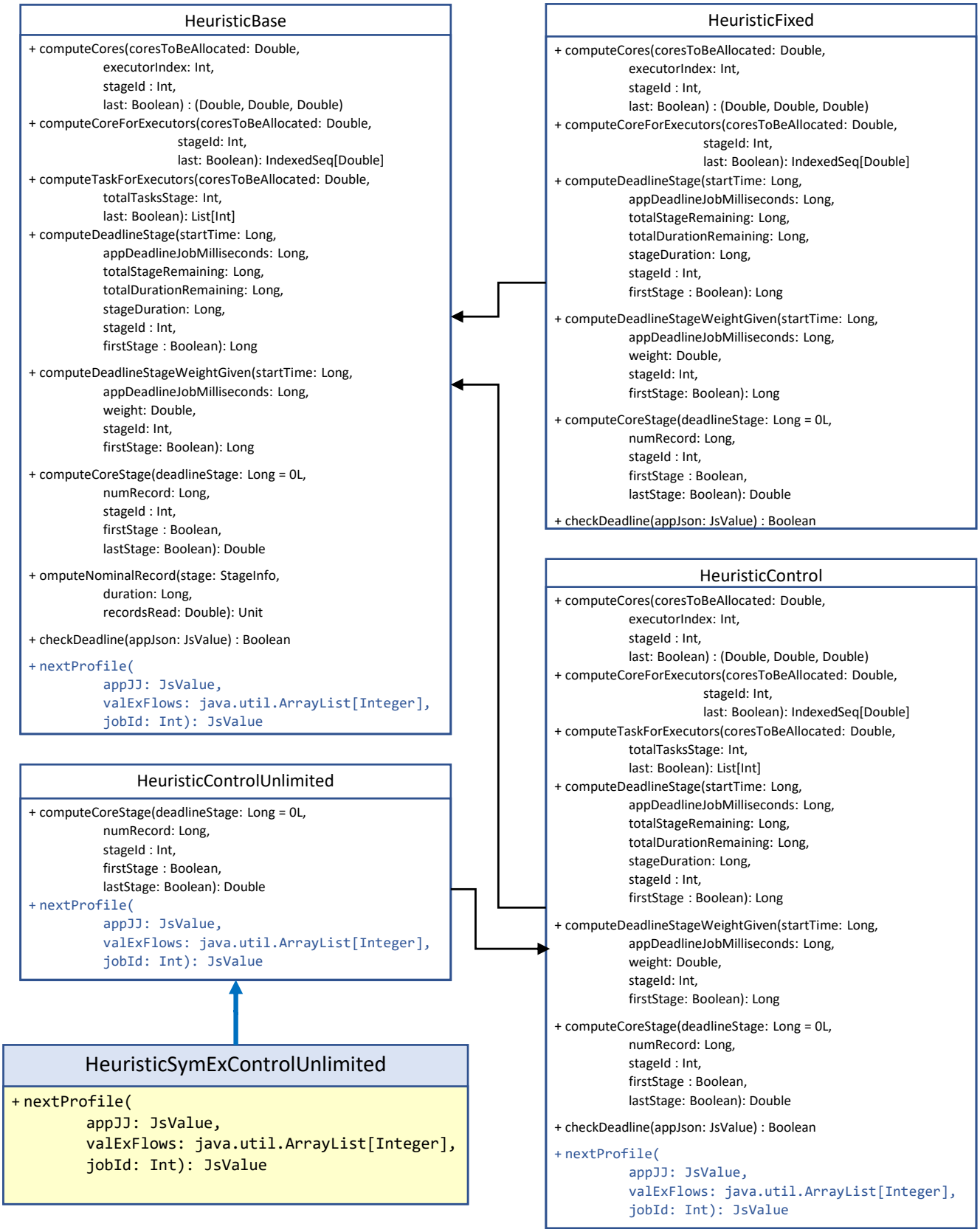

Figure 5.5: Class diagram of Heuristic related classes.

Listing 5.10: Class HeuristicSymExControlUnlimited.scala implementation.

```
class HeuristicSymExControlUnlimited(conf: SparkConf)
                      extends HeuristicControlUnlimited(conf) {
  override def nextProfile(appJJ: JsValue,
      valExFlows: java.util.ArrayList[Integer] = null,
      jobId: Int = 0): JsValue = {
    var setP = appJJ.asJsObject.fields
    val stageId = if (valExFlows != null)
        setP(valExFlows.get(0).toString()).asJsObject.fields("o")
        .asJsObject.fields("jobs")
        .asJsObject.fields(jobId.toString())
        .asJsObject.fields("stages")
        .convertTo[List[Int]].sortWith((x, y) => x < y).apply(0)
    if (valExFlows != null)
      setP = setP.filter(
        {case (k,v) => valExFlows.exists(x => x == k.toInt)})
      var wCaseProfId = setP.toList.map({ case (k, profile) => {
        val numStage = profile.asJsObject.fields.filter(
              {case (k, stage) => {
              !stage.asJsObject.fields("skipped")
              .convertTo[Boolean]}})
        .size()
        (k, numStage)}}).reduce({ (x, y) => {
          if (x._2 > y._2)  x else y
         }})._1;
    setP(wCaseProfId)
  }
}
```

Class `HeuristicSymExControlUnlimited` implementation code is shown in Listing 5.10. The *PEP* selection is made by choosing the *"worst case"* among the valid *PEPs*, that is the *PEP* with the maximum number of stages still to be executed, as we want to be conservative and minimize the deadline violations. If we wanted to optimize another performance indicator, like minimum resource utilization in absence of strict deadline commitment, we could choose the profile with an average number of remaining stages to be executed.

5.9 SYMBOLS

As mentioned in Section 5.7, we have to update the Symbol Store everytime a variable associated to a symbol is evaluated by the concrete execution of the application. We adopted the convention to identify a symbol by the string *arg_n* if it refers to a runtime application argument, where n is the position of the argument (e.g. *arg_0*), or by a string obtained by concatenating its *CallSite* and *IterationNumber* separated by an underscore character *"_"*, where *Call Site* is a string obtained by concatenating *SparkActionName*, *ApplicationClassName*, *SourceLanguageName:SourceLineNumber* separated by an underscore character *"_"*,

and *IterationNumber* is an integer starting from 0 and incremented everytime a call is originated by the same *CallSite*. i.e. the same line of code is re-executed (e.g. due to iterative loops). Examples of formal symbol names are shown in Figure 5.6.

```
arg_0
count_PromoCalls_java:45_2
```

Figure 5.6: Example of symbol formal names.

As stated above, a symbol is identified by its *CallSite*. This means that to identify the symbols we have to intercept calls coming from their *CallSite*. Since the values associated to a symbol can only be changed by xSpark *actions* which delimit jobs, we added code to the method `runJob` of class `DAGScheduler` to extract the *CallSite*, generate a symbol and push it in the *Symbol Store* everytime the method `runJob` is called. The code is shown in Listing 5.11.

Listing 5.11: Changes to class DAGScheduler.scala - method runJob.

```
1  var symbolMap = HashMap[String, Int]()
2  var symbolName: String = ""
3  def runJob[T, U](
4      rdd: RDD[T],
5      func: (TaskContext, Iterator[T]) => U,
6      partitions: Seq[Int],
7      callSite: CallSite,
8      resultHandler: (Int, U) => Unit,
9      properties: Properties): Unit = {
10   val actionCallSite = callSite.shortForm.replace(" at ", "_")
11   symbolName = actionCallSite + "_" + symbolMap.getOrElseUpdate(
         actionCallSite, 0).toString()
12   symbolsMap.put(symbolName, null)
13   symbolMap(actionCallSite) += 1
14   ...
```

The variable `actionCallSite` is initialized using the value of the `callSite` parameter. An auxilliary structure, the *HashMap[String, Int]* `symbolMap` keeps track of every `actionCallSite` and counts how many times each of them has called the method runJob. The value of the count determines the suffix of each symbol. Symbols are initially assigned a `null` value, and stay in the symbol store waiting to be assigned the result of the *action* originated from the `actionCallSite`.

This task is performed by the new method `resultComputed`, that was added to the xSpark class `DAGScheduler`. `resultComputed` is called by the homonymous method, added to the class `SparkContext`, that passes to it the computed result of the action received from an *action* method in the class `RDD`. Lastly, we have indeed modified the methods that execute the *action*s in the `RDD` xSpark class by inserting a call to the

method `resultComputed` of the `SparkContext` instance `sc` and passing to it the computed result of the action.

Updating the value of *symbols* in the *Symbol Store* is not the only task fulfilled by the `resultComputed` method of class `DAGScheduler`. It also executes the method stored in the variable `guardEvalMethod` with the map of the known *symbols* getting in return the variable `new_validExecFlows` containing the list of valid profiles. It is also in charge of selecting the profile `appJson` to be used to run the next job. It fulfills this task by calling the method `nextProfile` of the `HeuristicSymExControlUnlimited` instance `heuristic` and passing to it the list of valid profiles. In returns, it gets the json profile to be used in the next job run, stored in the variable appJson.

The new and modified methods are shown in Listings 5.12, 5.13, 5.14.

Listing 5.12: Changes to class DAGScheduler.scala - new method resultComputed.

```
private[scheduler] def numTotalJobs: Int = nextJobId.get()

def resultComputed(result: Any ): Unit = {
  if (heuristicType > 2) {
    symbolsMap(symbolName) = result
    val resultType = ClassTag(result.getClass)
    var new_validExecFlows = guardEvalMethod.invoke(guardEvalObj,
        symbolsMap).asInstanceOf[java.util.ArrayList[Integer]]
    if (new_validExecFlows.size() > 0)
      validExecFlows = new_validExecFlows
    else
      println("Warning! GuardEvaluator returned an empty set of
          profile ids")
    val highestJobId: Int =
      if (validExecFlows != null) {
        appJumboJson.asJsObject.fields(validExecFlows.get(0).
            toString())
        .asJsObject.fields("o").asJsObject.fields("jobs")
        .asJsObject.fields.keys.max.toInt}
      else 0
    appJson = if (numTotalJobs <= highestJobId) {
                heuristic.nextProfile(appJumboJson,
                    validExecFlows, nextJobId.get())}
              else appJson
```

Listing 5.13: Changes to class SparkContext.scala - new method resultComputed.

```
  def resultComputed(result: Any ): Unit = {
      dagScheduler.resultComputed(result)
      }
```

Listing 5.14: Changes to class RDD.scala - modified methods count, collect and reduce.

```
def count(): Long = {
val res = sc.runJob(this, Utils.getIteratorSize _).sum
sc.resultComputed(res)
res
}

def collect(): Array[T] = withScope {
val results = sc.runJob(this, (iter:Iterator[T]) => iter.toArray)
val res = Array.concat(results: _*)
sc.resultComputed(res)
res
}

def reduce(f: (T, T) => T): T = withScope {
val cleanF = sc.clean(f)
val reducePartition: Iterator[T] => Option[T] = iter => {
  if (iter.hasNext) {
    Some(iter.reduceLeft(cleanF))
  } else {
    None
  }
}
var jobResult: Option[T] = None
val mergeResult = (index: Int, taskResult: Option[T]) => {
  if (taskResult.isDefined) {
    jobResult = jobResult match {
      case Some(value) => Some(f(value, taskResult.get))
      case None => taskResult
    }
  }
}
sc.runJob(this, reducePartition, mergeResult)
jobResult.getOrElse(throw new UnsupportedOperationException("
    empty collection"))
val res = jobResult.getOrElse(throw new
    UnsupportedOperationException("empty collection"))
sc.resultComputed(res)
res
}
```

5.10 SCHEDULING JOBS

Jobs in xSpark are delimited by *actions* (e.g. *count(), collect() etc...)* and are composed by a number of stages, that reflect the *transformations* operated on data. Data are abstracted as Resilient Distributed Datasets (*RDD's* structures. As explained in Chapter 4, the application parameters can be part of a path condition as they could have been associated to a symbol by the symbolic executor part of *SEEPEP*. Hence, we have

introduced in xSpark a mechanism to intercept and store these application parameters in the *xSpark*$_{\text{SEEPEP}}$ *Symbol Store*, as it was shown in Figure 4.4. The code listed in Listing **??** was added to method *submit* in xSpark module *SparkSubmit* to read the values of the application's arguments passed via the Spark *submit* command and write them as separate lines to textfile *args.txt*.

5.11 GETTING RESULTS OF ACTIONS

6 EVALUATION

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

— Verne *Journey to the Center of the Earth* 1957

EVALUATION - sezione su esperimenti

6.1 EXPERIMENTS

7 CONCLUSION

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

— Verne *Journey to the Center of the Earth* 1957

Conclusioni e future work

7.1 CONCLUSION

7.2 FUTURE WORK

BIBLIOGRAPHY

[1] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics." In: *14th Symposium on Networked Systems Design and Implementation)*. USENIX Association, 2017 (cit. on p. 4).

[2] *Apache Flink.* 2019 (cit. on p. 18).

[3] *Apache Flink Architecture.* 2019 (cit. on p. 18).

[4] *Apache Spark.* 2019 (cit. on pp. 2, 14).

[5] *Apache Storm.* 2019 (cit. on p. 19).

[6] *Apache Tez.* 2019 (cit. on p. 2).

[7] Apache.org. *RDD Programming Guide - Spark 2.4.0 documentation*. https://spark.apache.org/docs/latest/rdd-programming-guide.html. Accessed: 2019-02-20 (cit. on pp. 69, 73).

[8] Anonymous Author(s. "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 2019)*. (2019), pp. 1–11 (cit. on pp. 6, 7).

[9] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. "AEG: Automatic Exploit Generation." In: *Proc. Network and Distributed System Security Symp.* NDSS'11. 2011 (cit. on pp. 49, 51).

[10] *AWS.* 2019 (cit. on pp. 2, 10).

[11] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques." In: *ACM Comput. Surv.* 51.3 (May 2018), 50:1–50:39. ISSN: 0360-0300 (cit. on p. 40).

[12] L. Baresi and G. Quattrocchi. "Towards Vertically Scalable Spark Applications." In: *Proc. of Euro-Par 2018: Parallel Processing Workshops*. Springer, 2018 (cit. on pp. 5, 64).

[13] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi. *Fine-grained Dynamic Resource Allocation for Big-Data Applications*. Tech. rep. 2018 (cit. on pp. 5, 64).

[14] Clark Barrett, Daniel Kroening, and Thomas Melham. *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*. Knowledge Transfer Report, Technical Report 3. London Mathematical Society, Smith Institute for Industrial Mathematics, and System Engineering, 2014 (cit. on pp. 40, 42).

[15] *Big Data Companies 2019*. 2019 (cit. on pp. 2, 10).

[16] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. "Fast Unfolding of Communities in Large Networks." In: *Journal of statistical mechanics: theory and experiment* (2008) (cit. on p. 8).

[17] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution." In: *Proc. of Int. Conf. on Reliable Software*. Los Angeles, California: ACM, 1975, pp. 234–245 (cit. on p. 39).

[18] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. "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*. 2016, pp. 1018–1022 (cit. on p. 70).

[19] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. "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*. 2017, pp. 90–101 (cit. on p. 73).

[20] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. "SUSHI: a test generator for programs with complex structured inputs." In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018*. 2018, pp. 21–24 (cit. on p. 73).

[21] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs." In: *Proc. 8th USENIX Conf. on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224 (cit. on pp. 48, 51).

[22] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later." In: *Communications of the ACM* 56.2 (2013), pp. 82–90. ISSN: 0001-0782 (cit. on pp. 44, 49).

[23] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death." In: *Proc. 13th ACM Conf. on Computer and Communications Security*. CCS'06. Alexandria, Virginia, USA: ACM, 2006, pp. 322–335. ISBN: 1-59593-518-5 (cit. on p. 48).

[24] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code." In: *Proc. 2012 IEEE Symp. on Sec. and Privacy*. SP'12. IEEE Comp. Society, 2012, pp. 380–394. ISBN: 978-0-7695-4681-0 (cit. on pp. 48–51).

[25] Satish Chandra, Stephen J. Fink, and Manu Sridharan. "Snugglebug: A Powerful Approach to Weakest Preconditions." In: *Proc. 30th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI'09. Dublin, Ireland: ACM, 2009, pp. 363–374. ISBN: 978-1-60558-392-1 (cit. on p. 49).

[26] Ting Chen, Xiaodong Lin, Jin Huang, Abel Bacchus, and Xiaosong Zhang. "An Empirical Investigation into Path Divergences for Concolic Execution Using CREST." In: *Security and Communication Networks* 8.18 (2015), pp. 3667–3681. ISSN: 1939-0114 (cit. on p. 47).

[27] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications." In: *ACM Transactions on Computer Systems (TOCS)*. TOCS 2011 30.1 (2012), 2:1–2:49 (cit. on pp. 47, 48, 51).

[28] Peter Dinges and Gul Agha. "Targeted Test Input Generation Using Symbolic-concrete Backward Execution." In: *Proc. 29th ACM/IEEE Int. Conf. on Automated Software Engineering*. ASE'14. Vasteras, Sweden, 2014, pp. 31–36. ISBN: 978-1-4503-3013-8 (cit. on p. 49).

[29] *Docker Documentation*. 2019 (cit. on pp. 5, 38).

[30] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. "Principles of Elastic Processes." In: *IEEE Internet Computing* 15.5 (2011), pp. 66–71 (cit. on pp. 23, 25).

[31] *EMC*. 2019 (cit. on pp. 2, 10).

[32] Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation." In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 276–291 (cit. on p. 73).

[33] Giovanni Paolo Gibilisco et al. "Stage Aware Performance Modeling of DAG Based in Memory Analytic Platforms." In: *Proc. of IEEE 9th International Conference on Cloud Computing*. IEEE. 2016 (cit. on p. 64).

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI'05. Chicago, IL, USA, 2005, pp. 213–223. ISBN: 1-59593-056-6 (cit. on pp. 44, 45).

[35] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." In: *Proc. Network and Distributed System Security Symp.* NDSS'08. 2008 (cit. on pp. 45, 47, 51).

[36] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "SAGE: Whitebox Fuzzing for Security Testing." In: *Queue* 10.1 (2012), 20:20–20:27. ISSN: 1542-7730 (cit. on p. 40).

[37] *Google*. 2019 (cit. on pp. 2, 10).

[38] *Google Map-Reduce*. 2019 (cit. on p. 2).

[39] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. "Elasticity in Cloud Computing: What It Is, and What It Is Not." In: *ICAC*. 2013 (cit. on p. 23).

[40] William E. Howden. "Symbolic Testing and the DISSECT Symbolic Evaluation System." In: *IEEE Transactions on Software Engineering (TSE)* 3.4 (1977), pp. 266–278. ISSN: 0098-5589 (cit. on p. 39).

[41] *HPE*. 2019 (cit. on pp. 2, 10).

[42] *IBM*. 2019 (cit. on pp. 2, 10).

[43] *Internet Live Statisics*. 2019 (cit. on p. 1).

[44] M. Islam et al. "dSpark: Deadline-based Resource Allocation for Big Data Applications in Apache Spark." In: *Proc. of the 13th IEEE International Conference on eScience*. 2017 (cit. on pp. 4, 64, 65).

[45] Jeffrey O Kephart and David M Chess. "The Vision of Autonomic Computing." In: *Computer* 36 (2003) (cit. on p. 22).

[46] James C. King. "A New Approach to Program Testing." In: *Proc. Int. Conf. on Reliable Software*. Los Angeles, California: ACM, 1975, pp. 228–233 (cit. on p. 39).

[47] James C. King. "Symbolic Execution and Program Testing." In: *Communications of the ACM* 19.7 (1976), pp. 385–394. ISSN: 0001-0782 (cit. on p. 39).

[48] D. E. Knuth. "Computer Programming as an Art." In: *Communications of the ACM* 17.12 (1974), pp. 667–673 (cit. on p. 113).

[49] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap." In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Springer Berlin Heidelberg, 2013 (cit. on p. 22).

[50] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. "Steering Symbolic Execution to Less Traveled Paths." In: *Proc. ACM SIGPLAN Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA'13. 2013, pp. 19–32. ISBN: 978-1-4503-2374-1 (cit. on p. 48).

[51] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. "Directed Symbolic Execution." In: *Proc. 18th Int. Conf. on Static Analysis*. SAS'11. Venice, Italy, 2011, pp. 95–111. ISBN: 978-3-642-23701-0 (cit. on pp. 48, 50).

[52] Francesco Marconi, Giovanni Quattrocchi, Luciano Baresi, Marcello Bersani, and Matteo Rossi. "On the Timed Analysis of Big-Data Applications." In: *Proc. of the 10th NASA Formal Methods Symposium*. Springer, 2018 (cit. on p. 64).

[53] Patrick Wendell et al Matei Zaharia Reynold S Xin. "Apache Spark: A unified engine for big data processing." In: *Communications of the ACM* 59 (Nov. 2016), pp. 56–65 (cit. on p. 3).

[54] Phil McMinn. "Search-based Software Test Data Generation: A Survey." In: *Software Testing, Verification & Reliability* 14.2 (2004), pp. 105–156. ISSN: 0960-0833 (cit. on p. 49).

[55] *Microsoft.* 2019 (cit. on pp. 2, 10).

[56] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems,TACAS 2008*. 2008, pp. 337–340 (cit. on p. 70).

[57] Andrew Or. *Understanding your Apache Spark Application Through Visualization*. 2019 (cit. on p. 17).

[58] *Oracle.* 2019 (cit. on pp. 2, 10).

[59] Poornima Purohit, DR Apoorva, and et al PV Lathashree. "Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf." In: *Procedia Computer Science* 53 (Dec. 2015), pp. 121–130 (cit. on p. 2).

[60] *Running Spark on Mesos.* 2019 (cit. on p. 32).

[61] *Running Spark on YARN.* 2019 (cit. on p. 31).

[62] Mythreyee S, Poornima Purohit, Apoorva D.R., Harshitha R, and Lathashree P.V. "A Study on Use of Big Data in Cloud Computing Environment." In: *International Journal of Advance Research, Ideas and Innovations in Technology* 3 (2017), pp. 1312–1318 (cit. on pp. 2, 22).

[63] *SAP.* 2019 (cit. on pp. 2, 10).

[64] Anuj Sehgal, Vladislav Perelman, Siarhei Kuryla, and Jürgen Schönwälder. "Management of Resource Constrained Devices in the Internet of Things." In: *Communications Magazine, IEEE* 50 (2012) (cit. on p. 23).

[65] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *IEEE Symp. on Security and Privacy*. SP'16. 2016, pp. 138–157 (cit. on p. 40).

[66] S. Sidhanta, W. Golab, and S. Mukhopadhyay. "OptEx: A Deadline-Aware Cost Optimization Model for Spark." In: *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2016 (cit. on pp. 4, 64, 65).

[67] *Teradata.* 2019 (cit. on pp. 2, 10).

[68] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. "Apache Hadoop YARN: Yet Another Resource Negotiator." In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1 (cit. on p. 27).

[69] J. Verne. *Journey to the Center of the Earth*. Classics illustrated. Huge Print Press, 1957. ISBN: 9780758311993 (cit. on pp. 1, 9, 53, 63, 77, 101, 103).

[70] *VMware.* 2019 (cit. on pp. 2, 10).

[71] Danny Weyns, M. Usman Iftikhar, Sam Malek, and Jesper Andersson. "Claims and Supporting Evidence for Self-adaptive Systems: A Literature Study." In: *Proc. of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. Zurich, Switzerland: IEEE Press, 2012. ISBN: 978-1-4673-1787-0 (cit. on p. 22).

[72] *What is Big Data*. https://www.oracle.com/big-data/guide/what-is-big-data.html. 2019 (cit. on pp. 1, 9).

[73] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. "Fitness-guided path exploration in dynamic symbolic execution." In: *Proc. 2009 IEEE/IFIP Int. Conf. on Dependable Systems and Networks*. DSN'09. 2009, pp. 359–368 (cit. on p. 49).

[74] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling." In: *Proc. of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010. ISBN: 978-1-60558-577-2 (cit. on p. 3).

[75] Qi Zhang, Lu Cheng, and Raouf Boutaba. "Cloud computing: State of the Art and Research Challenges." In: *Journal of internet services and applications* 1 (2010) (cit. on p. 23).

[76] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. "Regular Property Guided Dynamic Symbolic Execution." In: *Proc. 37th Int. Conf. on Software Engineering*. ICSE'15. Florence, Italy, 2015, pp. 643–653. ISBN: 978-1-4799-1934-5 (cit. on p. 49).

A

APPENDIX EXAMPLE: CODE LISTINGS

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Knuth, "Computer Programming as an Art," 1974

A.1 THE `listings` PACKAGE TO INCLUDE SOURCE CODE

Source code is usually not part of the text of a thesis, but if it is an original contribution it makes sense to le the code speak by itself instead of describing it. The package `listings` provide the proper layout tools. Refer to its manual if you need to use it, an example is given in listing A.1.

Listing A.1: Code snippet with the recursive function to evaluate the pdf of the sum Z_N of N random variables equal to X.

```
std::vector<int> values_of_x(number_of_values_of_x,
  min_value_of_x);
for (unsigned int i = 1; i < number_of_values_of_x; i++) {
  values_of_x[i] = values_of_x[i - 1] + 1;
}
prob_x = 1.0 / number_of_values_of_x;
std::vector<std::vector<double> > p_z;
for (unsigned int idx = 0; idx < p_z.size(); idx++) {
  p_z[idx] = std::vector<double>(
    (max_value_of_x * (idx + 1) - min_value_of_x
      * (idx + 1)) + 1, INIT_VALUE);
}

double prob(int Z, int value_of_z) {
  if (value_of_z < min_value_of_x * Z ||
    value_of_z > max_value_of_x * Z) {
      return 0.0;
  }
  if (value_of_z < min_value_of_z ||
    value_of_z > max_value_of_z) {
      return 0.0;
  }
  int idx_value_of_z = -(min_value_of_z - value_of_z);
  int idx_N = Z - 1;
  if (p_z[idx_N][idx_value_of_z] == -2.0) {
    if (Z > 1) {
      double pp = 0.0;
      for (unsigned int i = 0; i < number_of_values_of_x; i++) {
        pp += prob(Z - 1, value_of_z - values_of_x[i], p);
      }
      p_z[idx_N][idx_value_of_z] = prob_x * pp;
    } else {
      if (Z == 1) {
        for (unsigned int j = 0; j < number_of_values_of_x; j++)
            {
          if (value_of_z == values_of_x[j]) {
            p_z[idx_N][idx_value_of_z] = prob_x;
            break;
          }
        }
      }
      if (p_z[idx_N][idx_value_of_z] == INIT_VALUE) {
        p_z[idx_N][idx_value_of_z] = 0.0;
      }
    }
  }
  return p_z[idx_N][idx_value_of_z];
}
```