A Live Debugging Approach for Big Data Processing Applications

Matteo Marra

Dissertation submitted in fulfillment of the requirement for the degree of Doctor of Sciences

May 3, 2022

Promotors:
Prof. Dr. Elisa Gonzalez Boix, Vrije Universiteit Brussel, Belgium
Dr. Guillermo Polito, Univ. Lille, CNRS, UMR 9189 CRIStAL, France

Jury:
Prof. Dr. Geraint Wiggings, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Bas Ketsman, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Jan De Beule, Vrije Universiteit Brussel, Belgium
Prof. Dr. Kim Miryung, University of California, Los Angeles, USA
Prof. Dr. Guido Salvaneschi, University of St.Gallen, Switzerland
Prof. Dr. Luc Fabresse, IMT Nord Europe, France

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789464443202
NUR 989

The work in this dissertation has been funded by a PhD fellowship of the Research Foundation Flanders (FWO) - Project number 1S63418N.

Abstract

The modern world heavily relies on data: in 2020, more than 64 ZBs of data were created, captured, copied, or consumed globally. As a result, novel software platforms have emerged to analyze large data sets from several domains in a parallel and scalable way. The two most prominent programming models for Big Data processing are Map/Reduce and Apache Spark. Both models envision the programming of complex problems through well-known functions and let the framework take care of the distribution aspects such as parallelization and fault tolerance to node failures.

Debugging Big Data applications is difficult due to their distributed and parallel nature, which increases the distance between the root cause of the bug and the observed failure. Furthermore, developers tend to use a large technology stack, which also complicates the debugging. A common debugging practice is to analyze log files, but they lack contextual information about which record(s) caused an error. Recently, Record & Replay debuggers have been explored, but replaying Big Data applications can be very costly since they are normally long-lasting. Checkpoint-based debugging has been explored to lower the replay time, but still requires the creation of a checkpoint and a replay step.

In this dissertation, we explore a live debugging approach tailored to Map/Reduce and Spark-like programs. We first propose out-of-place debugging, a novel debugging architecture to debug remote and distributed applications. In this model, when there is an error in an application running remotely (e.g., in a cluster), the state of the computation is transferred to the developer's machine, in which the application can be debugged. This avoids replaying the execution while offering a full interactive debugging environment.

We then explore the applicability of out-of-place debugging to parallel distributed Map/Reduce and Spark-like applications, through two novel techniques for optimizing their debugging: composite debugging events, i.e., the grouping and centralized debugging of multiple similar debugging events, and dynamic local checkpoints, i.e., dynamic capturing of the execution state. Thus, we enable centralized debugging of remote Big Data applications and extend it with domain-specific debugging operations. Finally, we complement our debugging approach with a relaxed computational model that allows developers to instruct the runtime to automatically ignore a defined number of exceptions that happen at runtime. This feature is especially relevant for those data analytics applications that can accept a loss in accuracy (e.g., because of dirty data).

We implement our debugging techniques in Pharo Smalltalk on top of Port and Spa, our frameworks implementing the Map/Reduce and the Spark-like model, respectively. Furthermore, we generalized all the call-stack operations needed to enable our debugging approach in Sarto, a call-stack instrumentation layer for stack tailoring. The proposed out-of-place debugging approach applied to debug Map/Reduce and Spark-like programs, together with Sarto, represent the main contributions of this dissertation.

Our validation is two-fold: we validate our debugging approach quantitatively and qualitatively. For the quantitative study, we conducted performance benchmarks that show that our model scales to an increasing amount of both data and parallel exceptions. For the qualitative study, we conducted a user study to assess the usability of our solution for solving different debugging tasks and compare it to a reproduction of a state-of-the-art debugger for Spark applications. The results show that participants reported a better debugging experience using our debugger and validated positively the advanced features offered by our debugger.

Samenvatting

De moderne wereld is sterk afhankelijk van data: in 2020 zal wereldwijd meer dan 64 ZBs aan data worden gecreëerd, vastgelegd, gekopieerd of verbruikt. Als gevolg daarvan zijn nieuwe softwareplatforms ontstaan om grote datasets uit verschillende domeinen op een parallelle en schaalbare manier te analyseren. De twee meest prominente programmeringsmodellen voor Big Data-verwerking zijn Map/Reduce en Apache Spark. Beide modellen richten zich op het programmeren van complexe problemen via vertrouwde functies, en laten het framework de distributieaspecten afhandelen, bijvoorbeeld de parallellisatie en de fouttolerantie voor node-fouten. Het debuggen van Big Data-toepassingen is moeilijk door hun gedistribueerde en parallelle aard, waardoor de afstand tussen de hoofdoorzaak van een bug en een waargenomen fout groter is. Bovendien hebben ontwikkelaars de neiging om een grote technologiestack te gebruiken, wat het debuggen ook bemoeilijkt. Een gebruikelijke debugging methode is het analyseren van logfiles, maar deze missen contextuele informatie over welke record(s) een fout veroorzaakte(n). Recentelijk zijn Record & Replay debuggers bestudeerd, maar het opnieuw afspelen van Big Data applicaties kan erg kostbaar zijn omdat ze normaal gesproken lang duren. Checkpoint-gebaseerde debugging technieken kunnen in principe de replay tijd verminderen, maar vereisen steeds de creatie van een checkpoint en een replay stap. In deze dissertatie onderzoeken we geavanceerde online debugging oplossingen afgestemd op Map/Reduce en Spark-achtige programma's. Eerst stellen we out-of-place debugging voor, een nieuwe debugging architectuur om remote en gedistribueerde applicaties te debuggen. In dit model, wanneer er een fout optreedt in een applicatie die op afstand draait (bv. in een cluster), wordt de staat van de berekening overgebracht naar de machine van de ontwikkelaar, m.a.w. naar de locatie waar de applicatie gedebugged kan worden. Dit vermijdt het op-

nieuw herhalen van de uitvoering, terwijl het een volledig interactieve debug-omgeving biedt. We verkennen vervolgens de toepasbaarheid van out-of-place debugging op parallelle en gedistribueerde Map/Reduce en Spark-achtige applicaties, door middel van twee nieuwe technieken voor het optimaliseren van het debug process: composite debugging events, dat wil zeggen het groeperen en gecentraliseerd debuggen van meerdere soortgelijke debugging events, en dynamische lokale checkpoints, dat wil zeggen het dynamisch vastleggen van de uitvoeringsstatus. Zo kunnen we Big Data-applicaties gecentraliseerd vanop afstand debuggen, en breiden we het uit met domeinspecifieke debugging-operaties. Ten slotte complementeren we onze debugging-aanpak met een versoepeld computationeel model waarmee ontwikkelaars de runtime kunnen opdragen om automatisch een gedefinieerde hoeveelheid uitzonderingen te negeren die tijdens runtime optreden. Deze functie is vooral relevant voor data-analyse toepassingen die een verlies in nauwkeurigheid kunnen accepteren (bijv. als gevolg van vervuilde gegevens). We implementeren onze debugging technieken in Pharo Smalltalk, bovenop Port en Spa, onze frameworks die respectievelijk het Map/Reduce en het Spark-achtige model implementeren. Verder hebben we alle call-stack operaties die nodig zijn om onze debugging aanpak mogelijk te maken veralgemeend in Sarto, een call-stack instrumentatie laag voor stack tailoring. De voorgestelde out-of-place debugging aanpak toegepast op het debuggen van Map/Reduce en Spark-achtige programma's, samen met Sarto, vertegenwoordigen de belangrijkste bijdragen van dit proefschrift. Onze validatie is tweevoudig: we valideren onze debugging aanpak kwantitatief en kwalitatief. Voor de kwantitatieve studie hebben we performantie benchmarks uitgevoerd die aantonen dat ons model schaalt naar een toenemende hoeveelheid van zowel data als parallelle uitzonderingen. Voor de kwalitatieve studie hebben we een gebruikersonderzoek uitgevoerd om de bruikbaarheid van onze aanpak voor het oplossen van verschillende debugging taken te beoordelen en deze te vergelijken met een reproductie van een state-of-the-art debugger voor Spark applicaties. De resultaten tonen aan dat de deelnemers een betere debugging-ervaring rapporteerden door gebruik te maken van onze debugger, en dat ze de geavanceerde functies die onze debugger biedt positief waardeerden.

Acknowledgements

I would like to start these acknowledgments by thanking my two promotors Elisa Gonzalez Boix and Guille Polito for the constant support, help, and drive they have given me throughout these last 6 years, starting from the first meeting we had for my master thesis until defending this PhD. Thank you for all your dedication, work ethic, patience, and for all the off-work moments we shared in front of a beer, or a cola zero, or even dark and stormies!

Thanks to the members of my jury: Prof. Dr. Geraint Wiggings, Prof. Dr. Bas Ketsman, Prof. Dr. Jan De Beule, Prof. Dr. Kim Miryung, Prof. Dr. Guido Salvaneschi, and Prof. Dr. Luc Fabresse. Thank you for reading this dissertation and giving me feedback to improve the text.

I would also like to thank the different colleagues that shared this experience with me, starting from my friend, debugging duck, and officemate Jim Bauwens who I have to thank for many things, including the cover of this book, the *delta stacks* name, and all the ducks and Catalan flags in our office. Thanks to Carmen for reviewing my text multiple times, including a chapter of this thesis. Thanks to the other members of DisCo (Scull, Isaac, Kevin, Carlos, and Clement) for all the dicussions and the fun moments! Thanks to my former colleague and friend Dario for all the coffees, pizzas, F1 races, and discussions about Italian politics that helped me feel more at home. I would also like to thank all the members of the Software Languages Lab (SOFT) that I had the pleasure to work with during the last 4 years. Thanks for the continuous feedback and thank you also for the Friday "gatherings". A big thank you also goes to the members of the RMoD lab that has hosted me many times during my PhD, starting from Stéphane Ducasse until the last of the Argentinians, with a special mention to Pablo, Santi, and Steven.

You will now excuse me if I will switch to Italian for the next few lines, which will be the only ones my family will be able to understand in this whole dissertation. Vorrei ringraziare la mia famiglia per tutto il supporto e l'amore che mi hanno dato non solo in questi anni a Bruxelles, ma in tutta la mia vita, accademica e non. Grazie Mamma, Grazie Papà, perchè senza tutto il vostro appoggio e senza la vostra apertura mentale non sarei mai arrivato qui e soprattutto non sarei quello che sono oggi. Un grande grazie anche al resto della grande famiglia Marra-Massaccesi, dal più piccolo a chi non c'è più.

Un grand merci to the Diarra-Donvil family for welcoming me in their home and supporting me throughout these years, becoming de facto my second family.

Thanks to my friends in Brussels, Italy, and scattered around the world for helping me and supporting me in this experience. A particular thank you goes to my dear daughter Paola, my great flatmate Kelly, the unreplaceable Vignu, and the frequent visitor Steve.

Finally, I would like to thank my love, friend, girlfriend, and life companion Marie-Louise for always being by my side during this PhD. Thanks for still being there, even after spending a couple of lockdowns with me while I was submitting papers, preparing a user study, and writing this thesis!

Ah, I almost forgot! Thank *you* for reading at least two pages of this dissertation! If you dare reading the rest, let me first give you a little piece of advice borrowed from the great Poet

"Lasciate ogni speranza, o voi che entrate!"

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Big Data processing is surely one of the biggest trends of the last couple of decades. In 2001, D. Laney introduced three concepts that lay the basis for today's understanding of Big Data: data Volume, Velocity, and Variety, commonly referred to as the *3 Vs* [Gar]. Particularly, the three Vs were first described in the context of e-commerce as the increase of depth and breadth of data generated by a transaction (i.e., Volume), the pace of data generated to support interactions (i.e., Velocity), and the variety of incompatible data formats and data structures (i.e., Variety).

Not only do the 3 Vs apply to e-commerce data, but they also do apply to any sort of data that needs to be stored, interacted with, and analyzed. In 2004, J. Dean and S. Ghemawat described the solution Google had started using to process large amounts of data: Map/Reduce [DG04]. Map/Reduce introduced a model based on two popular functional programming constructs: *map* and *reduce*. It lets developers focus on the application logic by handling all aspects of distribution, tolerance to node failures (i.e., nodes becoming unavailable), task granularity, and more. Through the open-source version of Map/Reduce (i.e., Hadoop Map/Reduce [Apaf]) Big Data processing then became available for analysts in any field. This, in turn, led to the development of different frameworks targeting different domains, e.g., Apache Giraph [Apaa], DryadLINQ [Micb], Apache Pig [Apac].

In 2010, another now popular Big Data framework came to light: Apache Spark [Apad]. Spark builds on the concept of a distributed data structure (i.e., an RDD [ZCD+12]), on which a large functional API is

available to transform and analyze data. Through its focus on fault tolerance to node failures, optimizations, and more fine-grained control over handling intermediate data, Spark quickly became popular in both research and industry.

Despite the popularity of these frameworks, developing Big Data applications remains challenging due to their distinguishing properties. Big Data applications are notoriously (i) long-running, due to the high volume of data they have to analyze; (ii) subject to a complex configuration, due to the stack of heterogeneous technologies they are based on; and (iii) generally executed remotely on clusters, which increases the time of deploying and initializing of an application, as well as the retrieval of information about an execution. Those properties also lead to different kinds of failures. First, errors due to the misconfiguration of at least one of the different libraries and frameworks used in the technological stack [RK13]. Second, developers also often have to deal with dirty data sets [FDCD12, MLW+19] which increase the number of errors in their programs: a single record could in fact invalidate a long computation. Finally, a 2019 study [BK19] shows that 40% of reported failures are due to errors introduced by the developers.

This dissertation focuses on the debugging of Big Data applications, an integral part of software development and still a major concern for Big Data developers. A 2015 field study [ZLZ+15] shows that debugging is the third most recurring topic after questions about the models themselves in StackOverflow, i.e., a popular platform for developers to request help on any programming language and framework. More in general, a 2013 study [BJC+13] shows that developers spend at least 50% of their time debugging, costing the global software industry an estimated amount of 312 billion USD, including developers' salaries and overheads.

Debugging Big Data applications remains difficult due to their distinguishing characteristics. In the literature, some work has focused on automated debugging in the form of static and dynamic analysis to find inputs that cause errors in a program [LRS+13, GMMK19, ZWG+20]. Other work has focused on debuggers, i.e., tools to dynamically observe and control the execution of a program. Several prior works have focused on offline debugging for Big Data, i.e., debugging an execution after it failed by reconstructing it. They are mainly based on replaying the execution [DZSS13, SSK+15], which can be impractical since Big Data

applications are long-running. More recently, BigDebug [GIY+16] and Daphne [JYB11] have explored the use of online debugging concepts such as breakpoints. They do this by adding breakpoints to an offline debugging backend [GIY+16, JYB11]. In this dissertation, we explore an online debugging approach especially designed for the properties of Big Data applications.

1.1 Problem Statement

As mentioned above, due to their unique properties debugging Big Data applications is a challenging task that developers have to deal with. Although Big Data frameworks are fault-tolerant to node failure, they will abort the execution upon application failure after retrying to execute it on another node. This can lead to hours of computation being invalidated and replayed for debugging because of a few records that cause an application to fail. For example, in a Map/Reduce application a bug while reducing will invalidate the results of the map. Current debugging solutions for Big Data applications rely on the replaying of at least part of the execution of an application, which may take long due to the long-running property of Big Data applications. Furthermore, if errors do not affect the accuracy of an application, they could even be systematically ignored, but current solutions for debugging Big Data applications do not allow it.

For the purpose of this dissertation, we identify below different shortcomings of current debugging approaches for Big Data applications:

High Replay Times. Replaying even parts of Big Data applications is a time-consuming process due to the size of the data that the applications process.

Limited Online Debugging Capabilities. Current solutions offer limited online debugging capabilities, providing developers with few contextual information on an execution necessary for understanding the behaviour of the application to identify errors.

Limited Live Code Updating Support. Updating Big Data applications, especially when deployed remotely, is a cumbersome process that requires several steps such as packaging, uploading, restarting, etc. While some solutions offer limited code updating capabilities, live code updating has not been explored for general code changes.

No Acceptability of Failures. The debugging models for Big Data applications do not react automatically to application failures to ignore them. This could be used to avoid tedious debugging of minor problems that could be ignored on applications which can deal with a level of inaccuracy.

The research explored in this dissertation is guided by the following research statement:

We conjecture that an online and live debugging approach that provides replay-free debugging of a remote Big Data application and live code updates is a suitable solution to deal with the properties of Big Data applications.

1.2 A Live Debugging Approach

In this dissertation, we propose a novel debugging approach for Big Data application inspired by two ideas:

Live Coding and Debugging. Smalltalk [Gol84] pioneered the idea of an interactive environment to write, run, debug, and update a program while running it, all in the same environment. This entails a high level of interaction between the developer, the tools of the environment, and the program. For example, the debugging of a breakpointed execution or an unexpected error happens in the same way, since the debugging support is always-on. The developer can evaluate and open a debugger on any expression in their program. The concepts of live programming are not only embraced by classical Smalltalk systems: as an example, the concepts of live editing and execution of arbitrary code (and expressions) are nowadays very popular in the field *interactive programming* in the form of programming notebooks (e.g., Jupyter).

Acceptability-oriented Computing. Acceptability-oriented computing was first defined by Rinard [Rin03] as a failure-oblivious system that describes the properties that state and behaviour must preserve for the program's execution to be acceptable, and that then monitors and enforces these acceptability properties and eventual violation.

The concept of acceptability-oriented computing was later used by Carbin et al. [CKMR12] to further define *relaxed programs*, i.e., programs that "have been extended with additional nondeterminism to relax their semantics and enable greater flexibility to the execution". While this approach has been explored in different fields [ZM19, RPM18], to the best of our knowledge it has not been applied to parallel Big Data applications.

In this thesis, we argue that:

- Live debugging reduces the duration of the debugging cycle of Big Data developers by providing replay-free debugging of an execution with all the error's contextual information, live code updates, breakpoints, and an overall interactive environment to develop applications. However, a live debugger normally controls the execution of applications deployed in the same process and not distributed. Thus, we explore its applicability in the context of Big Data applications.
- Acceptability-oriented computing can avoid invalidating long computations due to a minor error. Since many data science applications can afford to lose a degree of accuracy [CCRR13], we explore its integration in an execution and debugging model for Big Data.

We explore our live debugging approach for Big Data applications to two programming models, i.e., Map/Reduce and Spark. We also devise domain-specific debugging modes and operations which enhance the debugging experience for the two models. Finally, we explore acceptability-oriented computing through different execution modes and extensions to the programming models' API to enable the systematical ignoring of errors.

1.3 Contributions

This dissertation presents four main contributions.

Out-of-Place debugging. A debugging model for the online debugging of remote executions inspired by Smalltalk's live programming model. It enables the debugging of remote applications by transferring the remote execution state to a process running at the developer's machine where debugging operations are performed. Thus, debugging

happens locally with an online debugger, and the developer can then apply code updates back to the remote application once the bug is fixed through committing their changes.

Live debugging of Map/Reduce applications. An approach for debugging Map/Reduce applications based on out-of-place debugging combined with the introduction of *composite debugging events*, domain-specific debugging modes. We prototype our approach in IDRA_{MR}, a debugger for Map/Reduce applications in Pharo Smalltalk. We validate this approach by using IDRA_{MR} in a real-world scenario of a blockchain analysis application.

Live debugging of Spark-like applications. A two-fold approach for debugging Spark-like applications that enables both debugging and ignoring of errors. Our debugging approach extends out-of-place debugging to centralize debugging sessions with *dynamic local checkpoints*. It improves the debugging experience with domain-specific debugging operations for Spark-like applications (i.e., coarse-grained stepping operations). We complement our debugging approach with a relaxed computational model for the systematic ignoring of errors. We validate our overall approach through performance benchmarks and through a user study for assessing the usability of our debugger in comparison to the state of the art.

A stack tailoring instrumentation layer. An instrumentation layer to tailor call-stacks for debugging framework executions through a set of six operations. These operations are used in our prototype debuggers for Big Data applications but fall beyond the scope of Big Data. We validate the different operations by showing their applicability to debugging three different frameworks and their impact on the execution.

1.3.1 Technical Contributions

To support our contributions, we developed different artifacts that represent a technical contribution.

Port and Spa. Two Big Data frameworks for Pharo Smalltalk supporting the Map/Reduce and the Spark-like model, respectively. These

two frameworks enable the programming and execution of parallel Big Data applications in Pharo Smalltalk.

IDRA$_{\text{MR}}$ and SpaDebug. The implementation in Pharo Smalltalk of our live debugging approach for the Map/Reduce and the Spark-like model, respectively.

Sarto. A library for tailoring the call-stack of framework applications to enable debugging in Pharo Smalltalk.

1.3.2 Supporting Publications

In what follows, we list the publications that support this dissertation.

- **Out-of-place debugging: A debugging architecture to reduce debugging interference.**
  The Art, Science, and Engineering of Programming, Volume 3, Issue 2, Article 3, 2018. [MPGB18]
  Matteo Marra, Guillermo Polito, Elisa Gonzalez Boix.

  This publication introduces out-of-place debugging as an online debugging architecture for the debugging of remote applications, that lies at the basis of our debugging solution. It presents IDRA, the first implementation of out-of-place debugging for Pharo Smalltalk, and validates the performance of the debugger in comparison to remote debugging. The concepts of out-of-place debugging are described in Chapter 5.

- **Framework-Aware Debugging with Stack Tailoring.**
  Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, pp. 71-84, 2020. [MPGB20b]
  Matteo Marra, Guillermo Polito, Elisa Gonzalez Boix.

  This publication introduces Sarto, our call-stack instrumentation layer for tailoring the stack of framework execution errors for improving debugging described in Chapter 4. It presents different operations to tailor and manipulate call-stacks and uses them in the context of four different execution frameworks also validating their performance.

- **A debugging approach for live Big Data applications.**
  Science of Computer Programming, Volume 194, Article 102460, 2020. [MPGB20a]
  Matteo Marra, Guillermo Polito, Elisa Gonzalez Boix.

  This publication introduces our live debugging approach for Map/Reduce applications by using out-of-place debugging to debug Port applications in a centralized way. It validates the solution by showing how this debugger can be used to debug both application and configuration errors. The mains concepts of this publication related to the debugging of parallel exceptions, domain-specific debugging modes, and live code updating are described and expanded in Chapter 6.

- **Practical Online Debugging of Spark-Like Applications.**
  To appear in Proceedings of the 21st IEEE International Conference on Software Quality, Reliability, and Security, 2021. [MPGB21]
  Matteo Marra, Guillermo Polito, Elisa Gonzalez Boix.

  This publication presents a live debugging approach for Spark-like applications, complemented with a relaxed computational model to ignore errors. This paper presents the bulk of our concepts for debugging Spark-like applications and introduces the idea of ignoring exceptions. Both of these aspects are further explored in Chapter 7. The paper validates the solution through several performance benchmarks to show the scalability of both the debugger and the relaxed computational model. It also presents the results of a user study to assess the usability of our solution in comparison to state-of-the-art. The results of this validation are presented in this dissertation in Chapter 8, together with more benchmarks for assessing the impact of different optimizations.

1.4 Dissertation Outline

This dissertation has the following outline:

Chapter 2: Context and Motivation. This chapter provides the context to our work, starting by describing Map/Reduce and Spark, two popular Big Data frameworks that we aim to debug. Then, it de-

tails debugging and several general-purpose debugging approaches, before delving into current debugging solutions for Big Data frameworks. We conclude this chapter by extracting from the state of the art different criteria that debuggers for Big Data applications should uphold.

Chapter 3: Scalable Big Data Frameworks for Pharo Smalltalk. This chapter describes Port and Spa, the frameworks that we built to support the development and execution of Map/Reduce and Spark-like applications in Pharo Smalltalk. We start the chapter by detailing our experimental platform: Pharo Smalltalk. Then, we describe the infrastructural layer used by both Port and Spa to enable the execution of Big Data programs. We then introduce a running example that we use to describe the internals of Port and Spa. We conclude the chapter with a description of how Port and Spa are deployed on clusters, to clarify the setup that we use for validating our approach later in the dissertation.

Chapter 4: A Call-Stack Instrumentation Layer for the Debugging of Framework Code. This chapter describes our stack tailoring instrumentation layer which includes six stack tailoring operations and shows how they are used in practice on four debugging cases through Sarto, the Pharo Smalltalk library that implements the six operations. Finally, we validate Sarto by assessing the overhead of the different operations on the execution.

Chapter 5: Out-of-Place Debugging. This chapter introduces out-of-place debugging, a live and online debugging model, by first describing its debugging architecture, and then how it is enabled in different runtime environments for debugging distributed programs.

Chapter 6: Debugging Support for Map/Reduce. This chapter introduces out-of-place debugging for Map/Reduce applications starting by describing the adapted out-of-place architecture for Big Data execution models. Then, we describe how we extract contextual information from debugging events, centralize the debugging session, and enable live code updating. We present IDRA_{MR}, a live out-of-place debugger for Map/Reduce applications in Port, and we conclude the chapter by showing how our solution can be used to debug a realistic blockchain indexing application.

Chapter 7: Debugging Support for Spark-like Applications. This chapter introduces our out-of-place debugging approach for Spark-like application by describing how to extract contextual information, introducing domain-specific stepping operations, and describing a relaxed computational model for Spark-like application that enables the systematical ignoring of failures. We then present SpaDebug, a live out-of-place debugger for Spark-like applications in Spa.

Chapter 8: Validation. This chapter validates our solution presented in Chapter 7 by showing the results of several performance benchmarks to assess the overhead and scalability of our debugger and of the relaxed computational model.We then detail the design, methodology, and results of a user study with 17 participants that shows the usability of our solution in comparison to a state of the art debugger.

Chapter 9: Conclusion. This chapter concludes the dissertation by giving an overview of our approach, revisiting the contributions, discussing the limitations of our work, and describing possible avenues for future work.

Chapter 2

Context and Motivation

This chapter provides the research context of our work and motivates the need for debugging support for Big Data applications. Particularly, we start by detailing the Master/Worker model that lies at the basis of Map/Reduce and Apache Spark, which we also describe in this chapter. This should give to the reader the necessary background on the two programming models proposed by the frameworks.

Since this dissertation focuses on debugging support, we then delve into the description of basic concepts of debugging and several offline and online debugging techniques. This is followed by a description of common bugs in Big Data, and a review of the state of the art of debugging approaches for Big Data applications and their characteristics. We conclude this chapter by defining the criteria of debuggers for Big Data applications.

2.1 The Master/Worker Model

The *Master/Worker* model is an execution model enabling the execution of parallel tasks in a coordinated way. In a distributed context (e.g., a cluster of machines), one node of the distributed system takes the role of *master* and coordinates the other nodes (i.e., the *workers*), assigning jobs (or tasks) to them and retrieving their results. The *Master/Worker* architecture is often an easy approach to divide the work between multiple workers, and, because of its simplicity and scalability, it is used as an underlying execution model in many concurrent and distributed architec-

tures, including Map/Reduce, Spark, and several streaming frameworks such as Apache Flink.

In this thesis we focus on Big Data processing, hence in the rest of this section, we look at the Master/Worker model in the context of a parallel execution model on a cluster of machines.

In the Master/Worker model, the goal of the master is to optimally divide the work between all its workers, thus efficiently employing the computational power of the system. To accomplish this, the master includes a *scheduler*, which extracts a task from the task list and selects a worker on which the task will be executed. A scheduler can be implemented both as a component or as a scheduling function, and it assigns tasks to workers in different ways, depending on the implementation. Tasks can be assigned by time slots, by their complexity, or, in more advanced distributed systems, they can also be assigned by distance and network latency. For example, in the case of Big Data frameworks such as Apache Spark, how tasks are assigned mainly depends on the locality of the data. After a worker processes a task, it returns a value (or a set of values) that has to be handled by the master. The master can then return the results to the user or feed the result to other workers for other tasks.

Despite being simple, the Master/Worker model also presents a degree of tolerance to node failures: due to its underlying centralization, a failure in a *worker* is easily handled by the *master* node, which can then decide to reschedule the execution in another worker or to graciously stop the execution in all other workers. In this architecture, however, the master is the single point of failure of the system. For instance, if the *master* fails or is disconnected from the network, the *workers* will not receive more tasks from a *master* node and will not be able to return the processed results. As consequence, the whole system could stop working. In practice, the benefits of the model often outweigh this limitation, since terminal failures in the master node are reportedly less frequent than the ones in worker nodes [DG04].

2.2 Map/Reduce

The Map/Reduce model [DG04, DG08] is a programming and execution model, created for a necessity at Google to analyze large input data in parallel. While they were doing that for years, they realized that the actual

code of the analysis was pretty simple and obscured by complex code to handle the distribution of the computation and its fault tolerance. This led to a new programming model to easily write computations on large data, supported by an execution model based on the Master/Worker model to perform the execution in parallel (execution model). In this section, we first discuss the Map/Reduce model by first looking at the programming and execution model, and then look at how it handles distributed data, and finally how it deals with fault tolerance.

2.2.1 Programming and Execution Model

As the name suggests, a Map/Reduce application is expressed in the implementation of two functions: a map and a reduce. The runtime then takes care of retrieving the data, parallelizing the computation, scheduling it into the workers, and handling failures.

Let us first describe the two functions that developers need to implement:

Map The map function takes as input a key/value pair and returns an intermediate key/value pair.

Reduce The reduce function takes as input an intermediate key and the set of intermediate values associated with that key. Typically, those values are merged (i.e., reduced) together into one or no values, representing the result of the computation for that particular key.

To give an example, consider Listing 2.1 which shows the implementation of a classical distributed wordcount application, similar to the one presented by Dean and Ghemawat [DG04]. As expected, the listing shows the implementation of the two functions: map and reduce. In this case, `map` takes the document name as the key and the contents of the document as the value. In the `map` function, a `for each` looks through all the words of a document and emits a pair containing the current word as key and one as value, representing an intermediate result. The `reduce` function takes as parameters one of the words and an iterator to the list of values (i.e., a list of 1s) associated with that key, i.e., the results from the application of the `map`. It then initializes a result integer and loops through the values parameter to add the value to the result. Finally, it emits the final result for that particular word.

```
1  map(String documentName, String documentContents){
2     for each word in documentContents{
3        emit (word,1)
4     }
5  }
6
7  reduce(String word, Iterator values){
8     int result = 0
9     for each number in values{
10       result += number
11    }
12    emit result
13 }
```

Listing 2.1: An example wordcount in Map/Reduce.

The implementation of this application includes no boilerplate code to parallelize the execution, manage intermediate data, and handle fault tolerance: the framework handles all of these concerns. Recall that the execution model of Map/Reduce is based on the Master/Worker model. When executing the program, the master splits the data into several input files, replicated in the cluster, so that the workers can start executing map tasks on it. In the following section, we describe in more detail how data is treated during the execution.

2.2.2 Partitioning and Data Locality

Map/Reduce assumes the availability of a distributed file system to load the data from. For instance, Google's Map/Reduce uses GFS (Google File System), while Hadoop's version uses HDFS (Hadoop Distributed File System). Map tasks are scheduled so that a certain worker loads the portion of the input that is already stored in it. When this is not possible, e.g., because the worker holding the file is busy, tasks are scheduled so that the data is loaded from a nearby worker, reducing network usage.

Upon a map, before the intermediate results are loaded by workers to apply a reduce, the execution framework performs two operations: first, the intermediate results are partitioned, and then they are sorted. In practice, intermediate results are buffered from the worker's memory to disk and partitioned into different regions (or partitions) according to a

partitioning function. The default partitioning function uses the hash of the key to separate the data in a balanced number of partitions: if R is the number of reduce tasks to apply, the data is partitioned using *hash(key) mod R*. The developer can also provide, as part of their Map/Reduce application, a customized partitioning function to their application, for instance by hashing only part of the key to make sure data is partitioned as it is required by their reduce function.

After intermediate data is partitioned and stored on disk, the location of those partitions is returned to the master. This enables the master to schedule reduce tasks on workers by indicating a certain location (pointing to a region) as input data. Most schedulers, when possible, will schedule reduce tasks in workers that are close to the location of the data, thus reducing reduce network delays that would be introduced by reading the data stored in another worker. The assigned worker then loads the input data and sorts it before applying the reduce. In this way, multiple values referring to the same key are grouped (and later analyzed) together.

Part of the success of the Map/Reduce model is surely attributed to the way data is handled. Moreover, the partitioning approach is important to enabling fault tolerance, which we discuss in the next section.

2.2.3 Fault Tolerance

Map/Reduce was originally designed to run on a cluster of not necessarily reliable machines. Therefore, fault tolerance to node failure is embedded in the execution model and made transparent to users.

How the system reacts to a failure depends on where the failure is located. The master regularly pings the workers to verify that they are still alive. If a worker does not respond for a certain amount of time, the worker is marked as failed, and all the running and completed map tasks are marked as idle, i.e., ready to be scheduled in other workers. This is because, if a worker became inaccessible, the intermediate results of running or completed tasks are no longer retrievable from its disk. Results of completed reduce tasks, instead, are stored on the distributed file system and hence those tasks do not need to be re-executed.

The fault tolerance mechanisms of Map/Reduce not only enable the handling of node failures but also solve other problems that may lead a program to never terminate. An example of this is the handling of *stragglers*, i.e., workers that take a long time to terminate a task or that

never terminate because of an internal problem. A commonly adopted solution to this is to use *backup tasks*, i.e., tasks that are scheduled by the master when a task is taking particularly long to complete when compared to similar tasks. If the original task completes, then the backup task is removed from the list of tasks that have to be scheduled. Otherwise, it is left to be executed. If the backup task terminates before the original task, its result is used and the original task discarded. Otherwise, the opposite happens.

Finally, in case of failure of the master, the computation is terminated and needs to be restarted. However, according to the original authors [DG04] it is possible to checkpoint the state of the master (that includes information on the workers, tasks, and locations of intermediate files) to restart the master from there in case of a failure.

2.2.4 Conclusion

In this section, we have described the main concepts of the Map/Reduce model.

Despite its success, the Map/Reduce model presents limitations in how programs can be expressed: the simplicity of using just two functions forces to implement the behaviour of other operations within the two main operators. Grouping and joining, for example, needs to then be implemented manually. Furthermore, storing intermediate data only onto disk represents another limitation, since doing so can create high delays in the execution when compared to persisting intermediate data in memory. While it is needed in many cases, developers have little control to avoid this from happening.

2.3 Spark

Apache Spark [Apad] is a more recent Big Data framework that tries to solve some of the weaknesses of Map/Reduce. It is currently supported as a library in Scala, Java, Python, and R. In Spark, programs consist of the manipulation of data through functional calls over a distributed data structure called RDD (Resilient Distributed Data Structure) [ZCD+12]. The framework minimizes the materialization of RDDs in memory and offers control to the developers to trigger the persistence of data and to

modify its partitioning. As Map/Reduce, it is deployed on a Master/-Worker model to be executed in a distributed system.

Through RDDs, Spark often achieves better performance than Map/Reduce by avoiding persisting data on disk unless necessary. As explained in section 2.2, this happens by default in Map/Reduce before the reduce.

In what follows, we describe Spark's programming and execution model, data handling, and fault tolerance properties.

2.3.1 Programming and Execution Model

The main concept in Spark's programming and execution model is the concept of RDDs. An RDD is a fault-tolerant distributed data structure, that offers control over the partitioning of the data it contains and persistence in memory (or disk). RDDs are immutable data structures that store data into different *partitions* across several workers.

Developers manipulate RDDs through a large functional API of operations that either return a new RDD or a final result (i.e., a conventional data structure). Operations that transform an RDD into another RDD are called *transformation*. Those that return a conventional data structure or export the data contained by an RDD to disk are called *actions*. Thus, RDDs are created either by loading data from storage/memory or from other RDD(s).

Examples of common transformations are `map`, `filter`, and `reduce`. Examples of actions are `count`, `aggregate`, and other operations that collect data from the RDD such as `take` and `collect`.

```
1 main(..){
2   contents = Spark.textFile(filesFolder)
3   words = contents.flatMap(line => line.split(' ')).map(word => (word,1)
      )
4   resultRDD = words.reduceByKey(value1,value2 => value1 + value2)
5   result = resultRDD.collect()
6 }
```

Listing 2.2: An example wordcount in Spark.

To showcase the use of RDDs, Listing 2.2 shows an implementation of a distributed wordcount, similar to the one presented in Section 2.2. First, the file is loaded. Then, words are extracted by splitting every line of the file to identify the words, and then mapping every word to a pair

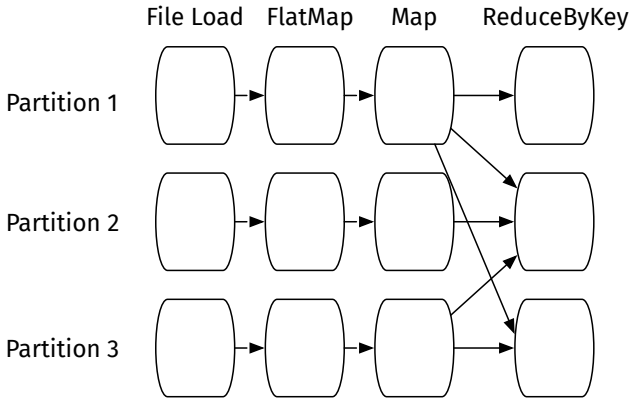


Figure 2.1: Overview of the partitions of an RDD across narrow and wide transformations in the wordcount example.

with the word itself as key and 1 as value. In the third line, `reduceByKey` is called on the words RDD to sum all the values associated with each key. Finally, `collect` is called on the resulting RDD to fetch the result.

Transformations are lazy, so they are not executed until one action is applied. This makes it possible to pipeline transformations to optimize their execution and reduce the materialization in memory of intermediate data.

As RDDs are generated from other RDDs through transformations, they present a dependency to the RDD(s) they are generated from, thus creating a *lineage*. Recall that an RDD stores data in partitions, distributed across different workers. Figure 2.1 displays an overview of the partitions of the RDD as it is transformed through the code of the wordcount presented in Listing 2.2. Each rectangle represents a partition, each column represents an RDD after the operation listed on top. The arrows indicate a dependency between parent and child partitions. There can be two kinds of dependencies between two RDDs:

- A dependency is *narrow* when each partition of the parent RDD is used by at most one partition of the child RDD (e.g., when flatMap and the map are applied in the figure).
- A dependency is *wide* when more than one partition of the child RDD depends on one parent RDD partition (e.g., after the reduceByKey is applied).

When an action needs to be executed in the user program on a particular RDD, the master analyzes its lineage (i.e., the set of transformations to be applied) to generate several *stages* (i.e., a logical execution unit), each presenting a DAG of narrow transformations. A stage contains as many narrow transformations as possible and ends with a wide transformation or an action. The set of stages represents the execution. Thus, a stage represents the execution of an action or wide transformation, and of all of the narrow transformations that happen before it. If the stage is executing an action, data is returned to the driver. If instead it is executing a wide transformation, a new RDD is returned to the driver.

2.3.2 Handling Intermediate Data

In this section, we discuss how Spark enables the persisting of intermediate results and how it handles data partitions with shuffling and partitioners.

2.3.2.1 Persisting

To avoid re-executing several times the same computations, RDDs can be persisted (i.e., materialized) in memory or disk through the *persist* transformation. When persisted in memory, the data belonging to an RDD is materialized in the partitions. Each worker may hold one or more partitions of a particular RDD. When persisted to disk, each partition is stored in a file on the local worker's storage. Persisting is in fact a sort of a checkpointing operation that stores the data of an RDD in memory or disk at a specific point. Persisting is normally used by developers to avoid replaying the same long-lasting computation multiple times, but it can also be used by tools. In a debugger, for example, the persisted data can be used to replay an execution partially, without reconstructing the persisted RDD. Persisted data also simplifies the process of *data provenance*, i.e., the tracking of record dependencies, since the data is available in memory to be analyzed[1].

[1] *Data provenance* is a technique used in databases and Big Data frameworks to track the provenance of records across different modifications.

2.3.2.2 Shuffling and Partitioning

Recall that wide transformations are executed at the end of a stage to produce a new RDD and that they involve an operation in which at least one partition of the resulting RDD is dependent on data of different partitions of the parent RDD. We call *shuffling* when data is moved for applying a wide transformation (e.g., `reduceByKey`)

As it potentially involves a lot of data being transferred over the network shuffling is considered a costly operation. Shuffling, however, is necessary to perform certain operations that are very common in data analysis such as grouping and reducing.

To improve how data is partitioned and thus to reduce the amount of shuffling, Spark offers a *repartition* functionality, available for RDDs that contain key-value pairs. Data is partitioned according to one of three different partitioners: (i) the hash partitioner, used by default, that repartitions the data by the hash of the record, (ii) the range partitioner, that divides data into equal amounts among the different partitions, or (iii) a custom partitioner, definable in a partitioning function, similarly to Map/Reduce.

Repartitioning is useful especially when the data of an RDD needs to be shuffled multiple times in different stages. By repartitioning, shuffling happens only at the moment of triggering the repartition. When repartitioning, the number of new partitions can be specified. This helps to adapt the partitions to the number of workers. Repartitioning is also useful to avoid hampering the parallelization of tasks to balance the number of records in each partition. This avoids clustering and thus improves the performance of later operations.

2.3.3 Fault Tolerance

Fault tolerance in Spark is built into RDDs. RDDs handle node failures by recovering the values of a specific partition in case a worker goes missing. This happens by storing in an RDD, partition by partition, all dependencies to the parent RDD's partitions in a lineage.

In practice, if a worker falls and becomes unavailable, i.e., if the data in its memory cannot be accessed anymore, the system tries to continue the execution by recovering the data of the unavailable worker. Taken a now unavailable partition, the master analyzes its lineage and instructs

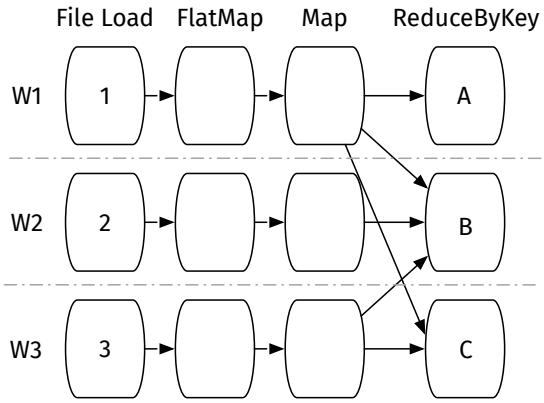


Figure 2.2: Overview of the partitions of an RDD across the wordcount example, highlighting the partitions after `reduceByKey`.

another worker to apply the actions and transformations contained in it, thus restoring the contents of the lost partition. The amount of re-executed operations depends on the type of transformations. To illustrate this, we show in Figure 2.2 a variation of Figure 2.1 in which we show which worker holds which partition (i.e., W1, W2, and W3) and we mark with a letter the partition after the `reduceByKey` operation. Arrows show dependency among the partitions. Particularly, partition A depends on partition 1, partition B depends on all other partitions, and partition C depends on partitions 1 and 3. If a worker becomes unavailable, all the data it holds also becomes unavailable and needs to be reconstructed. In case the linage includes only narrow transformations, once the parent partition is detected the set of transformations is applied to it to reconstruct the missing partition. In our example, if partition A needs to be recalculated, it suffices to re-execute the three transformations, starting with the file read, on the data of partition 1 to then execute the `reduceByKey`. When a wide transformation is in the lineage of the partition to reconstruct, it means that one partition of the parent RDD can be the parent of multiple partitions of the child RDD. In our example, partition B depends on all the three partitions of the RDD after the map. If worker 2 goes missing, partition B needs to be recalculated. Since partition B is dependent on all the partitions after the `map`, the framework needs to re-execute all the transformations on all partitions before applying the `reduceByKey` to reconstruct partition B. This is mitigated in case a partition is materialized

before applying a wide transformation: if this RDD is persisted after applying the `map` and worker 2 goes missing, only partition 2 needs to be reconstructed before applying the `reduceByKey` to reconstruct partition B.

2.4 State of the Art on Debugging

In this thesis, we study debugging support for Big Data applications. This section provides a state of the art on debugging, particularly focusing on debugging distributed programs and Big Data applications.

As identified by Zeller in the book "Why Programs Fail" [Zel09], every failure in a program is caused by an infection, which in turn is caused by an earlier infection, till reaching the erroneous code that generated this *infection chain*. *Debugging* is the process of identifying the defect that causes this infection chain (i.e., the root cause), and removing the defect so that it will no longer affect the program. According to Zeller's definition, the word *bug* has different connotations: *bug* can refer to a defect, i.e., an incorrect program code; an infection, i.e., an incorrect program state; a failure, i.e., an observable incorrect behaviour. Overall, the process of debugging refers to investigating any of the three aspects of a bug.

2.4.1 Debugging and Debugging Techniques

There are different techniques for debugging programs, depending on which point in software development they are applied. *Static* techniques apply before executing the program, *dynamic* techniques during the program execution, and *hybrid* techniques combine a static execution with data from a dynamic one.

Static analysis techniques focus on finding incorrect execution paths, so paths that lead to incorrect behaviour, without actually executing it. For example, *symbolic execution* [Kin76] is a kind of static analysis that tries to detect which input values cause the execution of which branch of the code. It does so by substituting program values with abstract values and executing the abstract program with an abstract interpreter [Cou96]. This abstract execution then generates a series of constraints over symbolic values to determine how a particular path can be reached, thus possibly isolating the constraints over an input that leads to incorrect

behaviour. The *constraint solver* then solves the set of constraints related to a path, extracting one or more possible values for the symbolic variables that lead the execution to that path. While these techniques are useful for generating faulty inputs or for contract verification, it does not scale to programs of big size, which might include many execution paths. This problem is also referred to as *path explosion* [XZ+10, KHL10] .

Hybrid static analysis techniques, such as *concolic testing* [Sen07], aim to reduce path explosion by performing both a symbolic execution and dynamic execution of the program. After identifying symbolic variables, the program is executed dynamically, feeding arbitrary input values to it. Operations that affect symbolic variables are logged and used to re-execute the program symbolically to generate a set of symbolic constraints, as a normal symbolic execution would do, and path condition. The symbolic execution is thus limited only to the path that the execution took with those specific input variables. At this point, path conditions are negated and fed to a constraint solver to calculate input values that lead to a different execution path than the ones that were already tested. Similar to symbolic execution, *concolic testing* can find input values that lead to an incorrect behaviour while limiting the number of explored paths thanks to the use of concrete (dynamic) values in combination with the symbolic ones.

Finally, dynamic analysis techniques analyze a program based on the values of a concrete execution. They do this by monitoring the execution and gathering information about it. This information is then analyzed to help the developer investigate the root cause of a bug. Dynamic techniques can be automatic or controlled by developers. Examples of automatic dynamic techniques are *delta debugging* [ZH02] and *fuzz testing* [MFS90]. Delta debugging is based on the execution of unit tests. Given an execution that fails with a specific input, delta debugging tries to reduce the size of the input to extract which part of the input is the one causing an error, in an automated way. Different parts of the input are isolated and fed into the execution until the minimal set of input is detected to be the one causing the error. Fuzz testing, also called *fuzzing*, similarly tries to identify input values that make particular tests fail, by variating this input. Fuzzing was originally designed to test UNIX utilities but now evolved into a common debugging technique [KRC+18]. Practically, random inputs values are fed to unit tests, until a particular input can be

found to make the test fail. In this way, the input causing errors can be isolated, so to help developers fix the behaviour of the program.

The main example of dynamic tools that can control the execution is *debuggers*. Debuggers, on the other hand, are dynamic tools that allow developers to observe and control the execution of a program to inspect and analyze its state throughout its execution. They do so by offering primitives to finely step through the execution, inspect values of variables, and/or execute expressions in the context of the debugged execution.

There exist two families of debugging techniques: *offline* and *online* debugging. Offline debugging, also known as *post-mortem debugging*, typically allows developers to analyze the execution of a program after it finished running. Online debugging, instead, allows developers to analyze the execution of a program while it is running by letting them pause the execution and control it step-by-step. In literature they are also referred to as *event-based* debuggers [MH89].

In the remainder of this section, we discuss more in detail the different offline and online debugging techniques and their architectures through examples of concrete debuggers. We then focus on debugging techniques for Big Data applications, some of which include the use of a debugger in combination with dynamic analysis techniques.

2.4.2 Offline Debugging

One of the first definitions of post-mortem debugging was given by Gill as far as 1951 [Gil51] in the context of debugging EDSAC programs. Namely, he defines *post-mortem debugging* as the process of teleprinting the state of the program through a second program. Moreover, he describes how this process "gives to the programmer a static picture of the machine", that is used to investigate both order and numerical failure. The concept of post-mortem debugging has evolved since then, and post-mortem debuggers are now largely used in industry to debug applications in many domains, from cloud computing to operating systems [Pac11].

In this section, we describe the main offline debugging techniques. We first talk about log-based debugging solutions, then move our focus to *Record & Replay* debugging approaches and finally describe *checkpoint-based* debugging.

2.4.2.1 Logs and Dumps

Offline debugging often involves capturing information of the program execution in a log for later analysis. The easiest form of populating a log is to add print statements in the code to extract and print state from variables, or to verify which path of the execution was taken by the program. This technique is called *printf debugging*. While it has been often perceived as the best method to debug [BSSZ18], especially for inexperienced developers, this technique is limited by the extent of the print statements, and the amount of information available at the moment of printing. Furthermore, this debugging technique requires modifying the code to be applied, and possibly having to track back and remove print statements after debugging.

It is the responsibility of the developer to wisely choose what to log: capturing too little information may require many debugging cycles to find the root cause of the bug, while too much information may add too much noise to the analysis [Pac11]. Logging can also be automated by automatically reporting in the log when particular events occur. For example, runtime errors are normally systematically logged in the form of a stack trace.

Another log-based debugging technique is *core dumps*. Instead of printing information explicitly, core dumps represent a dump of the state of the program, captured, for example, at the moment of a crash. Thus, the log does not only contain information about an event (e.g., a crash) but also about the state when an event happened. Hence, core dumps normally include more information than what print statements can capture. A core dump is often generated by the operating system and includes the values of the registers, as well as information on the call-stack and the actual memory dump of the crashed program. Core dumps often need to be loaded into other debugging tools such as GDB [GNU] to be analyzed.

Despite being popular, log-based approaches are limited by the amount of information available at a specific moment, thus they do not provide enough contextual information to find the root cause of a bug [Pac11].

2.4.2.2 Trace-based Debuggers

Another popular offline debugging solution is trace-based debugging, which is based on a trace of program events such as method activations and pa-

rameter values, recorded during the execution. A trace is automatically recorded during the execution and can be later browsed [BL07] or replayed through a Replay Debugger.

Record & Replay debugging is based on the deterministic replaying of a recorded execution, i.e., a replayed execution that follows the recorded trace. This means that the incorrect behaviour needs to be captured in the traced execution. If this is not the case, the execution may be recorded and replayed multiple times. During replay, Record & Replay debuggers often offer capabilities typical of online debuggers, such as state inspection and step-by-step execution.

Replay debuggers are especially useful when programs present non-determinism, as is the case for many concurrent and distributed programs. Non-deterministic inputs are dealt with by storing a partial order of variable accesses or events in the trace. In this way, the execution is replayed correctly. Trace recording, however, typically introduces a high overhead in the execution, and its scalability depends on the granularity of the trace[MH89].

Literature has also focused on reducing the size of the trace [WPP+14], for example by iteratively applying finer-grained traces on interesting parts of the program.

When all variable changes are stored, the debugger becomes *omniscient*, enabling back-in-time debugging. As an example, TOD [PT09] records all variable changes of the program in a trace, storing it in a distributed database. While the database enables TOD to make fast queries for inspecting a specific variable state in the past, it also introduces a further memory overhead. This is why other approaches that do want to achieve some level of omniscience, try to limit the number of variables that will be tracked. Actoverse [SW17], on the other hand, lets the developers annotate the fields of an actor that they want to trace, thus limiting the size of the trace.

2.4.2.3 Checkpoint-based Debuggers

In contrast to Record & Replay debuggers, *checkpoint-based* debuggers do not record an execution trace but focus on the program state but are based on the recording of application state in a checkpoint, i.e., a snapshot of the execution similar to a core dump. Those checkpoints are used by the debugger to analyze or replay the execution.

For example, Igor and DeLorean [FB88, MVB+16] periodically checkpoint dirty memory pages, i.e., memory pages that were changed since the last checkpoint. In this way, the execution can be restored to a specific checkpoint. As is the case for core dumps, Checkpoints can become heavy depending on how big the application state is. DeLorean tries to tackle this by optimizing the checkpoint size by not storing multiple times those variables that do not change.

Finally, some debugging techniques combine trace recording with snapshotting. An example is Jardis [BMM+16], a debugger for Javascript, which records a trace of I/O events and combines it with state snapshots of the event-loop taken at regular intervals (i.e., every few seconds). Debugging is then provided starting from the oldest stored snapshot and replayed using the recorded trace. To limit the size of the trace it removes old snapshots after a few seconds, thus also limiting the amount of replayable execution.

2.4.2.4 Discussion

Offline debugging techniques, especially the ones based on traces and checkpoints, present some interesting concepts for finding the root cause of bugs and have been applied to Big Data applications (cf. Section 2.4.5.2). On the one hand, Record & Replay allows developers to deterministically replay an execution until the root cause of the failure is found. On the other hand, checkpointing the state of the application provides the debugger with contextual information, that can be used by the developer to better understand the execution and where it deviated from the expected state while minimizing the number of replays. Recording a trace or performing a checkpoint, however, often has a performance and memory impact on the execution of an application.

2.4.3 Online Debugging

Online debugging, often called *breakpoint-based debugging*, allows developers to control the execution of the target program by marking "interesting points" of the execution at which the program can be paused, known as *breakpoints*. Once the program is paused, the debugger typically offers commands to (1) inspect the state of the program, often giving access to some data (*e.g.,* a stack trace) that helps to understand how the pro-

gram reached the current breakpoint, and (2) to control the debugged applications step-by-step using operations such as stepping into or over a particular call.

Through an online debugger, developers also interact with the program state. This happens via inspecting the value of variables, or by executing expressions and analyzing their result. This is a very powerful instrument in program comprehension, but it can also introduce side-effects since developers can change values of variables, or execute through expression evaluation some execution path that was not originally going to be executed by the program.

From an architectural point of view, we categorize online debuggers into two families: in-place (or in-process) and remote debuggers. Figure 2.3 shows the two architectures.

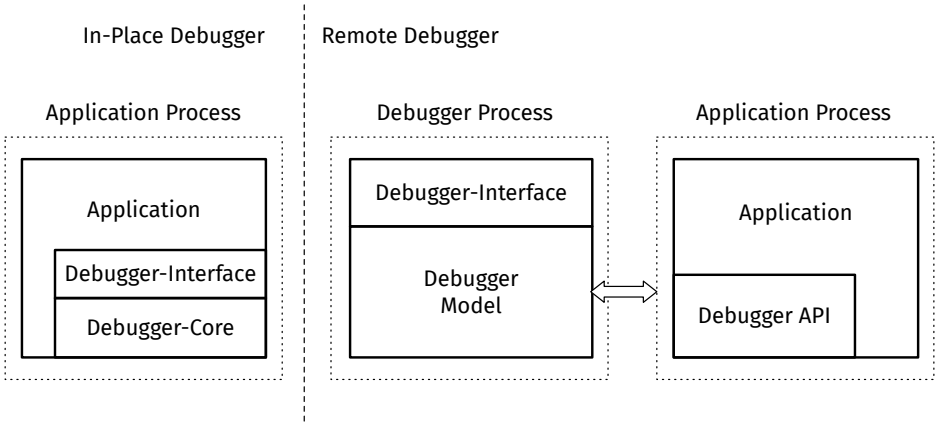


Figure 2.3: In-place and remote debugging architectures.

2.4.3.1 In-place Debuggers

An in-place debugger is an online debugger that executes in the same process as the application. It shares an address space with the application and can directly access its data and control its execution. Developers control the debugger through the Debugger-Interface, i.e., a graphical user interface or a command-line interface. The Debugger-Core includes the components of the debugger that instrument the running application.

With this architecture, developers can typically modify through the debugger all objects of the application, including classes, instances, environments, and in some cases runtime contexts. Examples of in-place debuggers include mainstream debuggers such as the ones for Python, Perl, and Pharo [Fou, Por, BNDP10].

Since the debugger runs on the same process as the executed application, the developer does not experience latencies when applying operations during a debugging session (e.g., stepping into a method). This results in a generally good user experience since the debugger is highly interactive and provides immediate answers to the issued debugging commands.

On the other hand, to operate such debuggers, developers need to have direct access to the application process. For instance, debugging with Pharo's in-place debugger requires a screen and keyboard plugged into the machine that is being debugged. To overcome the need for direct access, a second architecture was designed: remote debuggers.

2.4.3.2 Remote Debuggers

A remote debugger is an online debugger that controls the execution of the debugged application from a separate process, i.e., the *debugger process*. The debugger process offers the same commands and features to the developers as an in-place debugger through its debugger interface. It does this by dividing the Debugger Core into two components: the Debugger API, running in the application process to instrument the application, and the Debugger Model, running in the debugger process to transmit debugging commands from the Debugger Interface to the debugger API. In the resulting architecture, the target application is thus instrumented by the debugger API that receives in turn its commands from the debugger model. Examples of such debuggers are JPDA [Ora] for Java, GDB [GNU] for C/C++/Objective C, Visual Studio remote debugger [Micc] for .NET, and Mercury [PBFD15] for Pharo Smalltalk.

The main benefit of this architecture is that it allows the debugger to be deployed either on the same machine (typically a development scenario) or remotely, i.e., deploying the two processes on different machines connected over the network. All debugging operations in a remote debugger, however, require inter-process communication between the debugger and the application process. As such, users may experience extra latency of the debugging operations for communication delays especially when de-

bugger and application are deployed on different machines connected over the network.

2.4.3.3 Discussion

In contrast to offline techniques, online debuggers capture the context of an error at the moment that it manifests and provide tools to further explore the program execution such as stepping commands, expression evaluation, and state inspection. This removes the replaying time typical of offline debugging approaches. The operations performed during a debugging session with an online debugger may introduce side-effects and delays in the execution. This is often referred to as *probe effect* [Gai86]. These may alter the behaviour of an application and affect the reproduction of a bug, especially in a distributed and/or concurrent setting.

2.4.4 Comparison

To compare online and offline debugging approaches, we define different properties and analyze how the different debugging models apply them.

Capture Error Context. Approaches with this property support an active capturing of the error context upon an unhandled exception. When a program fails, online debugging approaches capture the state of the execution, to enable debugging. Offline debugging approaches, instead, capture a trace to later reproduce the error, or rely on the last captured checkpoint to replay from it.

Remote Access. Approaches with this property support the debugging of remote executions, e.g., on a cluster.

Latency. This property indicates the amount of latency a debugging approach introduces to the program execution. Trace recording, checkpointing, using breakpoints, and performing stepping operations affect differently execution times by introducing delays or pauses.

Residual side-effects. This property indicates whether a debugging technique introduces side-effects in the program execution as a result of debugging operations (*e.g.,* assigning a variable, writing to an output stream). Once such side-effects are applied, in traditional online debuggers they are not rolled back automatically and may affect the

behaviour of the debugged program when it is resumed. Residual side-effects are problematic because they alter the application context, making it more difficult to reproduce the original bug. We say that residual side-effects are *global* when they directly affect the application context and *scoped* when they are limited to a different environment.

Table 2.1: Overview of debugging techniques and their characteristics.

Debugging Technique	*Capture Error Context*	*Remote Access*	*Latency*	*Side Effects*
Printf Debugging	✗	~	Low	Global
Dumps	~	~	Low	N.A.
Record & Replay	✗	~	High	Limited
Checkpoint-based	✗	~	High	Limited
In-place	✓	✗	Low	Global
Remote	✓	✓	High	Global

Table 2.1 summarizes the different debugging approaches and their properties. In particular, notice that offline debugging approaches other than dumps do not capture directly the context of the error, since they involve a re-execution to access the state of the failing program. Similarly, they do not explicitly support remote access, although this is supported when log files, dumps, traces, and checkpoints are shared across remotely connected machines. While dumps manage to capture the error context, these are often generated by the OS, including part of the current program's memory without any abstraction (e.g., pointers instead of variable names), but also system calls and lower-level information. For these two properties, both online debuggers support the direct capture of the error context. In-place debugging, however, does not support remote access by design, which is instead fully supported by remote debugging approaches.

Regarding latency, both record & replay debugging and remote debugging introduce high latency. The first one is due to the trace recording, the second one is due to the network communication required for every debugging operation. Printf and in-place debugging approaches, instead, present a low latency because they do not require heavy trace recording, and all debugging operations, if any, happen locally. Finally, residual side-effects are global to the application state for all debugging approaches

that generate them. For instance, executing something in the context of the printing operation when debugging using printfs can alter the context of the executed application. Similarly, debugging operations such as expression evaluation in online debuggers (in-place and remote) also have a global side-effect. Debugging through dumps, Record & Replay, and checkpoint-based approaches does not directly affect the execution, but recording a trace or a checkpoint could add side-effects to the computation by adding delays that potentially alter the behaviour of the debugged application.

In this dissertation, we focus on Big Data applications, which often run remotely in a cluster of machines. For this reason, we will focus on approaches that allow remote access. Debugging approaches for Big Data applications should also scale to the different characteristics of such applications. For this reason, in the next section we give an overview of bugs in Big Data, and then describe several debugging approaches that can be found in the literature.

2.4.5 Debugging Big Data Applications

Big Data frameworks, as described earlier in this chapter, simplify the development of distributed programs by providing a programming and execution model that abstracts away several complex concepts such as distribution, parallelization, and fault tolerance. We now turn our attention to debugging support for Big Data applications. To this end, we first describe common errors in Big Data applications and then delve into current debugging approaches.

2.4.5.1 Bugs in Big Data

Bugs in Big Data applications normally manifest in several ways. Big Data frameworks are, in fact, often subject to a complex configuration. Misconfiguration is often the cause of failures in different stages of the computation [RK13]. Furthermore, this is amplified by the recent trend of executing Big Data processing applications in the cloud. Zhou et al. [ZLZ+15] analyzed what issues are reported in cloud processing services, showing that the majority of reported failures are generated by hardware and (hosting) system side problems. The 37% of reported failures, how-

ever, are attributed to customer-side code, i.e., to errors introduced by the developers. They further categorize these errors in three types:

- *code defect*, when the error is related to buggy code, i.e., explicit errors inserted by the developer.
- *operation fault*, when the error is caused by common operational mistakes, e.g., non-intentional data deletion, file renaming, missing resources or data, etc.
- *misuse*, when the error is caused by configuration errors, e.g., using a wrong library version or a wrong proxy configuration, improper input parameters, and improper system assumptions (e.g., overestimating the capabilities of the system).

Interestingly, Zhou et al. remark that almost 40% of reported errors are introduced by the the developers. The authors find this surprising, since customers of a cloud computing service tend to report only problems that they think are not caused by themselves.

Besides the aforementioned study [ZLZ+15], other studies [BK19] show that debugging Big Data applications is not as easy as it looks and that bugs can take many forms, i.e., application bugs, configuration bugs, and bugs that are not caused by developers but by the service they use to run their programs.

Finally, another recent study has shown that developers spend hours trying to debug data-cleaning errors [MLW+19], finding that minor programming bugs or a handful of fail-inducing records in the analyzed data are often traduced in hours of lost computations. In these cases, however, simply ignoring such errors at run-time may present little or no impact in the final results for many classes of Big Data applications. It is already common practice in data analytics AI algorithms to gain performance by sacrificing some accuracy [Mit16]. For example, Chippa et al. [CCRR13] show that a k-means algorithm may run up to 50x faster by giving up 5% of accuracy.

2.4.5.2 Overview of Debugging Approaches for Big Data

We now describe current debugging support tailored to Big Data applications, and, more in general, Big Data frameworks. Literature has

focused especially on dynamic automated debugging techniques, hybrid static analysis techniques, and debuggers.

Dynamic automated and hybrid static analysis. The goal of dynamic automated and hybrid static analysis approaches is to generate a set of the inputs that cause the execution to fail, adapting known approaches such as concolic testing and fuzzing to Big Data applications. For instance, Kaituo et al. propose Sedge[LRS+13], a tool that analyzes Pig Latin programs (i.e., a SQL-Like language that executes on top of Map/Reduce) by producing input values, and executing an adapted concolic analysis to generate through the analysis those input values that lead to an erroneous path. The analysis of Sedge, however, is limited to Pig Latin queries and does not support the analysis of UDFs (i.e., User Defined Functions). Gulzar et al. [GMMK19] propose instead an approach to generate faulty input data by similarly analyzing Spark applications through a hybrid analysis. The analysis takes care of combining a logical analysis of the relational operations of the Spark model, such as join and group-by, with the symbolic execution of user-defined functions. Zhang et al. [ZWG+20] instead propose to use fuzzing to find faulty input in Spark programs. Before applying fuzzing, however, BigFuzz performs an analysis of the AST of the application to produce the same application in an executable specification. A guided fuzzing step is then applied starting from user-defined input as seed, modifying the input data respecting its structure, so to find meaningful variations of the input that cause errors.

Debugging techniques. On the other hand, in the literature we can find different examples of debuggers for a variety of Big Data frameworks. Different debugging solutions for Big Data rely on replay or partial replay of all or parts of the distributed computation [DZSS13, SSK+15]. Some [GIY+16, JYB11] offer online debugging primitives in the remote debugging setup, since these applications are usually deployed on a remote debugging architecture. These approaches hence require often more than one execution to reach the moment of the bug. In long analysis programs such as the ones usually run on Big Data frameworks, this might require some time.

Record & Replay approaches for Big Data programs rely on the deterministic execution of these models, in which the runtime uses a directed

acyclic graph (DAG) of operations on certain data partitions, thus generating a trace of what was executed on certain data partitions. The Arthur [DZSS13] debugger, for example, uses this information to limit the amount of replayed execution to those jobs that are ancestors of the one the developer wants to debug. The replaying, however, needs to be performed on all ancestors, and this can be costly [GIY+16]. Arthur also allows developers to replay locally a recorded execution by retrieving data from the remote execution and enabling stepping operations through a conventional debugger.

Another approach to limit the amount of recorded execution is employed in Graft [SSK+15], a debugger for Apache Giraph, a graph processing system aimed at Big Data processing. Particularly, when using Graft developers select beforehand those computation vertices that they want to be captured, and the code is instrumented accordingly. After data is captured, developers browse it through the debugger's visualization. For debugging, Graft generates a test case from the trace of the recorded execution in which global variables are mocked. Developers then run this test using a classic online debugger, stepping line by line. If code is missing, however, the developer needs to copy-paste it into the debugging environment as the mocking does not substitute their local variables.

More recently, Daphne [JYB11] and BigDebug [GIY+16] explored online debugging support for Big Data applications. While Daphne employs remote debugging, BigDebug adds breakpoints to a checkpoint-based debugging solution.

Daphne is an online debugger for DryadLINQ [Micb] which provides a runtime view of the running system and the query nodes generated by a distributed LINQ query. It allows developers to inspect the program state through breakpoints and to guide the execution through stepping commands using the Visual Studio remote debugger Debugging is done remotely on the worker where the breakpointed node is executing, interrupting it to debug it. Daphne also offers a mode to replay an execution locally to the developer's machine. This is enabled by retrieving data from the halted vertex and replaying its execution under the Visual Studio debugger.

BigDebug [GIY+16] is a debugger for Apache Spark [Apad] which introduces online debugging primitives such as breakpoints and stepping on top of a checkpoint-based backend, that relies on checkpoints taken

during the execution and data provenance to enable debugging. BigDebug represents our closest related work and, as we will detail in Chapter 8, we compared our approach to it in an experimental user study. For this reason, in the next section we present BigDebug with more details than the other approaches.

2.4.5.3 BigDebug

BigDebug's philosophy is to limit execution replay, which can take a lot of time in the Big Data application domain, by leveraging on the execution model of Spark. It offers online debugging primitives on failed executions and *simulated breakpoints* triggered during the execution. Below we describe in detail the main functionalities of BigDebug.

2.4.5.3.1 Debugging a Failed Execution The debugging model of BigDebug is based on the concept of *failure-inducing records*, i.e., the records that caused the incorrect behaviour of the program. When an exception is thrown in a worker, BigDebug captures the intermediate record that causes the exception, returning it to the driver where the debugger is hosted and continues the execution. The execution, however, stops when all the other records reach the end of the current stage.

At this point, the developer can browse through the intermediate record(s) that caused an exception during the execution, the lineage of the current RDD (i.e., the series of transformations applied before the failing one), and a stack trace of the exception. From there, the developer can skip or manipulate the failure-inducing record. When skipping a record, it is not included in the rest of the computation; when repairing it, it will be replaced with another record indicated by the developer in a textual representation. The developer can also indicate a function through the lazy code fix functionality, to apply with the failure-inducing records as parameter to generate the record that substitutes it in the next computational stage.

Overall, these primitives offer control to the developer to fix failure-inducing records before the computation is aborted, while avoiding replay.

2.4.5.3.2 Simulated Breakpoints As part of their online debugging features, BigDebug offers to developers *simulated breakpoints*, i.e., special breakpoints that do not halt the execution. Simulated breakpoints take

advantage of the execution model of Spark: when a breakpoint is hit, the breakpoint stores the lineage of the current RDD since the last *materialization point*, i.e., the last point in which the current RDD (or its parent) was persisted. When the data from the breakpoint is requested, BigDebug replays the execution from the last materialization point to capture the breakpointed record and allows the developer to inspect a textual form of the record. On a breakpointed execution, upon retrieving the breakpointed record, the developer can also step over the halted transformation to its result or resume the execution.

2.4.5.3.3 Guarded Watchpoints BigDebug includes guarded watchpoints to inspect the state of the execution. The idea behind this is to capture all records that pass by the watchpoint and that satisfy the associated guard, i.e., a filtering predicate that returns true for all records that satisfy it. When a record passes the guard of the watchpoint it is stored in the worker and sent in batches, upon request, to the driver node where the debugger is hosted. Using a guard also ensures that not all of the records will be captured in the watchpoint, since this will force the entire dataset to be captured and transferred to the driver. The code of the guard can be updated through the debugger if different records want to be matched. In case a simulated breakpoint is inserted after the watchpoint, only the records passing the guard of the watchpoint will be captured in a simulated breakpoint.

Simulated breakpoints can only be added to the execution after a guarded watchpoint, i.e., only records that pass the guard of the watchpoint are captured by a simulated breakpoint.

2.4.5.3.4 Code Patching Upon a simulated breakpoint, BigDebug allows the developer to change the code of an applied transformation in the current lineage. Particularly, the developer can change the lambda function applied by a certain transformation to a new one, respecting the same type signature. To do this, the developer interacts with the debugger UI, inserting the new code to be applied into a template.

When the execution is resumed, BigDebug restarts the execution from the latest materialization point, using the new code for the changing transformation.

2.4.5.3.5 Backward tracing BigDebug also includes the backward tracing of a failure-inducing intermediate record, i.e., discovering the original dataset record that produced the now failure-inducing intermediate record. The tracing functionality is enabled by data provenance, implemented in Spark in the Titian framework [IST+15]. A trace is recorded by instrumenting the execution with tracing agents, that tag input and output data at the boundaries of a stage. The produced trace stores the identifiers in a provenance table, thus associating identifiers with each input and output record.

Tracing is performed on an output record and happens by joining the different provenance tables across the different transformations that are associated with that record. In this way, the debugger tracks the original record(s) that originated the analyzed output record. BigDebug allows both backward and forward tracing, so navigating such trace backward until the original record, or forward to the final result produced by the same record.

The same authors of BigDebug extended the data provenance functionalities of BigDebug in BigSift [GWK18] by adding test-based data provenance, i.e., data provenance based on the results of a custom test. After the execution of a program, the developer can write a test that will capture some of the final records that, for example, they believe to be faulty. Based on the result of this test, an optimized backward trace is applied to find the original records responsible for those final results.

2.5 Criteria of Debugging Approaches for Big Data Applications

As described in Section 2.4.5.1, Big Data applications are notoriously difficult to debug due to their distributed execution, the complex technological stack, and the amount of data they analyze. We believe that many of these errors, qualified by Zeller [Zel09] as *minor* and *trivial* problems, can be easily solved in local applications using interactive debugging tools. However, when those errors are present in Big Data programs, solving them with the current state of the art debugging tools becomes a time-consuming task even though the fix may be trivial because current debuggers require replaying at least part of the execution to enable the identification of a bug.

Particularly, building on the common errors in Big Data described in Section 2.4.5.2, we describe the criteria respected by debuggers for Big Data applications.

Scalability to data. Debugging approaches for Big Data applications should scale to the amount of analyzed data. For example, a debugger needs to have a trade-off regarding how much data is retrieved and displayed. This is a core difference between debuggers for Big Data applications and other debuggers for distributed systems.

Low Replay Time. Debugging approaches for Big Data applications should limit the amount of replayed execution to enable debugging, avoiding replaying long running applications from the beginning.

Halt & Inspect. Debugging approaches for Big Data applications should support halting and inspecting the state of all intermediate variables, as it is possible in many debugging solutions for sequential programs.

Stepwise Execution. Debugging approaches for Big Data should offer classical sequential operations on a debugged execution to let developers step into the execution of every line of their code.

Domain-specific Operations. Debugging approaches for Big Data applications should support domain-specific operations, for example, to step through the concepts of the framework's execution without using multiple sequential stepping operations.

Code Updating. Debugging approaches for Big Data applications should enable live updates of a remotely running application without having to repackage and redeploy the application or the whole system. This reduces the time of debugging new versions of an application.

Ignore Errors. Systematical ignoring of errors can help deal with tedious data-cleaning errors very present in data science applications [MLW+19] and, in general, errors that impact a low amount of data that doesn't impact the result of an analysis. As shown in a paper by Chippa et al. [CCRR13], many AI applications such as K-means, SVM training, and GLVQ, present high resilience to incorrect computation (i.e., injected failures or approximations). Thus, a debugging approach for Big Data should support for ignoring failures, within limits easily configurable by the developer.

Table 2.2: Survey of the related work based on the criteria.

Debugger	*Side Effects*	*Replay Point*	*Halt & Inspect*	*Stepwise Exec.*	*Dom.S. Ops.*	*Code Updates*	*Ignore Errors*
Arthur	Both	Start	✗	✓	✗	✗	✗
Graft	Both	Start Rec.	✗	✓	✗	✗	✗
Daphne	Both	Check.*	✓	✓	✗	✗	✗
BigDebug	Global	Check.	✓*	✗	✓	✓*	~

In Table 2.2, we present an overview of related work on Big Data debuggers described in Section 2.4.5.2 with regards to the criteria defined above.

Regarding side-effects, all debuggers introduce in some mode global side-effects. BigDebug introduces global side-effects on the debugged execution since this happens in the context of a breakpoint. Daphne, Arthur, and Graft, instead, present global side-effects when debugging remotely, but local side-effects when replaying part of the execution locally.

Regarding replays, the two Record & Replay solutions (Graft and Arthur) need to replay either from the beginning or anyway from the beginning of the recorded execution. Daphne and BigDebug instead only need to replay from the last checkpoint. Daphne, however, represents a special case, since replay is limited to the local replaying of the execution, and not to the debugging of a remote breakpointed execution, which is not the case for BigDebug.

Daphne and BigDebug offer breakpoints to halt the execution. BigDebug, however, does not support the inspection of the state of variables, expression evaluation, and classical stepping into the execution.

Arthur, Graft, and Daphne also offer support for fine-grained stepping operations, so stepping through a classical debugger. In all of them, this is limited to the locally replayed execution and happens through a classical debugger.

BigDebug is the only one of the analyzed debuggers providing domain-specific debugging operations, i.e., stepping across the constructs of the distributed execution, and a form of live code updates, albeit limited to the updating of single lambdas applied by transformations, and not to general code changes to the program.

Finally, to the best of our knowledge, systematic ignoring of errors, i.e., embedded in the execution and debugging of Big Data applications, is not currently supported in the state of the art. BigDebug does support the skipping of records that caused an exception, but this is a manual operation to be done record by record and cannot be done systematically on, for example, a predefined number of records.

Based on our analysis of related work, in this dissertation we propose a novel online debugging approach that offers all adheres to all the identified criteria debugging Big Data thus allowing developers to:

- Debug immediately a failing execution, without replays.
- Debug in isolation, scoping the side-effects to the debugged execution.
- Set breakpoints into the execution to debug at specific points.
- Offer both classical and domain-specific stepping operations in any of the debugged executions (failed or breakpointed).
- Update the code of any part of the remote application without having to redeploy it.
- Allow developers to ignore a predefined number of errors that would raise in the execution.

2.6 Conclusion

In this chapter, we first presented the state of the art on Big Data frameworks, focusing on the Map/Reduce and Spark models that we aim to debug with our debugger. Then, we moved our focus onto the state of the art of debugging, starting from the definition of bugs and moving onto offline and online debugging approaches. Moreover, we discussed domain-specific debuggers for Big Data applications and describe criteria for a debugging approach to be suitable to Big Data applications. Chapter 5 presents a live debugging approach for remote applications complemented with live code updating. Chapters 6 and 7 further explore applying our debugging approach to two Big Data models: Map/Reduce and Spark.

Before describing our debugging approach, we present in Chapter 3 the frameworks on which we built our debugging support for Map/Reduce and Spark-like applications.

Chapter 3

Scalable Big Data Frameworks for Pharo Smalltalk

In this chapter, we introduce the two Big Data frameworks that we build debugging support for. As our research platform, we use Pharo Smalltalk (Pharo for short), a modern and popular open-source implementation of the classic Smalltalk-80 [Gol84].

Recall from Chapter 2 that Map/Reduce and Spark are both based on the Master/Worker model. In what follows, we thus describe the implementation of a Master/Worker model for Pharo Smalltalk, as part of an infrastructural layer that we use to develop both Port and Spa: our two Map/Reduce and Spark frameworks in Pharo. Before delving into the details of our frameworks, however, we briefly describe Pharo Smalltalk.

3.1 Experimental Platform: Pharo Smalltalk

Pharo is a class-based object-oriented dynamically-typed programming language, in which (almost) everything is an object, including classes and methods, compiler, AST, etc. The object model is class-based with single inheritance and supports code reuse through stateful traits. Pharo is single-threaded and models the execution with so-called *processes*, representing a green thread running in the same OS process of the Pharo virtual machine.

Being a Smalltalk implementation, Pharo embraces the concept of the interactive programming environment, by offering an integrated development environment (IDE) to write, read, execute, test, and debug code. The environment is image-based, so the state of the IDE itself is stored in a full snapshot of the heap and can hence be restored at any time after saving it.

Furthermore, Pharo is a reflective language, with primitives for the reification of methods, class, and runtime structures such as the call-stack. Many of the tools that are part of Pharo are implemented using the reflective capabilities of the language, including the Pharo Debugger. This is an in-place debugger, i.e., it executes in the same process of the application, and is used interactively by developers to debug both breakpoints and unhandled exceptions. The Pharo debugger offers access to the execution stack, and all the variables referenced at each of its frames. Besides the classical stepping operations, it also provides an evaluator in the context of the currently selected stack frame, in which developers execute or debug arbitrary code. Furthermore, it offers a *restart* operation, that, when selecting a stack frame, restarts the execution from the selected frame, allowing to re-execute previous calls. Finally, it supports live code updating, thus letting the developers change the code of a method in the debugger and its reloading through the restart operation.

3.1.1 Pharo Syntax and Constructs

This section provides a summary of the Smalltalk syntax, that should help the reader understand the code listings of the rest of this dissertation in case they are not familiar with Smalltalk.

The listing below defines the `sum` method in the `Foo` class[1]. In Smalltalk, the period (.) is the statement separator. Local variables, `a` and `b` in this example, are defined directly after the method name between pipes (||). In line 3, an instance of the `Foo` class is created by sending the message `new` to the same `Foo` class. Classes are in fact also objects that can understand messages. Line 3 assigns the value 1 to the variable `b` through the `:=` operator. Finally, line 4 returns the result of the method, i.e., the sum between `b` and `c`, through the `^` operator.

[1]For displaying method signatures, we use the convention `Class >> methodName`

```
1 Foo>>sum
2  |a b|
3    a := Foo new.
4    b := 1.
5    ^ b + 1
```

Methods and Messages. In Smalltalk, objects represent are instances of one class and communicate by sending *messages*. Messages are looked up in the inheritance chain of the class of the receiver object, starting from the class of the receiver[2].

There are three kinds of messages, depending on the number of parameters. Unary messages, such as `new`, have no parameters. Binary messages have exactly one parameter and are named by symbols such as `+` and other mathematical operators. Finally, keyword messages accept one or more parameters encoded in their name. For example, the listing above shows four lines in which the different kinds of methods are called.

```
1   foo := Foo new.
2   bar := Bar new.
3   sum := 1 + 2
4   res := bar sumFoo: foo withParameter: sum.
```

For instance, the first and second lines send the unary message `new` respectively to the `Foo` and `Bar` class to instantiate them in the `foo` and `bar` variables. The third line sends instead the binary message `+` to 1 with 2 as a parameter to calculate their sum and store it in the `sum` variable. The last line sends the keyword message `sumFoo:withParameter:` to an instance of `Bar` (i.e., the `bar` variable), with `foo` as the first parameter and `sum` as the second parameter.

The listing below shows the definition of the keyword method `sumFoo: withParameter:`, i.e., method with two parameters: `aFoo` and `anInteger`[3].

```
1 Bar>>sumFoo: aFoo withParameter: anInteger
2   ^aFoo sumWithParameter: anInteger.
```

[2]If no method is found in the inheritance chain, then the `doesNotUnderstand:` message is sent to the receiver of the original method. The default behaviour of `doesNotUnderstand:` is to throw a runtime exception.

[3]In Pharo it is a convention to encode, when possible, the type of a variable in its name, so to increment the readability of the code. In this example, `aFoo` and `anInteger` indicate that the two variables will have an expected type of `Foo` and `Integer`

In line 1 the method is defined through two keywords, i.e., `sumFoo:` and `withParameter:`, and the name of each parameter after a colon. In line 2, this method returns the result of calling `sumWithParameter:` with `aFoo` as the receiver, and `anInteger` as the parameter.

Closures Finally, Pharo features closures, called `BlockClosures` or more familiarly *blocks*. For example, we show below a definition of a closure that takes two parameters, `a` and `b`, and returns their sum.

```
1 sumBlock := [:a | a + 1]
2 sumBlock value: 1.
```

A closure is defined within square brackets and starts with the definition of each parameter of the block, each preceded by a colon. A pipe separates the definition of the parameters from the actual body of the block. The body can contain multiple statements, separated by the statement separator (`.`). A block returns the last statement of its body. An explicit return of a block with the ^ sign returns the method that defined it. A block is called using the `value` message, or `value:`, `value:value:`, etc. when it accepts one or more parameters. For example, in the second line of the above listing, we call `value:` with the parameter 1 to execute the `sumBlock` closure.

Smalltalk languages generally include very few control flow statements. Common control flow patterns such as if statements are implemented as messages sent to boolean objects with closure(s) as parameter(s).

3.2 The Infrastructural Layer

In this section, we describe the infrastructural layer used by both Port and Spa, our Big Data frameworks supporting a Map/Reduce and a Spark-like model, respectively. The infrastructural layer is composed of a runtime based on the classical Master/Worker model [GKYL01]. Communication between the master and workers happens through a custom communication protocol over HTTP. We use HTTP because it is a reliable protocol that is already used internally in Big Data frameworks such as Apache Spark. Furthermore, we can reuse client and server components already available in Pharo as a library. In what follows, we describe the runtime and the communication protocol that enable the execution of Port and Spa.

3.2.1 The Master/Worker Model in Pharo

Recall that Big Data applications are normally executed on a cluster of machines, so Big Data frameworks are normally constructed to run on different instances of a runtime, deployed on different physical machines. In the case of Hadoop Map/Reduce or Apache Spark, the runtime is based on the Master/Worker model in which the master and each of the workers run in a dedicated process, with no shared memory.

Map/Reduce and Spark, however, normally execute on multithreaded runtimes that can exploit the parallelization capabilities of the hardware where they run. This is not possible in Pharo because its runtime is single-threaded. As such, our master runs on a unique single-threaded VM, having fewer capabilities than, for example, a master in Apache Spark, that could employ all cores of the machine in which it is deployed. Similarly, each of the workers also runs on a single-threaded VM. To reach the capacity of parallelization in the machine(s) the workers are deployed on, we run multiple single-threaded workers on the same machine. This leads to more inter-processes communication between the workers and the master. Even though is not as performant as the JVM-based runtimes of Spark and Map/Reduce, Pharo provides us with an environment in which it is easy to experiment our debugging approach thanks to its reflective capabilities.

Similar to Apache Spark, in our runtime the execution is guided by a *driver*, also running on its own single-threaded VM. The driver submits different tasks to the master for running the application. It also handles, returns, and/or prints the results of the executed applications.

Figure 3.1 shows an overview of our runtime. Each dashed box represents a single-threaded Pharo VM execution running in an OS process. On the left side of the figure, there is the driver that includes an application runner to run a particular application. All the components include and use different Pharo libraries, such as the serializer of the Fuel [DPDA14] serialization library, the HTTP Client and Server modules of the Zinc [Zin] library, and various other libraries available in the Pharo runtime.

The core of the driver is the application runner, which communicates with the master through the Master Server component, running an HTTP server. Similarly, the master communicates with the different workers through their Worker Server and the worker communicates to the master through the Master Server.

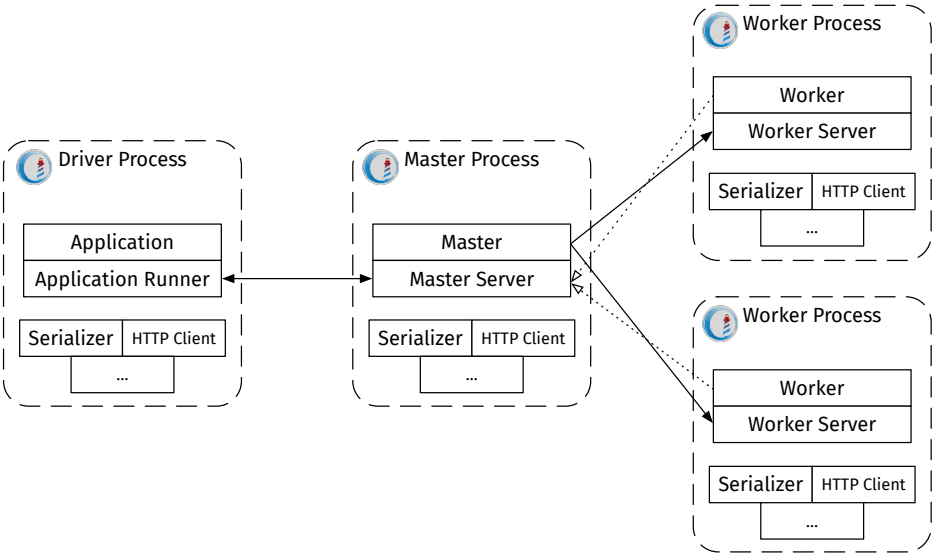


Figure 3.1: The distributed runtime of Port and Spa.

3.2.2 The Communication Protocol

All the entities of the Port and Spa runtimes communicate through a custom communication protocol based on simple synchronous and asynchronous messages on top of HTTP, i.e., the master and worker receive a synchronous HTTP request that they treat either synchronously returning the result within the answer to the request, or asynchronously by scheduling the execution and returning an acknowledgment. In practice, the master and each of the workers run an HTTP server on a particular port (fixed for the master, dependent on the *worker id* for the workers). Communication happens always through a single-use HTTP Client, that sends a POST request to the master or worker server. In the contents of the request, we include the message.

When the Master or Worker Server receives a synchronous message, they execute it immediately in the (green) thread that is handling the message reception. The result is then returned to the sender. When receiving an asynchronous message the master/worker schedules it instead in an internal thread by using the TaskIt library [Con]. TaskIt ensures that all asynchronous messages are executed sequentially, i.e. one by one.

The behaviour of the message handling is implemented as part of the message itself. Each message extends either the `AsyncMessage` or the `SyncMessage` class and reimplements a method (`executeOn:`), that takes a master or a worker as a parameter, to specify the behaviour of that message.

The Application Runner and the Master mainly communicate in terms of synchronous messages. The Master and Workers mainly communicate through asynchronous messages.

Messages are passed by copy among the different components of the runtime, thus they are serialized before being sent over HTTP. Serialization happens through Fuel [DPDA14], a general-purpose object graph serializer available as a library in Pharo. More in detail, Fuel serializes and deserializes object graphs using a custom pickle format [BH00] to cluster objects by their class before they are serialized. In this way, information about the class is stored in the cluster only once, thus reducing the amount of data that has to be serialized.

We rely on Fuel because it is stable, used by several other Pharo libraries, and relatively simple to use and configure to our use case. For example, in our configuration, Fuel does not serialize methods and supposes that classes (and methods) of the objects present in its graph have not changed between serialization and deserialization. This is because we assume that the master and the workers run the same codebase.

3.3 Running Example

This section describes a running example that we later use to describe how to program in Port and Spa. A classic example of a Big Data application is a *distributed wordcount*, a program that counts the number of times each word is repeated in a (distributed) file. In this thesis, we present a variation of such a wordcount application: an election poll analyzer, akin to the one presented by Gulzar et al. [GIY+16]. This application analyzes a dataset containing the results of the election polls and computes, for one region, the number of votes received by each of the candidates.

The election poll analyzer reads data from a CSV file, in which each line contains a region, the name of the candidate, and a timestamp, separated by commas. For the purpose of this application, we assume the file to be available at a certain path in all the nodes through a local, net-

worked, or distributed file system. Furthermore, the application should report results only for valid regions, returning a dictionary indicating how many times each candidate has been voted.

```
1  raw := FileSystem / filePath.
2  parsed := raw lines collect: [:line | line substrings: ','].
3  valid := parsed select: [:array | (self isValidTimestamp: array third) and:
       [self isValidRegion: array first] ].
4  mapped := valid collect: [:array| array second -> 1]
5  result := mapped reduceByKey: [:value1 :value2 | value1 + value2]
```

Listing 3.1: A sequential implementation of the election poll analyzer.

Listing 3.1 provides a sequential version of the polls analyzer application. Line 1 loads the file. Line 2 parses the file to split the lines on the comma character. Then, line 3 filters the valid records using `select:`, equivalent to a filter in functional programming languages. Line 4 maps each candidate name to one in a key/value pair using the `collect:` method, equivalent to a map. Line 5 sums all the values associated with the same key using `reduceByKey:`, resulting in a dictionary containing how many times each key (i.e., candidate) has been voted.

3.4 Port: A Map/Reduce Framework for Pharo

In this section, we introduce Port, the framework and programming environment that we built to write and execute Big Data applications using the Map/Reduce computational model in Pharo.

As described in Section 2.2, a Map/Reduce application is composed mainly of two functions: a *map* function, that transforms all the elements of the input collection, and a *reduce* function, executed after the map, that can reduce all the intermediate results to a final one. As for the popular Hadoop Map/Reduce, our implementation is built on top of the Master/Worker model described in Section 2.1.

In Port, the execution is controlled by the driver, that instructs the master to execute a Map/Reduce application. The master schedules the map tasks for this application, and, when all maps are finished, it schedules the reduce tasks. Before starting the map, the master assigns a portion of the input file(s) to each worker, that proceeds to load it. We assume that all workers have access to input files, expecting that they (i) all run on the same machine or (ii) on different machines that share the file system

(i.e., the data is in a folder mounted in all of the machines and shared over the network), or (iii) that the file is available on the HDFS distributed file system.

3.4.1 Map/Reduce by Example

A Map/Reduce application in Port is defined as a Pharo class implementing the methods `map:` and `reduce:`. Listing 3.2 shows the core code of the election polls analyzing application in Port. The `map:` includes all of the mapping and filtering code. It first splits the input string to parse it into an array with the `substrings:` method and then checks through a helper method whether the timestamp and region are valid. If so, it returns an association created using the operator `->` with the name as key and 1 as value. Otherwise, it returns an association of `nil` values.

The `reduce:key:` method is applied to a list of values that are associated with the same key and that key. If the key is valid (i.e., not equals to `nil`), it returns a key-value pair with the key (i.e., the name of the candidate) as key and the size of the list as the value. Recall that the framework handles the loading of the data and the parallelization for us, so the application does not include any code related to that.

```
1  PollsAnalyzer >> map: aLine
2      | splitted |
3      parsed := aLine substrings: ',',
4      ((self isValidTimestamp: parsed third)
5        and: [self isValidRegion: parsed first])
6          ifTrue: [ ^ (parsed second) -> 1 ]
7          ifFalse: [ ^ nil -> nil].
8
9
10 PollsAnalyzer >> reduce: aListOfVotes key: aKey
11     | dict |
12     aKey ifNotNil: [
13         ^ key -> aSetOfVotes size
14     ]
15
```

Listing 3.2: A Map/Reduce implementation of the election poll analysis application.

When the application is run, each entry in the input log files is first mapped by the `map:` method. The master instructs the workers to par-

tition and sorts the results by key before scheduling the `reduce:key:` method. The application finally returns a set of key-value pairs with the number of votes of each candidate.

3.4.2 Handling Intermediate Results

In Port, intermediate data can be persisted in memory or directly on the disk of a worker, depending on configuration parameters. This makes Port's handling of intermediate files more similar to the one of Spark than to other Map/Reduce implementations.

After a map computation, the resulting key/value pairs are physically at the worker that performed the map. To reduce by key in a distributed way, key/value pairs that have the same key have to be read by the same worker. Before scheduling reduce tasks, a shuffling step is executed. The master, knowledgeable of which worker holds which key, coordinates the transfer of data among the different workers using the key's hash to partition them.

Scheduling of tasks is important to tune the performance of Map/Reduce (cf. Section 2.2). However, since the focus of Port is on enabling debugging and not on full performance, the scheduler of Port does not include many optimizations. For example, the scheduler of Port is embedded in the master and schedules reduce tasks according to the location of intermediate data but does not schedule map tasks according to the locality of the input files.

3.5 Spa: A Spark-like Framework for Pharo

To support a programming and execution model similar to Apache Spark (i.e., Spark-like), we implemented a framework for Pharo Smalltalk using the same infrastructural Master/Worker layer as Port that we call Spa. As described in Section 2.3, the Spark model is based on the concept of a distributed data structure and on transformations and actions that manipulate their data. Hence, we introduce the support for a distributed data structure that we call DDD, similar to Spark's RDD. In this section, we detail how to write Spark-like programs in Spa and give some details about the adaptations to the original Spark model present in Spa.

3.5.1 The Spark-like Model by Example

Central to Spa's programming model is the concept of a distributed data structure (DDD), akin to Spark's RDD [ZCD+12], where applications are expressed in terms of functional operations on distributed data structures that are eventually executed in parallel by the infrastructure. Developers create a DDD by distributing a data-source (i.e., as a collection or a file) in a local or distributed file system, as shown in the following listing:

```
collection := 1 to: 1000.
collectionDDD := spa distribute: collection.
fileDDD := spa readFile: '/path/to/file/or/dir'
```

The `spa` variable is the entry point to the Spa framework. The contents of a DDD are stored in different partitions on the different workers, i.e., each of the workers holds a part of the content of a DDD in memory.

A Spa application is defined by extending the `SpaApplication` class, or by simply writing the code in the Spa playground (to be executed in the UI). Listing 3.3 shows the implementation of our running example, defined in Section 3.3.

```
1 raw := spa readFile: filePath.
2 parsed := raw map: [:line | line substrings: ','].
3 valid := parsed filter: [:array | (self isValidTimestamp: array third) and: [
    self isValidRegion: array first] ].
4 mapped := valid map: [:array| array second -> 1].
5 reduced := mapped reduceByKey: [:value1 :value2 | value1 + value2].
6 result := reduced getCollection.
```

Listing 3.3: An implementation of the elections poll analyzer in Spa.

As expected, the Spa implementation of the election polls analyzer is closer to the sequential code listed in Listing 3.1 than the Map/Reduce one listed in Listing 3.2. A Spa application is thus composed of a series of calls on the DDD generated by the `readFile:` message in line 1. The rest of the code follows the structure of the sequential implementation, until the `getCollection` message is sent in line 6. Since this is an action, it triggers the computation of all the (lazy) transformations called before and returns the resulting collection. In the following sections, we further detail how Spa handles the operations and persistence.

3.5.2 Actions and Transformations

As RDDs, a DDD supports two kinds of operations: *transformations* and *actions*. Both actions and transformations are functional and return a new data structure.

Transformations are operations that transform the data returning a new DDD. Common transformations include `map:`, `filter:`, and `reduceByKey:`. Transformations are lazy and return a new DDD that internally indicates that a transformation has to be applied to the original DDD. If more than one transformation is called in a sequence, all these transformations are pipelined. In Listing 3.3, each mapping operation returns a different DDD. For instance, after executing the first three lines, the DDD `valid` does not include data, but only the reference to the previous DDD (`parsed`) and the operation (`filter:`) that has to be applied. As in Spark, transformations are either *narrow*, i.e., each partition of the child DDD depends at most on one partition of the parent DDD, or *wide*, i.e., any partition of the child RDD depends on more than one parent's partition.

Actions are those operations that alter both content and structure of a DDD, returning a conventional data structure. Actions are eager, i.e., they will be executed immediately when they are applied. Common actions include `sum:`, `count:`, Examples of such actions are the `getCollection`, `take:`, and `takeSample:`.

When an action is applied, the computation of all transformations is triggered. If a wide transformation is present, the runtime first pipelines all narrow transformations before it and finally executes the wide transformation. Similarly, the execution of all narrow transformations is pipelined to be executed before the action, which then returns results in a conventional data structure to the driver. In the example of Listing 3.3, line 6 sends a `getCollection:` message that first triggers the transformations from line 1 to line 4, then executes a `groupByKey` in the context of the `reduceByKey:` method. The `reduceByKey:` transformation is then executed on its result and the action (i.e. `getCollection:`) returns the result in a collection to the driver.

3.5.3 Persistence

Following the original Spark model, intermediate data is not persisted between transformations and developers can explicitly persist intermediate data after a particular transformation. This is useful if that particular DDD is reused later in the computation to avoid re-computing it. In Spa, this is done by sending the `execute` message and the data is persisted always in the memory of the workers. This operation is similar to Spark's `persist`, except that in Spa it is an eager action. Furthermore, the developer cannot specify a persistence level (i.e., persisting in memory, on disk, or both) as in Spark.

Not persisting automatically intermediate results avoids systematical heavy file writes during the execution to store temporary data, common in Map/Reduce frameworks as already discussed in Section 2.3. It does, however, impact the fault-tolerance properties: in Spark even if data is not persisted, fault tolerance is given by the fact that it is always possible to recalculate a certain partition of an RDD/DDD by using the lineage of the distributed data structure, so by re-executing the set of transformations and actions that generated it, as discussed in Section 2.3.

3.6 Deploying Port and Spa

As described in Section 3.2, Port and Spa run on different processes acting as master, worker, and driver. The two frameworks can be deployed both locally or on a cluster. This is relevant for the validation of our debugging prototypes since we use this mode when testing our debugger. Deploying happens through an additional component, the *deployer*, that spawns and monitors all the processes.

Deploying locally. When deploying locally, the deployer takes a snapshot of the current image and starts the different processes using this snapshot. This ensures that the framework processes execute separately, and on a different image than the one used to spawn them, albeit having its same code base. The Master Server (cf. Figure 3.1) always runs at the same port, so the workers, once they are started, contact the master at that address to communicate their presence. The deployer monitors the execution of the different processes and can notify the master in case one of the workers crashes. In that case, upon confirmation of the master,

the deployer redeploys a new worker. When running locally, the process that the developer uses to spawn the master and workers (i.e., a Pharo execution) is also used to drive the execution, i.e., as a *driver* in Spark.

Deploying on a cluster. When deploying on a cluster, i.e., a set of different machines connected through a local network, the deployer communicates with an instance of Apache Yarn running on the cluster. Using Yarn allows us to abstract on the properties of the system, e.g., available memory, available CPU, general availability of a node, etc. Yarn is commonly used to deploy frameworks such as Map/Reduce and Spark on a cluster, especially when the size of the system increases. In our case, Yarn handles the configuration and execution of the cluster, providing deploying and node management.

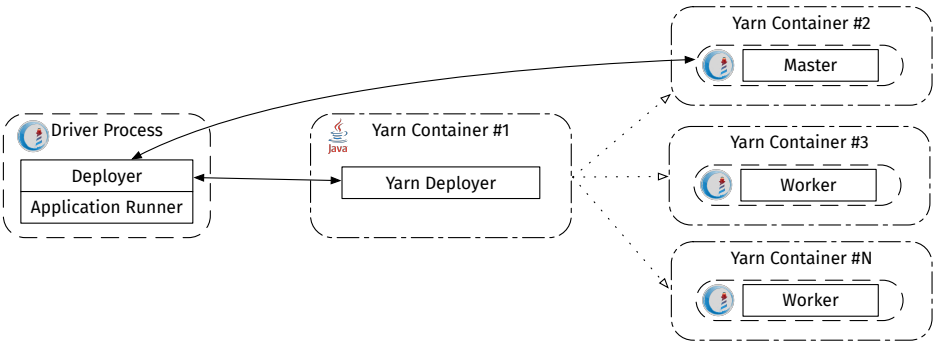


Figure 3.2: Overview our runtime when deployed on a cluster using Yarn.

Figure 3.2 shows the deployment architecture of Yarn. On the left, you can see the Driver Process, i.e., a Pharo process that runs the deployer together with the application runner. It communicates with the *Yarn deployer*, which is run by Yarn in a container. The Yarn deployer is a Java application responsible for requesting the allocation of new containers and for monitoring their state. Upon requesting to allocate a container through the deployer, the application master interfaces with the Yarn framework to know where (i.e., on which node) the container was allocated. All the resource constraints (e.g., requested memory vs available memory) are handled by Yarn. If a container cannot be deployed at a particular moment, it will be deployed when the resource constraints are satisfied.

The deployer regularly queries the Yarn deployer to know the state of the containers. In case the state has changed, e.g., a worker was added or removed, the deployer is responsible for notifying the change to the master. Deploying does not block the driver and the master/workers that are already deployed. If a particular container is never spawned by Yarn, the system can work with the ones that are already available.

3.6.1 Deploying and Running Port/Spa Programs

The main way to deploy Port/Spa is by using the dedicated front-end in Pharo. This is a GUI built in Pharo to deploy and monitor the Port/Spa runtime, as well as for running the applications.

Through this GUI, the developer can select which deployment mode (i.e., local or cluster) to use. Workers are added and removed dynamically through buttons in the UI, or through API calls on this instance of the framework. The tool also offers a dedicated playground, i.e., an environment in which developers can write and execute code through a variable that represents the entry point to the framework already bound. When executing locally (and optionally also when executing in cluster mode) the instance of the framework instantiated in the playground acts as the application runner of the application, i.e., it executes the sequential part of the code, and it spawns the parallelized computation upon actions.

In Port, an application is executed by calling the `runApplication: parameters:` method, making sure to pass a `PortApplication` and its parameters as parameters. In Spa, the developers execute a full Spa application through the same API call as for Port, or they can dynamically execute their application in the provided playground by calling methods in the API of Spa. For example, by calling `distribute:` or `textFile:` Spa will respectively distribute a collection passed as a parameter or the contents of a file, which will return a DDD that can be used in the playground.

3.7 Conclusion

In this chapter, we presented Port and Spa, two frameworks that implement the support for Map/Reduce and Spark-like programming and execution models in Pharo, respectively. They allow us to write and exe-

cute Big Data applications on a cluster and conduct research experiments on the debugging support.

Although Port and Spa do not present all the optimizations available in mainstream systems such as Hadoop Map/Reduce and Apache Spark, they are representative of such frameworks and provide a realistic environment to investigate and develop our debugging support. In Chapter 8, we validate the scalability of Port using a realistic workload and of Spa throughout the benchmarks of our debugging approach.

Chapter 4

A Call-Stack Instrumentation Layer for the Debugging of Framework Code

Libraries and frameworks are instruments that define a set of functions to accomplish common tasks. They are meant to be reused, so developers don't need to implement the same behaviour multiple times and can focus on the application they want to implement. For instance, they solve problems in a wide range of domains: from unit testing (e.g., the xUnit family of frameworks) to scalable parallel execution (e.g., Apache Spark and Hadoop Map/Reduce), passing through concurrency (e.g., Akka actors) or persistence (e.g., Hibernate). In this chapter, we explore Sarto, a call-stack instrumentation layer to enhance the debugging of framework code by the use of several operations to tailor the call-stack to the needs of the developer. Sarto represents one of the foundations of our debugging solution for Big Data applications that we use to implement some of the internals of our debugging solution in Chapters 6 and 7.

Even though stack traces are key in understanding the execution of a program, finding the root cause of bugs remains difficult because raw call-stacks are difficult to read: application frames and framework frames are interleaved in the stack, although most developers are often (if not only) concerned about their own code. Moreover, certain important information

may be absent from a call-stack because it may be contained in methods that already returned or that were executed in another thread. Moreover, when debugging parallel or distributed applications, important debugging information is scattered in different call-stacks of different processes that users need to manually relate.

Classical online debuggers such as IntelliJ and Eclipse offer solutions to filter stack-frames, but they require either developer interaction or rely on heuristics defined within the debugger. Other solutions focus on domain-specific debugging of particular frameworks, through specific views and operations, as it's the case for actors [GNV+11, SCM09] or Big Data frameworks [DZSS13, JYB11, GIY+16].

Our work revisits the concept of a call-stack to enable a framework-aware debugging experience. Application developers debug a call-stack that is previously *tailored* to the frameworks they are using and can dive into the original call-stack in case they are interested in the framework's code. To perform stack tailoring we propose Sarto, a call-stack instrumentation layer that framework developers use to hide, show or relate debugging information within the context of a framework execution. More concretely, Sarto proposes a set of six call-stack operations to (1) cut and (2) concatenate call-stacks, (3) insert framework-specific stack frames, and when given more than a call-stack, (4) compare two call-stacks to check if they are *similar*, (5) calculate a delta (similar to a diff between two call-stacks), and (6) apply such delta to other similar call-stacks.

Before delving into the concepts of Sarto and its stack-tailoring operations, we introduce four debugging use cases related to different frameworks. Then, we show in practice how we used Sarto in the different frameworks and provide some insights on the implementation details. We validate Sarto by analyzing our experience in using it in the context of the different frameworks, and by running performance benchmarks to assess that Sarto does not introduce noticeable overhead during normal debugging. Finally, we provide notes to the related work to describe in more detail other approaches for debugging frameworks.

4.1 Challenges of Debugging Frameworks

This section details the issues when debugging framework code employing four use cases in four different domains: web servers, unit testing, promises, and parallel executions.

4.1.1 Case 1: Debugging Web Servers

Web servers are frameworks that wait for network HTTP requests and dispatch the handling of such requests to the corresponding application code. If an error occurs in the application, the call-stack presents application frames, i.e., the method calls that produce the answer to an HTTP request, followed by several frames representing internal calls in the HTTP server. Figure 4.1 shows an example of such a call-stack when debugging a failing execution in the context of the Zinc HTTP framework of Pharo. The top two frames of the stack are application frames while the rest, under the dashed red line, are framework frames. When stepping in this execution, the developer easily ends up in framework-related frames.

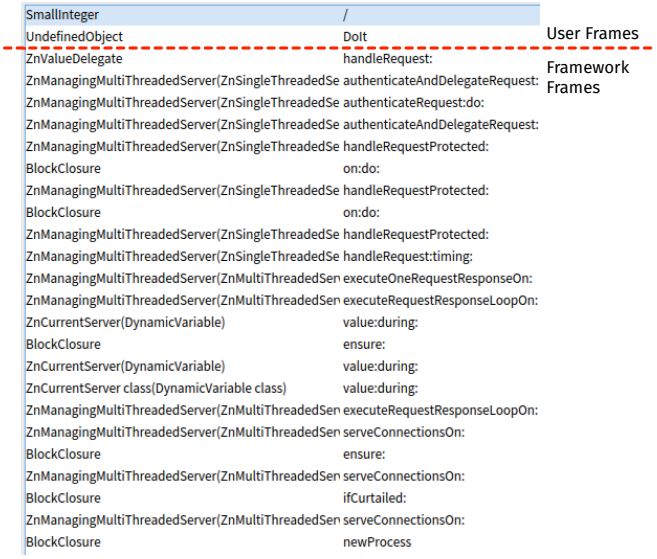


Figure 4.1: The call-stack when debugging a failing HTTP server in the Pharo debugger.

We say in this case that the debugger offers **irrelevant information** to the developer. The information about framework frames is not needed to understand the failing user code and just adds noise to the debugging experience. As mentioned in the introduction to this chapter, debuggers in mainstream IDEs (e.g., Eclipse's debugger) offer filtering operations to hide such framework frames in the stack. Such filters are either delegated to the application developer, who may not have enough knowledge to do it correctly, or are based on several predefined characteristics such as type and location, i.e., whether the class of the called method is in one of the project's dependencies, or it is part of a given list of packages. For example in IntelliJ, when the developer presses the filter button all frames that the IDE knows to be from known libraries are filtered out. This is based on heuristics to recognize library code and may filter out important frames, as it is the framework developer who has the best knowledge about the framework internals, but they have no control over how the stack is shaped once filtered by the IDE. Thus, the user of the debugger is presented either with a complete stack or with a filtered stack in which important methods are not present anymore.

4.1.2 Case 2: Debugging Unit Tests

Unit testing frameworks are tools available for most programming languages to support the implementation, execution, and reporting of tests results. A unit test generally focuses on testing the behaviour of one functionality of the system, i.e., a *unit*, using assertions that check that certain properties hold, e.g., checking that a function returns a specific value. In case of a test failure, i.e., in case the test raised an error or produced an unexpected result, developers normally proceed to debug the test to fix the application's behaviour.

When a developer debugs a test execution, the debugger halts at the point in which an exception is thrown or an assertion fails. In this case, the bottom of the stack has application code calling the testing framework, followed by frames of the testing framework, and finally the frame representing the failing test assertion at the top of the stack. While the framework frames are usually not interesting to the application developer, there is one particular method that could provide crucial information to find the root cause of a failing test: the `setup` method, i.e., a method that is called before any test execution to setup the necessary resources. This

method activation is usually either hidden in between the framework stack frames or is not even present in the stack anymore. However, the `setup` method is important for the developer to reason about the execution of the unit test as it contains information on how the test fixture was initialized. For example, it shows which variables were initialized and which methods were called. This information may help the developer to identify variables that were not correctly initialized or not initialized at all.

In this case, the debugger **misses information**: the stack frames with relevant information are hidden or absent in the stack, e.g., the `setup` and `teardown` methods.

4.1.3 Case 3: Debugging Promise Executions

Promises are recurrent programming abstractions in concurrent languages to reconcile asynchronous communication with return values [BGL98]. Many mainstream languages provide libraries with support for promises, e.g., Scala [Lig] and JavaScript [Sye14]. A promise represents the result of an asynchronous operation that may execute concurrently. When such execution succeeds, the promise is resolved with a value; if it fails, the promise is said to be ruined with an exception. Developers add callbacks to handle both successful and failing resolutions.

Consider a promise created to gather the results of an asynchronous execution as in Listing 4.1. In Pharo, a developer creates a promise from a closure. The promise is executed on a different thread and is resolved later, possibly with an exception. In the example, the promise divides by the argument `n`, and a callback is added to intercept the ruining of the promise and open a debugger on the thrown exception.

```
PromiseRunner>>promiseDivision: n
  promise := [1 / n] promise.
  promise onFailure: [:err | err debug].

PromiseRunner new promiseDivision: 0.
```

Listing 4.1: A failing promise.

Let us consider that a promise is created capturing the value `n = 0`. When debugging this failed promise, developers find a call-stack that includes only the promise execution, and not the frames that lead to the creation of the promise. This happens because the creation and execution

of the promised code happen in different threads, so it is difficult for developers to trace back the origin of the value zero. As such, two call-stacks are involved, one for the creation and one for the execution of the promised code. Moreover, the stack frame that created the promise is not available anymore because it returned right after creating the promise. Figure 4.2 shows the two threads: the left one represents the promise execution, and the right one the promise creation.

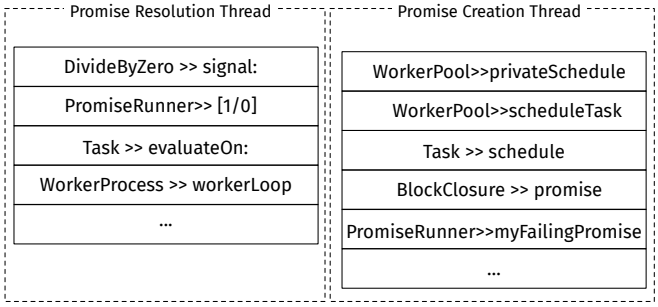


Figure 4.2: Representation of the two treads involving the creation and execution of a promise.

In this case, the debugger is again missing information that is **dispersed** over different call-stack(s): part of the application code potentially related to the error is in another thread or has already finished executing. This observation leads to different domain-specific debugging techniques whose goal is to reconstruct causal relations in asynchronous communication to offer an *asynchronous track trace* [SCM09, LCN17, Dra13]. In this work, we explore a generalization of those techniques to a solution that relates information present on different stacks.

4.1.4 Case 4: Debugging Concurrent Web Servers

Consider again an error happening multiple times when resolving certain HTTP requests in an HTTP server. HTTP servers are often multi-threaded and spawn a thread for handling new requests. When an error occurs while handling a request, the server will answer with a *500 Internal Server Error* as a response, staying available to resolve the next request. When an internal server error happens multiple times, debugging those problems requires an understanding of how those errors are related, as their root cause may be related even though they happened in different

threads. Even if those call-stacks are identical or even similar, the debugger shows no relation between them. It is up to the developer to verify whether they are related, debug them singularly, or abstract information from all of the different exceptions.

In this case, the information about the bug is again **dispersed** over different similar call-stacks. Issues related to debugging parallel executions also arise in the context of debugging Big Data frameworks. In a Big Data framework, operations on a certain data set are executed in parallel on a cluster of machines, over different portions of the data set. Parallel executions (e.g., for Map/Reduce frameworks) then generate multiple call-stacks.

4.1.5 Summary

Through the four case studies presented in this section, we identified three common problems when dealing with the debugging of framework executions. Particularly, some of the information available to developers is (i) **irrelevant**, that is the case when many framework frames bloat the call-stack hiding the user frames that actually present the incorrect behaviour; (ii) **missing**, when crucial information such as the setup method of a unit test is not present in the stack; (iii) **dispersed** when information about a parallel or asynchronous execution is dispersed in different call-stacks.

4.2 Sarto: a Call-Stack Instrumentation Layer for Framework-Aware Debugging

To enable a *framework-aware debugging* experience, we propose a call-stack instrumentation layer that tailors call-stacks based on framework information. Framework developers hook into this instrumentation layer to hide, show or relate debugging information within the context of a framework execution. More concretely, we introduce a debugging instrumentation layer with a set of six on-stack operations. These operations transform the call-stack before it is given to the debugger that we call *Sarto*. As a result, the debugger exposes framework-specific information when debugging a particular framework without having to adapt the underlying language or runtime.

In this section, we first present a general overview of the six stack operations and then describe how the different operations are used to tailor call-stacks for the debugging of the four aforementioned cases: HTTP server, unit testing, execution of promises, and parallel executions. In particular, we describe how they are used to *hide useless information*, *display useful information*, and *relate dispersed debugging information*.

4.2.1 Terminology

Before delving into the specifics of each operation, we provide our definition of the necessary terminology that we use to describe our work. A *call-stack* is a linked list of stack frames. A *stack frame* represents the activation of a method or function: it holds a reference to the method and the current program counter. Each stack frame references its caller (i.e., the stack frame that generated it), its arguments, the receiver (i.e., `self` in Smalltalk or `this` in Java), and the local variables.

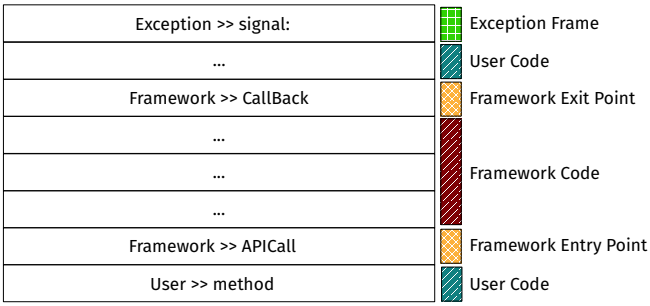


Figure 4.3: Representation of a call-stack, marking each frame as user code, framework entry/exit point or framework code. The most recent frame is at the top.

Figure 4.3 illustrates a call-stack where user code invokes framework code, and in turn framework code invokes user code in a callback. Starting from the bottom, a user frame calls the *framework entry point*, i.e., a framework stack-frame that was invoked by application code. This is followed by a series of framework-related frames and by a *framework exit point*, i.e., a framework stack-frame that invokes application code. Finally, a series of user-related frames calls the exception frame, i.e., the frame representing the signalization of an exception. In this example, the call to the framework API generated by the user method is the framework

entry point, and the method that performs the callback to user code is the framework exit point.

4.2.2 The Stack Operations

Sarto presents six operations to tailor the debugging information present in a call-stack. Table 4.1 briefly summarizes the proposed operations.

Table 4.1: Overview of the stack operations.

Operation	Description
Stack cutting	Produces a new call-stack by filtering out some activation frames.
Crafting a stack frame	Produces a new call-stack by inserting a custom stack frame in between two other frames.
Concatenating stacks	Produces a new call-stack from two call-stacks to simulate a sequential execution.
Stack Comparison	Compares two call-stacks to determine if they represent two similar exceptions.
Delta Stack Calculation	Produces a delta stack containing only the differences between two similar stacks.
Delta Stack Application	Produces a call-stack from merging a call-stack with a compatible delta stack.

Stack cutting. The *stack cutting* operation takes a call-stack and produces a new call-stack by removing all stack frames in between framework exit and entry points, thus hiding irrelevant framework stack frames. In our approach, framework entry and exit points are explicitly marked by framework developers in framework code with method annotations (cf. Section 4.3).

For example, Figure 4.4 shows the call-stack of a failing HTTP request in a web server that has been tailored by Sarto. In the figure, framework frames at the bottom of the stack are displayed in gray to illustrate the fact that they have been cut out, while the framework exit point and the user frames are kept in the debugged stack. With this operation, application code is isolated from framework code in the debugged execution. Letting the developers focus on debugging their code, without unintentionally

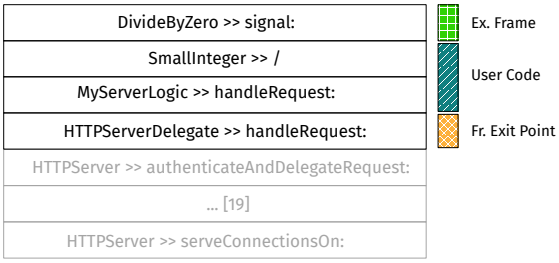


Figure 4.4: Representation of the stack upon an exception in the HTTP Server.

stepping into framework code. It is the choice of the framework developer to mark in the framework code which frames are framework entry and exit points, so that stuck cutting is applied at the right frames.

Crafting a stack frame. The *crafting a stack frame* operation produces a new call-stack that contains a custom stack frame inserted in between two other stack frames, thus introducing information that was otherwise missing. As explained in Section 4.1.2, this is the case of methods that already returned or were called in a different thread.

Framework developers define a method that will either substitute or go under the framework exit point. In this way, application developers are offered call-stacks augmented with relevant information for debugging.

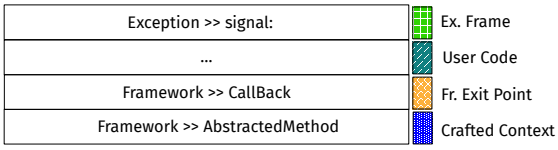


Figure 4.5: Representation of a call-stack, with a crafted context inserted before the framework exit point.

Figure 4.5 displays the stack of Figure 4.3, where the exit point was substituted with a custom frame. In this case, to avoid incompatibility between a frame and its caller, we employ stack cutting to remove the stack frames below the exit point.

Concatenating stacks. The *stack concatenation* operation produces a call-stack by concatenating two different call-stacks, thus reconciling two threads of execution and giving the user the illusion of a single sequential execution. This is the case of, for example, the call-stack of the remote resolution of a promise (cf. Section 4.1.3), or more in general, the call-stack of a user code callback within the remote execution of a framework.

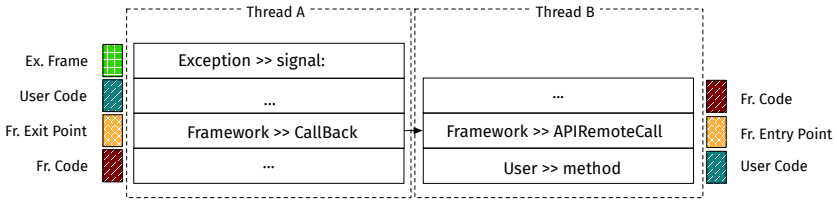


Figure 4.6: Representation of two call-stacks during a failing remote execution of a framework call.

Consider Figure 4.6, displaying two call-stacks related to two threads: thread B (on the right of the figure) that presents the call-stack including the original user call to the framework entry point, and thread A (on the left in the figure) in which the framework exit point calls back user code that then generates an exception. The *stack concatenating* operation links the two call-stacks using the entry and exit point, present and marked in both the call-stacks. Figure 4.7 shows the resulting call-stack.

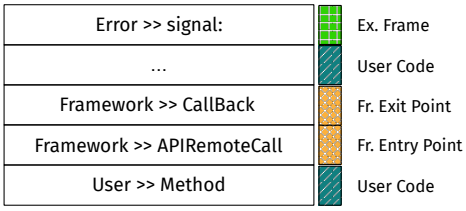


Figure 4.7: The results of concatenating two call-stacks from different threads.

In this way, when debugging an exception raised in thread A, developers are presented with a unique call-stack that includes both the exception raised by the framework exit point and the user code leading to the framework entry point. This implicitly hides the framework frames that e.g., take care of the network communication.

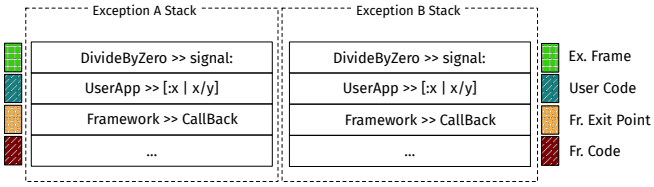


Figure 4.8: Representation of two similar call-stacks.

Stack comparison. The *stack comparison* operation analyzes two call-stacks and determines whether they are *similar*. Two call-stacks are considered *similar* when they are structurally the same, as in the case of Figure 4.8. More precisely: two call-stacks are similar when, by traversing their frames in pairs, they present the same sequence of method calls and in each pair of stack frames the program counter of the method is the same. Furthermore, each pair of frames should have the same type of receiver.

This is useful in combination with the remaining two operations (*delta stack calculation* and *delta stack application*), explained in following paragraphs. Call-stack comparison identifies similar call-stacks among those that are dispersed across a parallel or distributed execution, or across two or more different executions. When comparing the call-stack of two exceptions, the stack comparison first checks whether the two exceptions have the same type before comparing their two call-stacks. This is because the call-stack of two different exceptions will differ at least in one of the frames, i.e., the frame that throws the exception, making the two call-stacks not similar.

Delta stack calculation. The *delta stack calculation* operation takes two similar call-stacks and produces a shrunk call-stack that contains only the values that differ between the two call-stacks.

Given two similar call-stacks, thus under the assumption that the two call-stack are structurally the same, frames are traversed in pair and all their associated values are compared (i.e., receiver, arguments, and temporary variables). When they differ, the different values are stored in the resulting *delta stack*, i.e., a representation of the call-stack that includes frame by frame not a full stack frame, but just the computed delta among its variables.

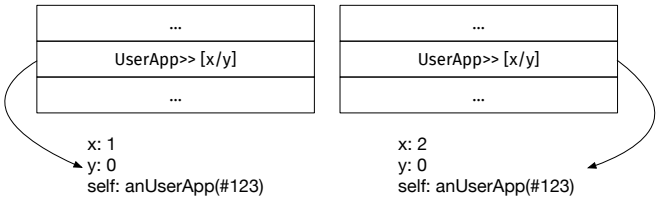


Figure 4.9: Two stack frames and their variables in the calculation of the delta stack.

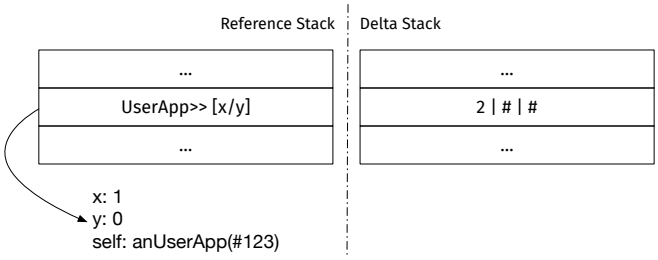


Figure 4.10: The calculated delta stack for this frame.

For example, consider Figure 4.9 showing a portion of the call-stack presented in Figure 4.8. In the example, the two variables `y` are the same in both stacks, as is the case for the receiver of the method. The two `x` variables, however, differ in the two call-stacks. Figure 4.10 shows the resulting delta stack on the right, compared to the reference call-stack. Instead of the full frame, a frame of a delta stack includes only the variables that do differ, in the same order as they are internally represented in the original stack frame. For the others, a placeholder indicated as # in the figure is included in the delta stack. Hence, in this delta stack the variable x, which was 1 in one stack and 2 in the other, is included in the delta. The other two variables (y and self) were not differing, so they are substituted with a placeholder marked as # in the figure.

Since only the values that differ are kept, and not the original one, the delta calculation is not a commutative operation.

Delta stack application. The *apply delta stack* operation takes one full call-stack and a delta stack associated with it and produces a new call-stack representing the application of the delta to the full call-stack. The delta stack must have been created from a similar full call-stack or a

copy of it. This enables the reconstruction of multiple call-stacks from a single full call-stack and several delta stacks.

In a scenario with many similar remote call-stacks, only the first must be stored entirely. All subsequent similar call-stacks are stored as delta stacks instead. This is the case for consequent executions for example of an HTTP framework, but also for Big Data applications, where several workers can fail with the same exception. This operation will be used and further discussed in Chapter 7.

4.3 Sarto in Practice

In the previous section, we defined the 6 call-stack instrumentation operations provided by Sarto. By using these operations, framework developers can provide an improved experience to application developers when debugging applications with library framework code. In this section, we elaborate on how these operations are used to enable framework-aware debugging tools.

We prototyped Sarto's set of operations as a library for Pharo Smalltalk. Sarto's operations apply after a runtime call-stack reification step, but before opening a debugger: the operations are applied to the reified call-stack, producing an instrumented call-stack that is fed to the debugger. In other runtimes, instrumentations could be applied in the debugging instrumentation layer.

Sarto supports the different operations through method annotations and an API of calls on a Sarto static instance. In Table 4.2 we detail the main methods of Sarto's API in relation to the operation.

In the next section, we describe the practical application of these operations in the context of the four use cases.

4.3.1 Enabling Sarto in Framework Code

Since Sarto manipulates the call-stack, the call-stack needs to be reified and available to the framework developer, as is the case in Smalltalk. To include support for Sarto, framework developers need to insert instrumentation calls in the points in which they want to apply Sarto's operations. In these points, the developers use Sarto to extract the reified call-stack and store it in a variable for further manipulation. In Smalltalk, the

Table 4.2: Overview of Sarto's API.

Operation	Description
Framework entry/exit point definition	`<frameworkEntryPoint: #FrameworkName>` `<frameworkExitPoint: #FrameworkName>`
Stack cutting	`Sarto cutAndDebug:`
Crafting a stack frame	`Sarto newSubstituteMethod methodAt:`
Concatenating stacks	`Sarto combineStackOfRemote:withLocal:`
Stack Comparison	`Sarto compareStackException:with:`
Delta Stack Calculation	`Sarto calculateDeltaStackBetween:and:`
Delta Stack Application	`Sarto applyDelta:toException:`

stack is reified on-demand, i.e., explicitly by the developer through a call to the runtime, or at the moment of an exception if an exception handler is present. The framework developer explicitly extracts the call-stack through a call to `Sarto captureStack`[1]. Alternatively, if instrumentation happens in the context of an exception handler, framework developers use the error reification of Smalltalk, which includes the call-stack.

Furthermore, the runtime also needs to support method annotations to let developers annotate methods as framework entry and exit points.

The framework developer defines framework entry and exit points by adding method annotations[2]. The following snippet of code illustrates how to add such annotation to identify framework exit points. Please note that the procedure to add a framework entry point is equivalent.

```
HTTPServerDelegate>>handleRequest: aRequest
  <frameworkExitPoint: #HTTP>
  ...
```

4.3.2 Cutting the Call-stack Before Debugging

The framework developers use stack cutting to remove framework frames from the execution that is about to be debugged. For instance, in the

[1]Using `Sarto captureStack` is syntactic sugar in Pharo for copying the value of `thisContext`, a pseudo-variable that reifies the call-stack.

[2]In Smalltalk, a static method annotation is denoted between lower and greater signs at the beginning of a method definiton

case of the HTTP framework, developers can add an exception handler to the code that manages incoming requests, and thus capture errors and manipulate the stack before opening the debugger.

The listing below shows the code to enable this operation in an exception handler, using the error and call-stack reification provided by Smalltalk already in the exception handler.

```
[...] on: Error
      do: [:err| Sarto cutAndDebug: err].
```

This operation performs stack cutting on the defined framework entry and exit points. To perform cutting, this operation navigates the call-stack from the top to detect a framework exit point, and then further navigates it down to detect the framework entry point. If no entry point is found, the operation will cut the call-stack at the exit point. If it is found, the stack is cut between the two entry and exit points.

4.3.3 Crafting a Stack Frame

To illustrate the usage of *crafting a stack frame*, we extend the unit test framework scenario to augment it with `setup` and `teardown` methods. The framework inserts a stack frame with a method containing the code of the setup method otherwise not present in the call-stack, and then a call to the actual test. This method shows useful information to the developer while hiding all of the internal framework calls. Sarto cuts the call-stack at the framework exit point and inserts the crafted frame with the custom setup method below it. The developer sees the setup next to their test and can re-execute the setup and the single test without having to go through any framework code.

The framework developer adds a custom stack frame by specifying a mapping between the frame to replace and the method that will replace it in the call-stack. The replacement method is either an existing method or a method constructed on the fly, since Pharo allows to compile methods reflectively at runtime.

The following code snippets show how to replace the frame of the `performTest` method with an existent method (`setUp`) or with one crafted by hand (`contextForDebuggingSetup`). The framework developer optionally specifies the program counter to keep the stack consistent.

```
TestCase>>initialize
```

```
substituteContexts
  at: #performTest
  put: (Sarto newSubstituteMethod methodAt: #setUp})
```

```
TestCase>>initialize
  substituteContexts at: #performTest
  put: (Sarto newSubstituteMethod compiledMethodAt:
    #contextForDebuggingSetup ; pcAt: #performTest) }).
```

An optional flag enables developers to keep or hide the given framework entry and exit points in the call-stack, as shown below. If such flag is set to false, the crafted stack frame will substitute an entry/exit point.

```
TestCase>>initialize
  substituteContexts
    at: #performTest
    put: (Sarto newSubstituteMethod compiledMethodAt:
     #contextForDebuggingSetup ; pcAt: #performTest ; keepMethod: false) })
     .
```

4.3.4 Concatenating Stacks

Consider again the case of debugging a promise execution as explained in Section 4.1.3. We use the *stack concatenation* operation to take both call-stacks and link them at the framework entry and exit points, in this case the promise invocation and the callback to the promise resolution, respectively. Stack concatenation requires framework developers to (i) identify framework entry and exit points, and (ii) explicitly capture the call-stack at an entry point using an eager stack capture. The result is a single *asynchronous* stack, exposing both the local and remote execution contexts, depicted in Figure 4.11.

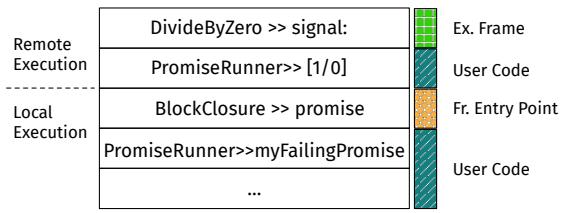


Figure 4.11: Representation of the instrumented stack of a failing promise, tailored for debugging.

Our approach takes inspiration from Leske et al. [LCN17], who proposed a debugging model for promises in which the call-stack that leads to the generation of the promise is linked to the one of the promise execution through proxies. However, our approach potentially applies not only to promises but also to other asynchronous execution models such as the actor model.

In the case of concurrent promises, as shown in Listing 4.2 we define the framework entry point in the method `promise`. We first eagerly capture the call-stack at the entry point (line 3) and then add a callback to the promise (line 5). In case of a failure, the callback will lazily capture the call-stack at that point and concatenate both call-stacks (line 7). We then forward this stack to the debugger (line 8).

```
1 BlockClosure>>promise
2   <frameworkEntryPoint: #Promises>
3   stack := Sarto captureStack.
4   promise := Promise from: self.
5   promise onFailureDo: [:err |
6     combinedStack := Sarto combineStackOfRemote: err withLocal: stack.
7     Sarto cutAndDebugError: err withStack: combinedStack)].
```

Listing 4.2: Capturing and debugging the stacks of a promise.

4.3.5 Debugging with Delta Stacks

To illustrate the usage of the delta stack operations (*stack comparison*, *delta stack calculation*, and *delta stack application*), we show the example of debugging several failed HTTP requests. In fact, when a server fails in handling several HTTP requests, chances are that similar requests failed several times, hence representing different instances of the same problem or bug. It is usually the developer's responsibility to tell if two call-stacks are similar or not and to debug them singularly.

With Sarto, we extended the debugger when debugging HTTP frameworks to detect similar call-stacks with stack comparison and create delta-stacks in those cases on the server-side. When debugging, only one exception is presented for different similar exceptions, grouping the exceptions by their call-stack shape. A representation of such a debugger UI is shown in Figure 4.12.

By selecting one of the four x values, the developer debugs that particular call-stack. This triggers the application of a delta stack to the

original call-stack and presents the user with a call-stack that is a copy of the original call-stack generating the problem.

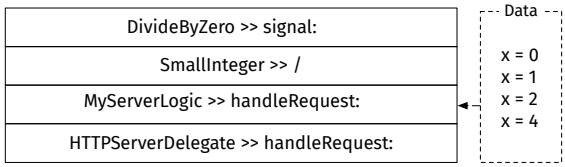


Figure 4.12: Representation of a debugger showing an instrumented call-stack, with the different possible variables found in the delta stacks.

To use delta stacks, the framework developer relies on the call-stack reification upon handling an error to store it. If a captured call-stack is similar to an already stored call-stack, they add the instrumentation code to calculate the delta stack and store that one instead. To illustrate this, we show how delta stacks are used in the case of multiple consecutive errors happening in a web server.

```
1 MyRequestHandler>>handleRequest: req
2   [ ...] on: Error do: [:error |
3   similarError:= errors at: error exceptionID ifFound:  [:similar |
4   compare := Sarto compareStackOfException: error with: similar.
5   compare ifTrue: [delta := errors at: error exceptionID put: (Sarto
      calculateDeltaStackBetween: error and: similar.
6     deltas at: error exceptionID put: delta.] ]
7   ifNotFound: [errors at: error exceptionID put: error].
```

Listing 4.3: Handling errors using delta stacks in an HTTP server.

Listing 4.3 shows the code of the request handler of our web server including support for delta stacks. When an error happens, the error handler captures the call-stack and checks if there is already an entry for that call-stack (line 3). If there is, then `#compareStackOfException:` is called to check whether the exceptions are similar (line 4). If they are, a delta stack is calculated and stored in a different data structure (line 5). On the client-side, the framework developer retrieves the exceptions and uses `Sarto>>#applyDelta: delta toException: exception` to do the inverse.

4.4 Validation

To validate our solution, we conducted two sorts of experiments. We first show that Sarto applies to various frameworks, each with different requirements. Second, we show that our solution is practical and efficient by conducting performance benchmarks to show that Sarto does not introduce significant overhead to the execution or debugging code of the three use cases.

4.4.1 Experiences in using Sarto

To validate that our approach works for a variety of different frameworks, extending the debugging support for the following Smalltalk frameworks: Zinc, an HTTP server framework; SUnit, the classic unit testing framework of Smalltalk; TaskIt[3], a framework for task scheduling that we adapted for remote execution.

Depending on the debugging needs of each framework, a different combination of operations from Sarto was used. Table 4.3 summarizes which operations are applied for which framework. Three operations are applied, in different combinations, to all of the frameworks: stack cutting, crafting a stack frame, and concatenating stacks. The *Δ (delta) stack operations* row groups the use of the remaining three operations, i.e., stack comparison, delta stack calculation, and delta stack application. This is because they are used in combination, and only when debugging multiple executions that happen either in parallel or across long computations. Finally, we applied *ΔStack operations* to debugging the HTTP framework.

Table 4.3: Usage of basic stack operations on the four frameworks.

Operation	Zinc	SUnit	TaskIt
Stack cutting	✓	✓	✓
Crafting a stack frame		✓	
Concatenating stacks			✓
ΔStack operations	✓		

For this validation, we analyzed a series of exceptions found while using the three different libraries (Zinc, SUnit, and Taskit). By analyz-

[3] `https://github.com/sbragagnolo/taskit`

ing those exceptions, we derived the framework entry and exit points as those methods that actually performed a callback to user code. From our experience, finding the framework entry/exit point was not particularly time-consuming, since it happened gradually while debugging application code using the aforementioned frameworks. Thus, stack-cutting was relatively easy to put in use in the case of Zinc, but in the case of the SUnit framework cutting also required crafting a new stack frame so that the instrumented call-stack would be consistent, and this proved to be a more complex task. For example, we had to analyze the code and test multiple times the way we crafted the setup method into the instrumented call-stack, so it wouldn't cause incompatibilities on the stack. The interface proposed in Section 4.3 allows developers to define a crafted stack frame by specifying which method should be used in the stack frame, and giving control on what context should be crafted, e.g., whether it should substitute the exit point.

In Sarto, concatenating stacks is related mostly to the correct definition of framework entry/exit points, and it was relatively easy to apply it to TaskIt as described in Section 4.3. Finally, the only requirement to using the delta stack operations (comparison, calculation, application) was to handle the exceptions by capturing the call-stack, grouping similar exceptions together (for example in a dictionary), and managing the calculated delta stacks.

4.4.2 Performance Benchmarks

In this section, we evaluate the performance overhead of exception handling when using our approach. Particularly, we run two experiments to assess Sarto's impact on the time of exception handling for debugging in the different frameworks. First, we analyze this for the Zinc and SUnit frameworks. Second, we analyze the same property for the TaskIt framework while increasing the size of the data referenced by the stack, since this may impact stack concatenation. The performance of delta stack operations will instead be assessed as part of the validation to the overall debugging approach in Chapter 8.

Setup

We perform our benchmarks using Pharo 8.0, which already packages Zinc, SUnit, and TaskIt, on a MacBook Pro 2017 with an Intel(R) Core(TM) i7-7567U CPU @ 3.50GHz and 16 GB of RAM DDR3.

In both experiments, we iterate each measurement 25 times and report averages and standard error. To measure exactly when a debugger is opened, we extended the debugger to store a timestamp just before it is opened. We compare that timestamp to the one retrieved before starting the execution of the framework call.

4.4.2.1 Overhead on Exception Handling

To verify the performance impact of our solution we compare the time to handle an exception with our approach and without it in the Zinc and SUnit frameworks. For Zinc, we setup an HTTP server (with and without Sarto's stack cutting operation), and we measure the time between performing an HTTP post request through a client, and the moment in which the debugger is opened. Similarly, for SUnit we measure the time between running the test and the moment in which the debugger is opened.

Table 4.4: Time to open a debugger with or without Sarto. Times in milliseconds.

Framework	*Sarto*	Err_S	*Default*	Err_D	Diff.
HTTP Framework	71.5	3.1	103.5	7.74	-30%
Unit Testing	28.32	1.5	75.08	4.2	-62%

Table 4.4 shows the results of our benchmark: the Sarto column shows the average execution when managing the exception with Sarto, the default column represents the average execution time when managing the exception with the default handler. The error columns represent the standard error of the mean. Both managing or not managing an exception show average delays in the order of 103.5 milliseconds at maximum, which we consider small enough for an interactive debugger. Handling an exception with our instrumentation layer resulted in about 50 milliseconds faster than the unmanaged one. Although this looks like a performance improvement, we believe the result is related to traversing the stack fewer

times. From those results, we conclude that our instrumentation layer does not introduce a significant impact on the execution.

We did not include the TaskIt framework in this analysis since it will be further discussed in the next section.

4.4.2.2 Scalability when Increasing Stack Size

This benchmark measures the influence of the stack size on our approach, particularly for the TaskIt framework. This use case requires a copy of the call-stack, hence it gives a measure of the overall overhead of our approach. This benchmark is a variation of the previous one, thus we measure the time since a promise is created until a debugger is opened on a failing promise. Note that, to avoid the influence of network communication, the promise is resolved in the same virtual machine that created it.

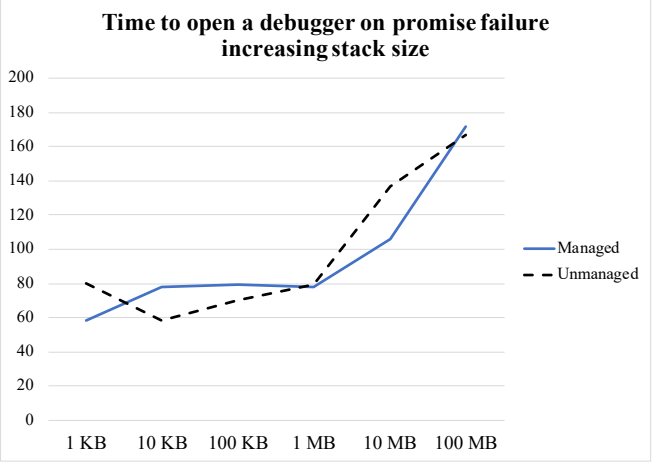


Figure 4.13: Runtime of a failing promise, when increasing the size of the stack.

Figure 4.13 shows the results of our benchmarks. When increasing the size of the stack our approach does not introduce a significant overhead. The black dashed line represents the execution time in the absence of our infrastructure (unmanaged), while the blue line represents the execution time in the presence of Sarto (managed). Both of the trends are linear to the amount of data, although, at first sight, it may look like the two curves are exponential: the X-axis grows exponentially by a factor of 10.

4.5 Notes to Related Work

In this section, we discuss several approaches that deal with domain-specific debugging of code, which relate to the concepts described in this chapter and complement the related work presented in Section 2.4. We do not include debuggers for Big Data frameworks, as they were already discussed in Section 2.4.5.2.

There exist several works in the literature about debugging techniques to help developers debug domain-specific code, specifically for framework executions. Particularly, many classical online debuggers offer primitives to hide from the call-stack those frames that include framework code. For example, when debugging Python code in GDB [GNU] developers can define different *frame filters* to hide some stack frames from the view. Similarly, Eclipse's debugging support offers a filtering operation that filters out some stack frames based on heuristics, e.g., the file or package where the executed method was defined. These approaches generally work for known frameworks but require in many cases to manually filter stack frames as part of the debugging process.

The moldable debugger [CDGN15] instead provides different views of the call-stack and the debugged method depending on which framework is being debugged. Developers can extend the debugger to display different information such as bytecodes or to target it to debug parsers, notification systems, etc. This approach mainly focuses on the visualization of the debugged information and requires interacting with the user interface of the debugger to extend it properly.

Debuggers for asynchronous execution While the above approaches represent a general take to the problem of debugging frameworks, several domain-specific debuggers have been designed to debug different programming and execution models. This is the case, for example, of debuggers for asynchronous execution models such as promises and actors. In an asynchronous promise execution, the thread that creates the promise has a relationship with the thread that is executing the promise. In a classical approach, however, the developer debugs only one of them, which is often the one that executes the promise. Different domain-specific solutions for better debugging asynchronous promise executions have been proposed. Dragos et al. [Dra13] focus on capturing stack frames at inter-

esting points such as future creation to enable debugging on them. Similarly, Chrome DevTools [Gooa] support debugging of asynchronous stack traces by storing such information in the stack trace. Other approaches [AZMT18, SBSB19] propose a graph visualization of the state of the promise, based on run-time information about the asynchronous promise, to help developers understand the execution. Leske et al. [LCN17] describe an approach in which the stack of a failing promise execution is linked with the stack of the promising thread, at the point of the promise creation. The developer then debugs a single stack that combines both the frames leading to the promise creation and the ones of the promise execution. In practice, the call-stack at promise creation is stored, to be then linked through a proxy to the call-stack of a failing execution of a promise.

As already partially described in section 2.4.2.2, different debugging approaches exist to debug actor programs. In the case of actors, the focus is not only on call-stack representation, but also on the display of actor-related information, such as mailboxes, or more complex information such as message causality [TLBS+17]. For example, Causeway [SCM09] logs the message sends and receive that happen through an actor concurrent execution. It then displays this information in the debugger UI showing how the different events relate through the *happens-before* relation, thus showing a partial order of messages that can help developers find a bug in their program.

While log-based solutions provide some interesting insight on past actor executions, online debugging approaches, such as REME-D [GNV+11], also enhance the debugging experience by providing domain-specific breakpoints on message send or receive (and more). Part of the breakpoint catalog offered by REME-D is inspired by the work on Wismuller [Wis97] on message breakpoints for the MPI model.

4.6 Conclusion

In this chapter, we explored a debugging instrumentation layer to enable framework-aware debugging. We introduced *Sarto*, a call-stack instrumentation layer that improves debugging of user code within framework code with six call-stack operations to tailor the stack accordingly to framework usage. With Sarto, framework developers define different entry/exit

points in their framework code to delimitate the information that may be hidden during debugging. They define custom stack frames to augment call-stacks with otherwise missing information. Our solution also offers operations to compose different call-stacks, e.g., to unify the execution of the promise with the promising stack, and to relate and compose different exceptions.

We applied our solution to three different use cases: an HTTP web server, a unit testing framework, and promise executions. They all use a different subset of the 6 operations. To show the validity of our solution we first presented our experience in instrumenting the different frameworks. Then, we conducted performance benchmarks to show that our approach does not add noticeable overhead in exception handling.

The concepts of Sarto presented in this chapter form the basis for the implementation of different features of our debugging approach for Big Data applications, as will be detailed in Chapters 6 and 7.

Chapter 5

Out-of-Place Debugging

In this chapter, we describe our novel debugging approach for Big Data applications which explores the idea of live debugging in the context of remote applications.

Recall from our comparison of debugging architectures in Section 2.4.4 that in-place debuggers execute in the same process as the debugged application, thus presenting low latency debugging operations, but no access to applications running remotely. A remote debugger, on the other hand, executes on a separate process than the application and thus performs debugging remotely, i.e., it sends operations via inter-process communication to the API of the debugger that then instruments the application. Debuggers for Big Data applications with online debugging features such as Daphne and BigDebug (cf. Section 2.4.5.2) adopt a remote debugging architecture. A remote architecture, however, needs to pause the execution of at least one remote node for debugging, thus possibly introducing delays in the computation. Furthermore, all side effects produced during debugging directly affect the execution of the debugged program, thus influencing its final results.

To tackle those concerns we introduce out-of-place debugging, a novel debugging architecture that combines the low latency of in-place debugging with the remote access of remote debugging, by moving the state of the debugged application to a different process to be debugged in isolation. This make sure that side effects are scoped to the debugger's process. Out-of-place debugging represents the main cornerstone of our debugging approach for Big Data applications.

5.1 The Out-of-place Debugging Model

The main goal of *out-of-place* debugging is to debug remote applications with low latency and without influencing their execution, i.e., avoiding residual side effects on the computation on the cluster. Similar to remote online debugging, an out-of-place debugger hosts the debugged application and the debugger in different processes. As such, the debugged application includes a debugging API in its infrastructure. However, in contrast to traditional remote debugging (e.g., [PBFD15]), out-of-place debugging transfers the entire debugging session (i.e., the state of the application) to the debugger process when the application reaches a breakpoint (or throws an unhandled exception).

This results in two properties. First, out-of-place debugging provides the user experience of an in-place debugger through debugging operations such as stepping, state inspection, and expression evaluation, without suffering from latency because all those operations happen locally. Second, since any manipulations and side-effects performed while debugging happen in the debugger process environment, they do not affect the debugged application. As a result, residual side-effects are *scoped* rather than global. These two properties of out-of-place debugging are relevant to the debugging of Big Data applications because they enable the use of a classical debugger on a captured remote computation, without replaying any part of the execution. This is possible for both unhandled exceptions and breakpoints.

In our work, we also embrace the support for dynamic code updates of a debugged application found in Smalltalk's live debugging model, by embedding it in our debugging approach. Particularly, in out-of-place debugging once the developer finishes debugging, they send a patch with the corresponding code changes. This allows the developer to debug in isolation the application, apply required changes, and update the remotely running application. This is particularly relevant in the context of debugging Big Data applications since it enables propagating large code changes to the remotely running system, without having to repackage the application or restart the whole system.

Overall, out-of-place debugging is a general-purpose debugging model usable in different contexts, and already experimented besides Big Data in the context of IoT devices [MGBC$^+$17, RCMBGB21]. In the remainder

of this section, we detail the concepts and components of the out-of-place debugging model.

5.1.1 Out-of-place Debugging Architecture

Figure 5.1 shows the architecture of an out-of-place debugger. Like remote debugging (cf. Section 2.4.3.2), out-of-place debugging enables the debugging of a remote application by being hosted in two processes: the debugger process and the application process. In the application process, the out-of-place debugger infrastructure consists of the *Debugger Monitor* and the *Updater*. The debugger process, besides the debugger UI, is composed of the *Debugger Manager*, and the *Changes Handler*.

In what follows we detail the role of each component when debugging an application. The numbers represent the order in which debugging operations take place when a breakpoint or exception halts the program's execution. The described debugging architecture, throughout its different components, represents the core model of out-of-place debugging. Debugging, however, is enabled by the concept of a *debugging session*, which captures the full application state and contextual information needed for debugging.

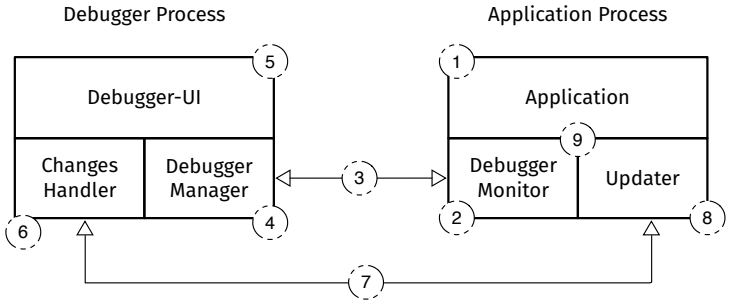


Figure 5.1: Overview of an out-of-place architecture setup in two different processes. The arrows represent inter-process communication: the one marked with a 3 transfers debug sessions, the one marked with a 7 transfers code changes.

Debugger Monitor. The debugger monitor is a component that resides in the application's process and takes the role of debugger API,

being in charge of communicating with the debugger process. Its main roles are to supervise and control the application execution. When the program hits a breakpoint or raises an exception (1), the debugger monitor suspends the program execution. Then it creates a *debugging session* that includes a copy of the execution state, i.e., the call-stack, and the associated application state (2). The debugging session is then serialized and transferred to the Debugger Manager (3).

Debugger Manager. The debugger manager resides in the debugger process and is the component that the debugger interface UI communicates to for initializing and performing debugging. Particularly, it deserializes the debugging session sent by the *Debugger Monitor* to recreate it in the debugger process. It then passes the reconstructed debugging session to the debugger UI for debugging (4). From the user perspective, debugging the application works similarly to an in-place debugger: through the Debugger-UI the developer can issue common online debugging commands such as stepping, expression evaluation, and state inspection (5).

Changes Handler. The role of the changes handler is to record all source code changes done by the developer in their IDE while interactively debugging the application (6). For this reason, it resides in the debugger process. The changes handler captures all code changes including class and method modifications, additions, and removals. Once the developer considers the code ready to be deployed to the remote application, they issue a commit operation. This generates a patch containing all the code changes that is sent to the *Updater* (7).

Updater. The updater resides in the application process, and its main role is to apply all code changes to the application that were recorded during the debugging session (8). It then notifies the debugger monitor which resumes the application execution (9) after updating the code.

In the next section, we describe in more detail what a debugging session consists of.

5.1.2 The Debugging Session

A crucial concept behind out-of-place debugging is the creation, transferring, and re-construction of the debugging session on a different machine. This enables an out-of-place debugger to (1) reduce latency during debugging because all of the operations happen locally, thus avoiding network communication (2) scope the side effects of the debugging session to the process of the debugger, while allowing the remote application to continue working in the case of a distributed parallel application, such as a Big Data application.

The process of creating a copy of the debugging session is akin to *remote cloning* in the domain of code mobility [FPV98]. We now further detail what creating a debugging session entails.

In most object-oriented programming languages, the application state is encoded as objects stored in memory, usually in the heap. The execution state is stored in a stack data structure (i.e., the call-stack) which references objects in the heap. Figure 5.2 illustrates how the stack and the heap are related in the context of an application that analyzes tweets. This application is used later in this dissertation while evaluating our solution (cf. Section 8.1.1.3). The figure shows that the `analyseTweets` stack frame points to instances of the `TwitterApplication` (i.e., the receiver) and `Tweet` classes (i.e., a local variable). Each of the stack frames contains local variables that reference different objects in the heap. In some cases, the stack frame also points to the object that the method is executed on (i.e., the receiver, often accessible through pseudo-variables such as *self* and *this*).

To create a debugging session, the debugger monitor extracts the call-stack to prepare it for serialization. In practice, by including the serialized call-stack, the debugging session includes two types of information: (i) information about the execution, i.e., the activated methods, their PC, and their order, (ii) the state of the application, i.e., state extracted from the heap that is referenced by the different stack-frames.

Similar to traditional remote debugging, out-of-place debugging assumes that the debugger environment has the same version of the code as the debugged application. This means that an out-of-place debugger does not need to copy and serialize the executed code, i.e., classes or bytecode. Not only does this simplify the creation of a debugging session but it also reduces the number of data transfers between the application and debug-

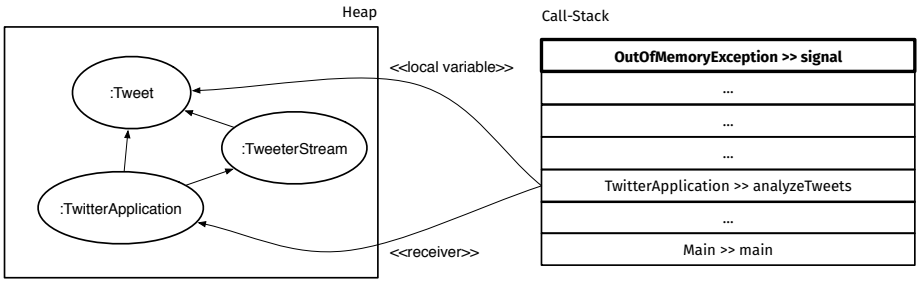


Figure 5.2: Relation between the heap and the call-stack in a sample application.

ger processes. In other words, an out-of-place debugger does not need to implement progress migration, which requires one to copy and transfer both code and execution state [FPV98].

5.2 Enabling Out-of-place Debugging

The main concerns in implementing out-of-place debugging are (i) how to capture a debugging session, (ii) how to perform code updates, and (iii) how to handle remote resources. In this section, we present details on how these three different concerns were approached when implementing out-of-place debugging for different programming models and runtimes. In particular, we describe our experience in applying out-of-place debugging to long-running distributed applications and IoT applications. In Chapters 6 and 7 we then detail how we implemented out-of-place debugging for Big Data applications in the Map/Reduce and Spark-like models.

There exists at the moment two main implementations of out-of-place debugging that target different runtimes and environments: (i) IDRA, The original implementation of out-of-place debugging in Pharo Smalltalk; (ii) WOOD, an adaptation of out-of-place debugging for debugging IoT applications [RCMBGB21], implemented by extending WARDuino [GSS19], a WebAssembly VM running on microcontrollers.

5.2.1 Capturing a Debugging Session

Depending on the execution environment on which out-of-place debugging needs to be implemented, capturing a debugging session can be challenging. For instance, in some environments the call-stack is reified and thus can be used directly to capture a debugging session without changing the runtime. In other environments, the developer needs instead to change the runtime to access this information.

Particularly, in Pharo Smalltalk the virtual machine deoptimizes and reifies the call-stack on demand [IMBGB20] and this reification is used in several reflective tools such as the Pharo debugger. Thus, in this dissertation we rely on the call-stack reification of Pharo to implement out-of-place debugging without changing the runtime.

When implementing out-of-place debugging in a VM that does not reify the call-stack such as WARDuino, the debugger implementor needs to extend the VM to extract the necessary information about the execution. This involves defining a format for representing the call-stack and using different techniques to walk through the values referenced by the stack. For example, to later reconstruct correctly a debugging session, it is necessary to manage absolute memory addresses, function pointers, and other low-level constructs.

Overall, these two implementations show that out-of-place debugging can be integrated into two different platforms at the opposite sides of the design space: one using existing reifications, and one modifying the runtime to extract the execution state.

5.2.2 Synchronizing the Codebase

When changing the code of the debugged application, out-of-place debugging models application updates through a commit operation. Detecting and applying changes, however, is not trivial for all platforms: some programming environments, like Pharo Smalltalk, reify code changes, others do not. Particularly, in Pharo detecting and applying changes happens through a library (i.e., Epicea [DCD13]) that records and reifies all changes that happen to the codebase through the IDE. It is also able, given some reified changes, to apply them in a different execution (i.e., in another process), provided that the starting codebase is the same. The Dynamic Software Update (DSU) support in Smalltalk (i.e., crating, modifying,

and deleting classes and methods at runtime) facilitate the task of implementing both the changes handler and the updater, since the debugger implementor does not need to add infrastructure to detect and apply changes.

DSU, however, is not a property unique to Smalltalk and it is present in many other platforms such as Lisp, Erlang, and WebAssembly. CLOS and other Lisp implementations allow developers to redefine classes, propagating the changes to active instances, as well as to redefine generic functions and methods. Erlang offers built-in support for hot code swapping through the recompilation of modules and, similarly, WebAssembly supports recompilation of modules, potentially enabling DSU for different languages that target it. Particularly, DSU can be implemented by recompiling full modules, instead of reifying and applying single code changes as in Smalltalk. Research has also explored the application of DSU in safe update points to avoid the impact of updating system libraryes [TPB+18]. Finally, other research has explored how DSU can be applied to mainstream languages missing native support such as Java [ORH02, PGS+11], Python [TZ18], and C [NHSO06].

5.2.3 Handling Non-transferable Resources

In out-of-place debugging the state of the application is moved from the application process, possibly located on a different machine, to the debugger process. The application, however, might reference *non-transferable resources*, i.e., external resources available only from the application process/machine such as files, sockets, and sensors. This raises the question of how to copy and transmit such non-transferable resources referenced by the debugged application.

To exemplify the issues of handling remote resources, let us consider an application that accesses the contents of a file, as shown in Listing 5.1. After the path is created in line 3, line 4 opens the file, returning a file stream. Then, a header is extracted in line 5 and checked for a pattern in line 6. Different results are returned depending on the header, returning in line 7 or 8.

```
1  FileAnalyzer >> analyzeFileNamed: aName
2  | aFileStream header |
3  path := basePath / aName.
4  aFileStream := (File named: path) openForRead.
5  header := aFileStream next: 2.
6  (header == #(0 1) asByteArray)
7     ifTrue: [^ aFileStream next: 10].
8  ^ aFileStream upToEnd.
```

Listing 5.1: Debugging a method accessing external resources.

Our approach is to instantiate a proxy so that calls to the non-transferable resources that happen in the debugger process are proxied to the original resource. proxied When this proxy is instantiated depends on where the debug session is constructed and on the environment.

In a high-level object-oriented language such as Pharo, if the non-transferable resource (i.e., the file in this example) is already referenced in the call-stack, then object-substitution is used to substitute the proxy to the non-transferable resource. This happens when the debug session is created after the execution of line 4. If the session is instead created when line 4 is executed, the debugging session captures the execution before the creation of the file object, so no file object would be included in the debug session. However, stepping through line 4 in the debugger process will attempt to open a file from the debugger process on a different machine, where the file is not available.

An out-of-place debugger thus needs to capture all accesses to possible remote resources while debugging is happening. Code instrumentation techniques, for example, allow us to substitute all accesses to pre-defined classes (such as *File*) with the instantiation of a proxy to the original resource. This operation happens transparently to the developer and is not visible in the code. Upon applying the instrumentation, the developer accesses the original file through the proxy. When implementing out-of-place debugging in Pharo, we use Metalinks [Den08], i.e., meta-objects that control the execution of AST nodes, to do code instrumentation. Particularly, during execution, a Metalink provides hooks to execute code before, instead, or after the execution of its annotated AST node. This facility gives fine-grained instrumentation at the sub-method level. By using Metalinks we transparently replace all accesses to external resource classes by accesses to a corresponding proxy class.

For those runtimes without meta-programming facilities for code instrumentation, supporting non-transferable resources requires changes to the runtime. In WARDuino, for example, remote resources such as sensors are not reified in memory, but they are represented by a certain function reference and thus object substitution is not needed. Particularly, in WOOD we extended the runtime to allow proxy calls on specific functions, specified beforehand by the developer. Calls to those functions are then substituted by the runtime with a proxy call to the application's process. This call then returns the actual value of the proxied resource (e.g. a sensor).

Overall, there are different strategies to handle non-transferable resources, and which one is used depends on the application domain and running environment. For instance, code instrumentation can easily be applied in environments with meta-programming facilities. Similar strategies, however, are also applicable in those environments that do not offer such facilities, e.g., WARDuino, by making changes to the virtual machine.

Scoping side effects in the context of non-transferable resources. Recall that out-of-place debugging ensures scoped side effects while debugging, i.e., side effects regarding local and global variables will be scoped to the debugger process. However, side effects may propagate to the target application when remote resources are used through proxies. For example, writing to a file may still perform the write in the file system of the running application. In those cases, other solutions such as mocking resources can be explored in combination with proxying, further scoping side effects.

Comparing to remote debugging. In an out-of-place debugger all the execution happens locally, i.e., all variable accesses are local, except for the proxied non-transferable resources. We can thus consider that in out-of-place debugging remote communication while debugging is limited to the initial transfer of the debugging session and to the proxified non-transferable resources. In contrast, in a remote debugger such as Mercury [PBFD15], the remote debugger for Pharo Smalltalk, all variable accesses during debugging are remote accesses through proxies. Particularly, Mercury uses proxies for the whole debugging session including the call-stack, the debugger model, etc. This leads to more network communication while debugging as all the stepping operations, expression evaluation, etc. have

to be applied through proxies. The network communication necessary for out-of-place debugging will be later assessed in the evaluation of our overall debugging approach (cf. Section 8.1).

5.3 Debugging Distributed Programs

As explained earlier in this chapter, the out-of-place debugging architecture devises the use of two processes: a debugger process and an application process. This makes the technique directly applicable to debugging an application that is executing on a different machine than the debugger. In this section, we discuss how the model also applies to applications that consist of more than one process, possibly running on different machines.

Out-of-place debugging can be used to debug distributed applications by running a debugger monitor in each application process and connecting them to a unique debugger manager. A debugger manager accepts debugging sessions from different application processes. This model also applies when different processes are running different applications, thus enabling to debug them in a centralized way from a unique debugger process.

Figure 5.3 shows a configuration similar to the one we used to debug cyber-physical systems [Mar17]. All application processes host a debugger monitor and an updater to enable out-of-place debugging of the distributed application. The debugger process instead hosts a debugger manager, the debugger UI, and a changes handler to detect changes to the application's code. Each monitor sends independently debugging sessions to the debugger manager, which is the one enabling debugging from a centralized point.

In the distributed setup, updating the code of the remote application now requires coordination to update all or only some nodes. This involves adding support for updating different processes at once, i.e., sending the same code updates to all connected processes, or selectively deciding which of the remote processes should receive the update and perform it. When committing code updates to a running distributed application, it is crucial to decide when to safely apply the update: applying an update at the wrong moment will disrupt the application by partially modifying the code that is being executed. In our example, when the developer commits, the changes handler creates a patch propagates it to the three different workers. If the patch is applied without a strategy, the update will be

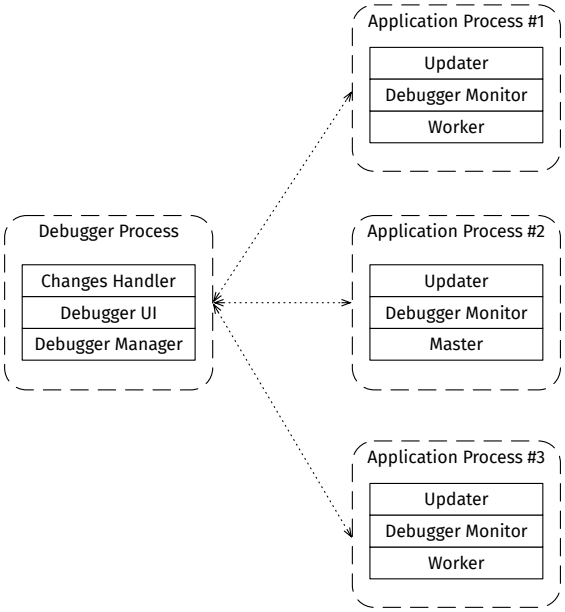


Figure 5.3: The out-of-place debugging architecture on a distributed system.

performed at different points in the execution in the different processes, possibly modifying the code of methods that are being run, thus leading to undefined behaviour.

We now further detail two different approaches for distributed software updates. First, we could opt for a stop-the-world solution, i.e., locking all the processes of the applications at specified synchronization points and applying code changes at that point in all the processes. This approach is however costly because it introduces delays in the execution resulting in bad responsiveness. Second, an asynchronous approach where updaters asynchronously decide when to apply code changes. This approach requires the design of application-specific updaters. In our debugging approach for Big Data applications, we decide to use specialized updaters that coordinate with the master to find the right moment to apply an update. Chapters 6 and 7 will describe more in detail how this works for Map/Reduce and Spark-like applications.

5.4 Conclusion

In this chapter, we have presented the out-of-place debugging model, the major building block to provide online debugging support for Map/Reduce and Spark-like applications. We now discuss out-of-place debugging with respect to the properties of debugging architectures discussed in Chapter 2.

Table 5.1: Overview of online debugging techniques based on their ability to capture bug context, scope side effects, and operate remotely, compared to out-of-place debugging.

Debugging Technique	*Capture Error Context*	*Remote Access*	*Latency*	*Side Effects*
Record & Replay	✗	~	High	Global
In-place	✓	✗	Low	Global
Remote	✓	✓	High	Global
Ours: Out-of-place	✓	✓	*Low*	*Scoped*

Table 5.1 revises Table 2.1 to include the properties of out-of-place debugging. As in-place and remote debugging, out-of-place debugging captures the context of an error when it happens, thus avoiding replaying the execution, which is typical of offline debugging approaches such as Record & Replay. Importantly, out-of-place debugging retains the remote access property of remote debugging, while providing the low latency of in-place debugging. It also keeps the ability to access non-transferable resources achieving a behaviour similar to remote debuggers. Finally, since debugging happens completely on a different process on a reproduction of the original execution, the side effects generated during debugging are scoped to the debugging session, and not global to the execution as in the other debugging approaches.

We now analyze out-of-place debugging w.r.t the criteria of a debugger for Big Data applications, defined in Section 2.5. Out-of-place debugging enables:

No Replays. Since debugging happens on a copy of a failed or breakpointed execution, it does not require replays of the execution to

reach the execution state of the error or breakpoint. This makes out-of-place debugging a replay-free solution.

Debug in isolation, scoping side effects. Out-of-place debugging ensures scoped side effects by enabling the debugging on a copy of the original execution.

Halt & inspect. In out-of-place debugging, breakpoints can be modeled as errors, thus they are captured by the debugger monitor and the developer can debug on a copy of this breakpointed execution.

Stepwise execution. Since the debugging over the reconstructed execution happens through a classical online debugger, out-of-place debugging ensures the availability of classical stepping operations.

On the other hand, out-of-place debugging does not satisfy the following criteria out of the box:

Scalability to Big Data. While out-of-place debugging enables replay-free debugging, this does not yet scale to Big Data since serializing full call stacks of an execution over a large amount of data might involve transferring very big debugging sessions.

Domain-specific debugging operations. Out-of-place debugging does not define any debugging operation tailored to the debugging of Big Data application, as it remains a general-purpose model for debugging distributed applications.

Live code updates. Out-of-place debugging offers an infrastructure for live code updating, but it needs to be adapted to the application domain. For instance, the propagation and coordination of code updates to a distributed system has to be adapted to the debugged execution model.

Ignoring of errors. Out-of-place debugging does not have explicit support for ignoring errors.

To adhere to all these criteria, we extend out-of-place debugging with necessary concepts to support the debugging of Map/Reduce and Spark-like applications in Chapters 6 and 7, respectively.

Chapter 6

Debugging Support for Map/Reduce

In this chapter, we present our debugging approach for Map/Reduce applications. Particularly, we start by describing an adapted out-of-place debugging architecture for the Master/Worker model, which lies at the basis of the Map/Reduce and Spark-like models. Then, we describe several extensions to the out-of-place debugging model specific to the domain of Map/Reduce programs, which introduce concepts for debugging a parallel execution in a centralized way, and some domain-specific debugging operations. After that, we describe a prototype implementation of our approach called IDRA_{MR} which employs some operations of Sarto (cf. Chapter 4). Finally, we evaluate our approach through a concrete debugging scenario.

6.1 Out-of-place Debugging for Big Data Frameworks

As mentioned before, out-of-place debugging is a good building block for debugging Big Data applications because it captures the error context of a remote application and enables scoped local debugging, thus enabling an in-place debugging experience that does not affect the remote application. Furthermore, the code updating capabilities of out-of-place debugging allow developers to update the code of a remote application without having to re-deploy it.

In this section, we revisit the architecture of out-of-place debugging to make it suitable to the Master/Worker model on which both Map/Reduce and Spark are based. This entails making sure the architecture is designed to centralize the debugging of a failure happening in a parallel execution context while reducing the communication overhead.

As shown in Section 5.3, the out-of-place debugging architecture is naturally distributed: a single debugger manager can connect to multiple debugger monitors at the same time, making it possible to debug different connected applications from a single point. This is important because when debugging a parallel execution across different workers, it enables the debugging of a bug that may manifest in different parallel tasks, raising multiple exceptions.

In the context of a Big Data application, however, deploying a debugging monitor on each worker and connecting it to an external debugging manager might lead to an increase in the amount of data to be transferred between single workers and the external developer's machine. We thus revisited the debugging architecture to include only one debugger monitor, hosted in the same process as the master. In this way, our approach limits the communication of the workers to the only master, internally to the cluster's local network. On the other hand, workers have to be extended to handle errors and breakpoints and report them to the master, which in turn reports them to the debugger monitor for debugging.

In Figure 6.1 we depict the revised architecture for debugging an application deployed on a cluster. The architecture thus presents a unique debugger monitor, hosted in the same process as the master. In this architecture, the debugger manager is hosted in a process external to the cluster execution, in the developer's machine, i.e., the machine the developer uses for debugging.

At the debugger process, alongside the debugger manager, there is a changes handler to record code changes that happen locally to the debugger session. Accordingly, the master and all workers run an updater, so they can be updated with new code when the developer decides so.

This architecture supports the deploying of out-of-place debugging on a Master/Worker model, thus enabling the out-of-place debugging architecture for Map/Reduce and Spark-like applications. As such, it lies at the basis of both our debugging approaches for Map/Reduce and Spark-like applications.

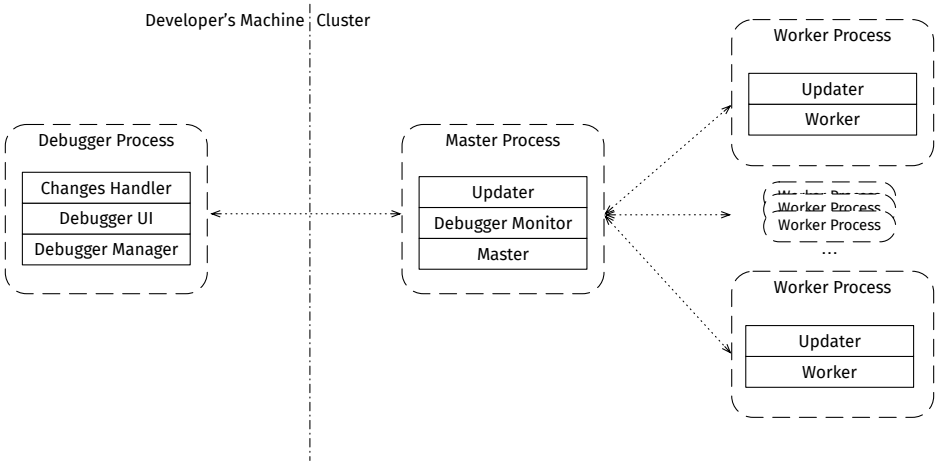


Figure 6.1: The devised out-of-place debugging architecture targeted to Big Data frameworks.

6.2 Debugging Map/Reduce Applications

To bring the out-of-place debugging model to Map/Reduce applications we first revisit three different concepts of the model. First, we decouple the concept of debugging session into two different steps:

Gathering debugging information. We introduce *debugging events* as a debugging session that gathers the call-stack and contextual information about the failure-inducing record and its partition. Debugging events are also designed to reduce network overhead when possible. This is further discussed in Section 6.2.1.

Centralizing the debugging session. To debug exceptions that happen in parallel, we combine the debugging events which are similar and raised from the same parallel execution to create unique debugging sessions, i.e., *composite debugging events*. This is further discussed in Section 6.2.2.

Second, we revisit the code updating capabilities of out-of-place debugging by handling distributed code updates of Map/Reduce applications and defining how to resume a Map/Reduce application upon an update.

This is important to let the developer choose what to re-compute once an update has been issued. We discuss this in Section 6.2.3.

Finally, we augment the debugger GUI to display composite exceptions, enable live code updating, and offer different debugging modes that allow developers to decide which data is used for debugging. This will be discussed later in Section 6.3.2.

6.2.1 Extracting Contextual Information into Debugging Events

During a parallel execution, an application may reach different *halting points*. A halting point is a point of the execution in which the execution is paused by a breakpoint inserted by the developer or by an unhandled exception. In our model, when a halting point is reached in a Map/Reduce worker, the worker generates a *debugging event*. A debugging event contains all the information necessary to construct a debugging session. Since we use the out-of-place debugging model, this entails the call stack and the different variables it references.

Reducing the size of debugging events. As detailed in Chapter 4, however, the debugging information present in the call-stack throughout a framework execution contains information related to the framework execution such as stack frames related to scheduling. Hence, not all of the frames are needed for debugging the user code of a Map/Reduce application. Including this information would only increase the amount of network communication to enable debugging.

To show this issue, consider as an example a Map/Reduce application running in Port. When a halting point is reached in a worker, the call-stack typically includes (i) frames representing method calls to the framework to initiate the execution, followed by (ii) a call to `map:` or `reduce:`, which leads to several calls to user-defined code. Figure 6.2 shows a representation of such a call-stack. Following the same color combination as defined in Chapter 4, we depict in red the framework codes, in orange the framework exit point, in this case the call to `map:`, in blue the framework(s) associated with the user code, and in green the frame representing the signaling of the error. To remove this information, we employ the *stack cutting* operation of Sarto. This will be further discussed in Section 6.3.1.

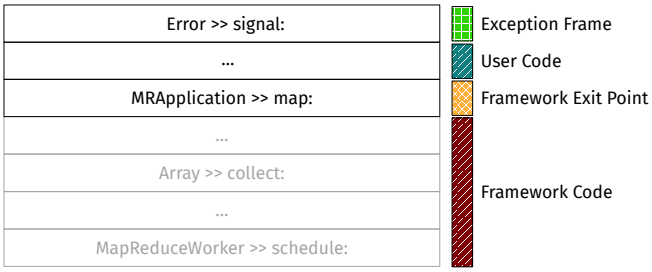


Figure 6.2: The simplified stack of the exception. Depicted in red the framework frames. Depicted in orange the last framework call before user code. In blue, the user code frames, and in green the frame causing the halting point.

Including failure-inducing records. When debugging Map/Reduce applications, it is crucial to have access to the data that makes an application fail to find the root cause of the bug. Sometimes, however, this record may not give enough context to debug the application when compared to having other data points from the same partition. For instance, a Map/Reduce execution is applied record by record in the map phase, and key by key in the reduce phase, i.e., the `map:` method is applied record by record within a particular partition/split of the dataset and the `reduce:` method is applied to a particular key and the set of its values. We call the record that was being analyzed (i.e., passed as a parameter to either function) *event inducing record*. Our approach is to have configurable debugging events that, upon the choice of the developer, can include just the failure-inducing record or all of its partition. This enables configurable local debugging of a remote Map/Reduce execution.

6.2.2 Centralizing the Debugging Session with Composite Events

As described above, a debugging event contains all the contextual information about the halting point. This debugging event is sent to the master which, as explained before, allows to centralize the debugging session. Note that the Map/Reduce master has global knowledge of the distributed program execution and status, not only of the failed worker(s) but also of the rest of the running tasks of the application. As such, while creating

a debugging session the master augments the debugging event with information about the executed operation (e.g., the identifier of the current execution, partition information, etc.).

The produced debugging event is then ready to be handled by the debugger manager, that then looks into the different events to conceptually merge the ones that are related into a unique *composite debugging event*. Let us show a concrete scenario in which this applies. While different Map/Reduce workers are performing parallel map and reduce tasks, the same bug may raise multiple exceptions while analyzing different data partitions in different workers. For instance, while parsing formatted data from a dataset, if more than one record has the wrong format, then the same failure will occur many times during the parallel execution. This generates many individual debugging events that have to be processed by the debugger monitor. All these events, however, conceptually belong to a single failure that manifested in different portions of data. For this reason, they will present the same identifier, since they are raised during the same execution.

To assess whether two events are part of the same composite event, the debugger monitor first checks that the execution identifier matches, i.e., whether the events were raised during the same parallel execution. If the event is raised by an exception, it checks whether the type of the exception matches the two events. The *stack comparison* operation defined in Chapter 4 is then used to determine whether two exceptions are similar, as detailed later in Section 6.3.1.

The debugger monitor then sends first a message to the debugger manager to indicate a composite event has been created, together with the debugging event generated upon the first event. Then the monitor sends update messages to the debugger manager containing the call-stack and metadata information of the following debugging events that are part of the same composite event.

Through this process, similar debugging events that happen during the same parallel execution are aggregated into a *composite event*, thus enabling the centralization of the debugging session for that composite event.

6.2.3 Live Code Updating and Resuming the Execution

Recall that in out-of-place debugging a developer the changes a developer applies to fix bug are recorded, and the developer can use the UI to create a code patch and propagate it to the master and the workers.

The code patch is propagated by the changes handler to the updater instance running alongside the Map/Reduce Master. At this point, the Updater notifies the master of the code patch, and in turn, the Master schedules a special updating task. Through this task, the worker contacts the updater to perform the updates indicated in the code patch.

Note that the update propagation does not happen atomically in all workers at once since each worker will apply the updates only when it finished executing the current task. Our approach, however, ensures that the different workers are not running a different codebase when executing a particular task. The Master makes sure that the update task is scheduled between different tasks, i.e., not in the middle of the execution of a map, or when the worker is idle. The way the update is propagated thus represents a variation of out-of-place debugging, designed specifically for the Map/Reduce model.

Once the code changes are deployed the debug session is finished, developers are offered the following operations:

1. Re-schedule all partitions that halted. This avoids the re-execution of tasks that finished with success.

2. Re-schedule the application from the start on all the partitions. In case the modified code requires an entire re-execution.

The first option is particularly useful when only a small part of the computation failed due to a few failure-inducing records. In this case, the developer preserves most of the execution, avoiding tedious replay times, and restarting only the failed part of the computation.

6.3 IDRA$_{MR}$: An Out-of-place Debugger for Map/Reduce

We prototype our debugging solution in IDRA$_{MR}$: an out-of-place debugger for Port Map/Reduce applications. The implementation relies on

different stack-tailoring operations of Sarto, discussed in Chapter 4. The GUI of the debugger offers dedicated views for the displaying of composite debugging events and code updating. Furthermore, it offers domain-specific debugging modes to select which records to use for debugging. While debugging, the developer uses classical online debugging features including common stepping operations such as step into or step over, and other debugging primitives such as restart, i.e., a debugging operation to restart the execution from a particular frame of the call-stack.

In this section, we first present how Sarto's operations are used to build our debugging solution and then describe the dedicated views and debugging modes that enable domain-specific debugging of Map/Reduce applications.

6.3.1 Sarto's Operations in IDRA$_{\mathbf{MR}}$

In our implementation, we use three of Sarto's stack tailoring operations to (i) improve the size of debugging events, (ii) identify similar debugging events to construct composite debugging events, and (iii) enhance the debugging experience.

Improving the size of debugging events. A debugging event includes the call-stack of the halting point, together with information about the event-inducing record. To reduce its size, before sending the debugging event from the worker to the master we perform the *stack cutting* operation to cut framework stack frames.

To do so, we have marked the two API methods of Map/Reduce, thus `map:` and `reduce:`, to be the framework exit point. All the methods called before the framework exit points are framework code, thus there is no need to add a framework entry point. We then apply stack cutting removing all framework frames and thus all reference to the worker, other data it references, etc.

Identify similar debugging events. Recall that the concept of *composite debugging events* is an abstraction of the same event (i.e., an exception or breakpoint) that happened multiple times during the parallel execution of a task, by using the *stack comparison* operation of our Sarto library (cf. chapter 4).

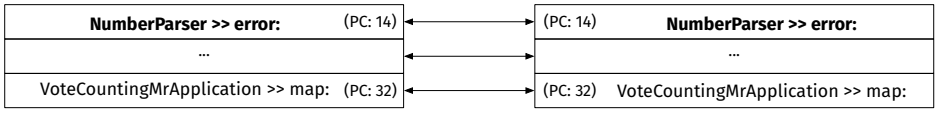


Figure 6.3: Two similar call stacks related to the same exception.

Figure 6.3 shows a simplified representation of two call stacks of exceptions part of the same composite event. In practice, through the stack comparison operation we consider two events to be similar when (i) they are generated by the same operation (e.g., `map:` in this case) and (ii) each of the stack frames, in order top to bottom, has the same method selector and points to the same program counter (PC). At this point, the call-stacks of the two similar events are aggregated in the same composite event. All following debugging events are compared with the first event; if they are similar, they are added to the same composite event.

Enhancing the debugging experience. Before opening a debugger, the call-stack is further tailored by using another stack-tailoring operation from Sarto: crafting a stack frame. In particular, since the stack was cut at the stack frame representing the `map:` method execution, an artificial frame is added to the bottom of the call-stack, in which the call on the (virtual or real) partition is performed, followed by a call to `reduce:`. This ensures that the developer can debug locally a complete Map/Reduce execution over the records present in the debugging event. For example, they can debug the map and then proceed to the reduce to further analyze the execution.

6.3.2 Domain-specific Debugging Operations

Debugging is enabled in the main view of IDRA$_{MR}$, displayed in Figure 6.4, marked with an A. This view is the one presented to the developer when they access, within the Port UI, the tab dedicated to the debugger. We display the debugging of an error raised in the elections poll application introduced in Section 3.4.1. Through this interface, developers see the debugging events that happened across the execution that they are monitoring, and they can analyze their stack-trace, the event-inducing

events, and they can start debugging an event of their choice. We mark with B and C the zoomed-in details of the view.

Particularly, on the left side of the main view, marked with a B, IDRA$_{\text{MR}}$ displays the list of distinct debugging events, only one exception in this case. The number 3 between square brackets denotes the number of times this exception was raised across the same parallel execution, meaning this is a composite debugging event for the three exceptions. On the right side of the main view, marked with a C, IDRA$_{\text{MR}}$ displays the stack and the different records that caused the exception.

The debugging Modes The overall view (Figure 6.4 A) allows developers to select a composite event, look at the call-stack, and inspect (through the *inspect selected* button) the different records that caused it. Furthermore, the buttons in the bottom right part, under the overview of the event inducing records, allow the developer to start a full live debugging session by using three different debugging modes.

The debugging modes enable the developers to decide the amount of contextual information available during debugging. As mentioned before, data in the Map/Reduce model is split into different partitions (or splits), hence every event-inducing record has an associated data partition. By collecting data from the partitions that contained the event inducing records, we offer different modes that contain more or less contextual information. IDRA$_{\text{MR}}$ provides:

Debug a single halted record. Developers start debugging the `map:` on one of the event-inducing records. Once the map on the associated record returns, the developers continue debugging on the rest of the records in the same partition. If no partition was included in the original debugging event (cf. Section 6.2.1), the partition associated with this event will include only the event-inducing record.

Debug a virtual partition with all halted records. The developers debug the `map:` on a virtual partition containing only the event-inducing records, regardless of their original partition. In our example, this operation will construct a virtual partition containing all records visible in the bottom part of Figure 6.4 C and let the developer debug the `map:` on this virtual partition.

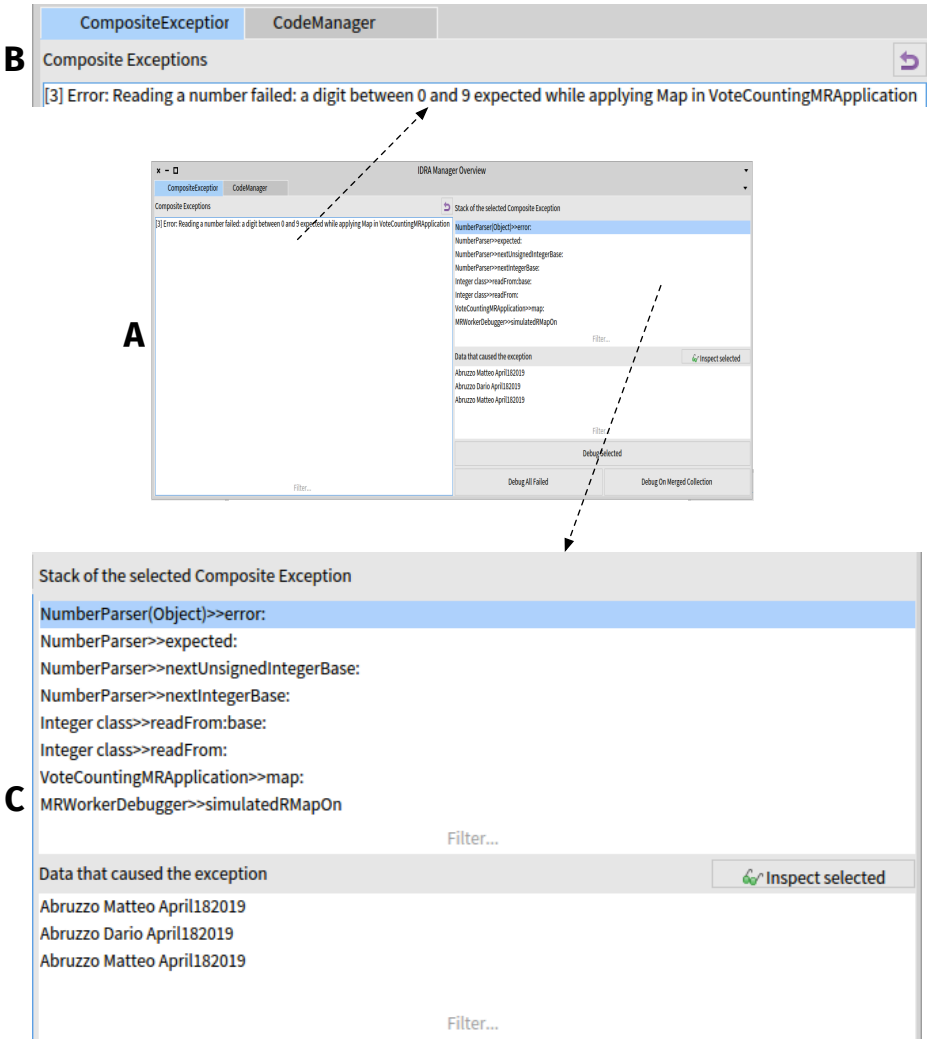


Figure 6.4: A screenshot of the IDRA$_{MR}$ UI when handling a composite event. A shows the main view, while B and C show detail respectively in the overview of composite exceptions, and the call stack and data view.

Debug a virtual partition with all halting partitions. The developers debug the `map:` on a virtual partition which is the union of all of the partitions that contain at least one halted record. This virtual partition will contain all records in those partitions, including those that do not halt. If no partition was included in the original debugging event, this operation is equivalent to the second one.

The selected mode affects the point from which the developers start debugging in relation to the crafted frame (cf. Section 6.3.1). For example, when debugging a single record debugging starts at the point the halting point was reached. When the developer continues debugging and returns to the `map:` method, the debugging will continue on the application of the `map:` to the next available record. If records are finished, the debugging session returns to the crafted method to proceed with the local `reduce:`. Instead, when debugging on virtual partitions the developer starts debugging from the crafted method and is able to step into the execution of `map:` of the different elements.

Committing code changes. Recall from Section 6.2.3 that the changes handler records all code changes that happen during debugging. Through the debugger, developers have access to a dedicated view to visualize and commit those code changes. Particularly, the code changes are visualized in the *Code Manager* tab of IDRA$_{\text{MR}}$, displayed in Figure 6.5. The right side of the code manager shows all of the changes made by the developer while debugging, and the diff between such code changes to the original versions. By clicking on the *commit changes* button, the developer sends a code patch to the master through its updater.

6.4 Evaluation

In this section, we evaluate IDRA$_{\text{MR}}$ using an application in the domain of blockchain analysis to index all the blocks of a blockchain (i.e., Ethereum).

More concretely, we developed this application in collaboration with researchers of the RMoD team in INRIA Lille to experiment with port and IDRA$_{\text{MR}}$. We first describe the implementation of the blockchain indexing application and then delve into the debugging cycle using IDRA$_{\text{MR}}$, showing how different errors can be detected and debugged. The described

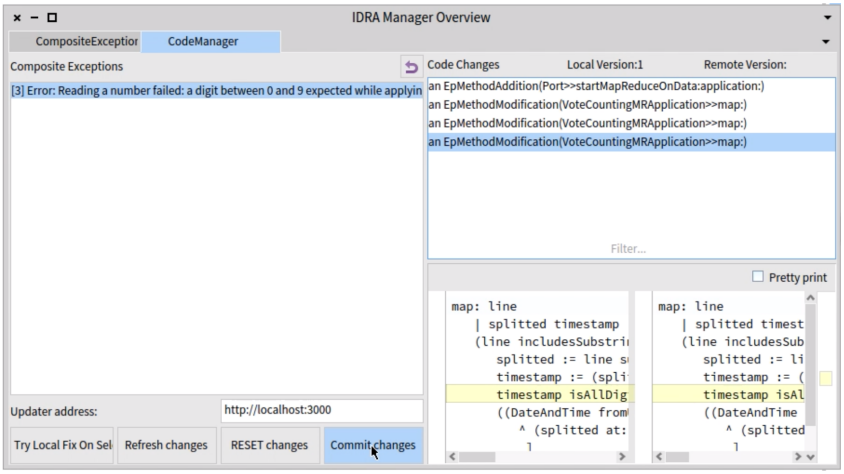


Figure 6.5: A screenshot of the code manager tab of IDRA_{MR}.

errors are actually some of the problems reported by the collaborating researchers when implementing this application in Port.

6.4.1 Blockchain Indexing in Port

The application we designed for this evaluation uses an indexing algorithm to index every block in the blockchain. Indexation is the first step to enable further analysis of the blockchain data.

We have developed an indexing algorithm that uses a relational database to store indexed data. Our index has the structure of a relational table with standard database indexes. For example, the table representing the block index has the block's hash, but also a timestamp and its parent block's hash. Figure 6.6 shows an example of a blockchain index table in which the two latter columns are indexed, so we can do fast queries on blocks by both timestamp and their parent blocks.

To set up such an index, our core algorithm performs a full scan of the blockchain inserting all the corresponding values in our database.

Listing 6.1 illustrates the implementation of this application in Port. The `map:` method (lines 1 to 6) obtains a block given a block index through a call to the blockchain driver, extracts the indexed property from the given block, and returns a key-value pair with the block index as key and

Block hash	Timestamp	ParentBlock
ca896d6	28/01/2019 ...	da6b261
da6b261	27/01/2019 ...	7aa96ae
7aa96ae	26/01/2019 ...	d6d3614
d6d3614	25/01/2019 ...	402d518

Figure 6.6: Example of a block index table. The Timestamp and ParentBlock columns are indexed.

the indexed value as value. For example, in the case of indexing blocks by timestamp, the indexed value is the timestamp of the obtained block. The `reduce:` method (lines 8 to 9) receives a collection of pairs produced by the `map:` method and instructs the database to store the indexed. values

```
1 MRIndexingApp>>map:blockIndex
2   | ethereumBlock mappedProperty |
3   ethereumBlock := BlockChain at: blockIndex.
4   mappedProperty := ethereumBlock
5     get: #timestamp.
6   ^ blockIndex -> mappedProperty
7
8 MRIndexingApp>>reduce:pairs
9   Database storeIndexedValues: pairs.
```

Listing 6.1: Pharo implementation of a blockchain indexing algorithm.

We use Geth [1] as blockchain data node. The communication with Geth is managed by the Fog Ethereum driver for Pharo [2]. Finally, we use a Postgres database to store the indexed data.

6.4.2 Experiments

In this section, we describe our experiments for executing and debugging the blockchain indexing application described above. These experiments show two things: first, the scalability of Port in executing applications on a large amount of data; second, a concrete debugging experience using our debugger on a real error we encountered while developing and executing the application.

[1] `https://github.com/ethereum/go-ethereum/wiki/geth`
[2] `https://github.com/smartanvil/Fog`

In what follows, we first describe the setup and then delve into the two experiments.

6.4.2.1 Setup

We run our experiments on a cluster composed of one *root* node and ten identical *slave* nodes. Each node presents an Intel Xeon CPU E3-1240 @ 3.50GHz, 32 GB of RAM, and 200 GB of SSD Storage. Nodes are connected via a 1 Gb/s local network.

For all the benchmarks, we deploy Port on the cluster using Hadoop Yarn, and we use 1 single-core master, and, depending on the benchmarks, several single-core workers. Hadoop Yarn takes care of the allocation of the master and workers on the cluster. The cluster runs Pharo 8.0.0 (x64) on a Pharo 8.3.0 Headless VM.

We control the execution and perform debugging from a 2017 MacBook Pro, running an Intel Core i7-7567U @ 3.5GHz CPU, 16 GB of RAM, and 500 GB of SSD storage. This machine uses SSH tunneling to communicate to the cluster. On this machine we run the same version of Pharo as on the cluster, but with the full VM, including UI, rather than with the headless one.

Before starting our experimentations we setup a Geth node (version 1.8.17-stable) with the following command

```
1 geth --datadir ./devdata --rpcapi eth,web,net --syncmode fast
    --cache 2048 --gcmode full
```

We waited several days until the node was completely synchronized with the blockchain to then run our blockchain indexing algorithm.

In our configuration, the root node runs:

- A Postgres server that handles the database.
- Port's Yarn deployer (cf. Section 3.6) to handle the external communication with the different nodes.

One of the slave nodes runs exclusively Geth, the blockchain data node. The rest of the slave nodes are used to deploy Yarn containers that run the Port master and workers. There is always one container running a Yarn application, one running a master, and a global maximum of 70 containers available to each run a Port worker. The number of actual containers running a worker varies depending on the experiment.

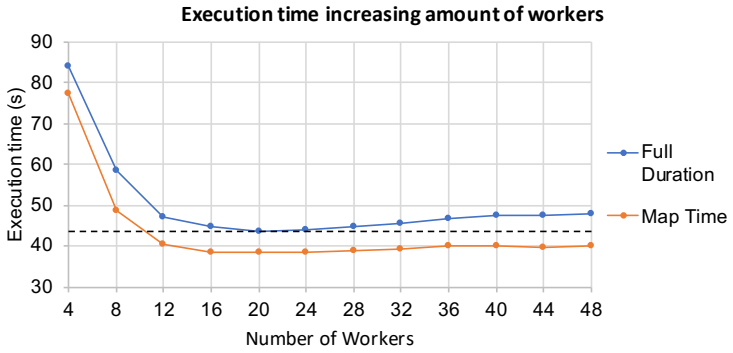


Figure 6.7: Execution time of indexing 10.000 block increasing the number of workers.

6.4.2.2 Experiments on Scalability

Indexing the whole blockchain, even using a Map/Reduce approach, is not trivial. Not only does the performance of the indexing depend on the number of workers that parallelize the work, but also on how these workers communicate with the blockchain data node, during the mapping phase, and with the database, during the reduce phase. While the employed database (i.e., Postgres) is designed for concurrent access and scales well to concurrent requests, the blockchain data node (i.e., Geth) queries the blockchain using the RPC protocol, which can support only a limited number of concurrent calls.

Before running this experiment, we assessed through a small benchmark the ideal number of workers to use in our configuration. We indexed a fixed number of blocks (10.000 blocks), variating the amount of workers from 4 to 48.

Figure 6.7 shows the result of this preliminary benchmark. The blue curve represents the average execution time of the full Map/Reduce, while the orange curve represents only the average time of the map phase. The black dashed line represents the average minimum amount of execution time (i.e., 43.6 seconds with 20 workers). Hence, we selected 20 as the number of workers to use in the rest of our experiments.

To further assess the scalability of Port, we run our indexing algorithms to increasing portions of the Ethereum blockchain to finally index the

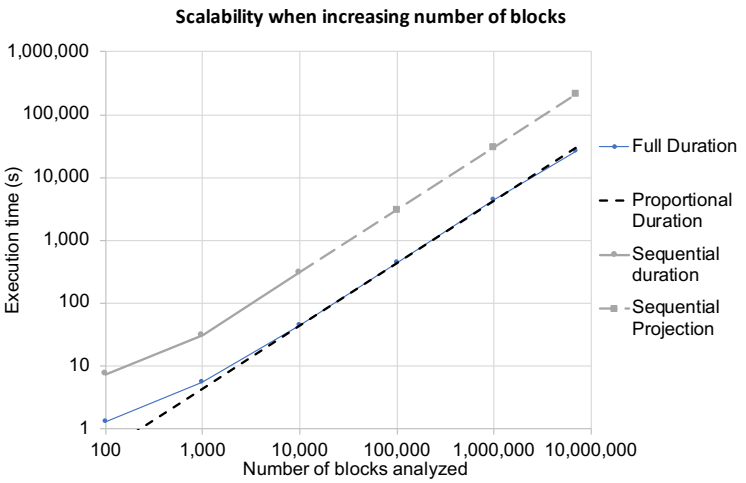


Figure 6.8: Execution time of indexing an increasing amount of blocks with 20 workers.

full blockchain. At the time of our latest experiments (29/01/2019), the Ethereum blockchian consisted of 7.080.006 blocks.

Figure 6.8 shows how the execution time increases when increasing the number of analyzed blocks. The blue line represents the execution time, the grey line represents the execution time of the sequential implementation of the indexing algorithm. The values of the dashed gray line are proportionally projected since the execution time was too high to experiment with. The dashed black line represents what the proportional expected duration is, calculated on the result for 10.000 blocks. Both scales are logarithmic (log_{10}), hence the two axes are proportional.

The graph shows that the execution time grows linearly, as the black dashed line, except for 100 and 1.000 blocks, where the execution time is impacted by the overhead of Port.

Using 20 parallel single-threaded workers, we managed to index the full Ethereum blockchain in 7 hours, 18 minutes, and 47 seconds. While the experiments on up to 100.000 blocks were repeated at least 10 times, due to time limitations the experiments over 1 million blocks were executed only 4 times and the one on the full blockchain only once.

6.4.2.3 Debugging an Error Using IDRA$_{MR}$

To demonstrate the debugging capabilities of our solution, consider a configuration and installation error in which we did not load the driver to the Postgres database in the codebase of the Pharo image deployed on the cluster, and used by the different master and workers. This is akin to not correctly packaging a library in the jar submitted for execution to Hadoop Map/Reduce or Apache Spark. Our application will correctly run through the execution of the map, but suddenly fail when executing the reduce.

While classic approaches will let the application crash and require log analyses to find the problem, Port reports to IDRA$_{MR}$ an exception, in the same way as a classic application exception, in the UI previously showed in Figure 6.4. At this point, the developer will see an `UndefinedClassError`, stating that the undefined class cannot be instantiated, and several key-value pairs as causing the error.

At this point, the developer loads the library locally, and those code changes are detected by the changes handler. A patch is then committed to updating the codebase of Port, and the developer restarts only the reduces since the intermediate data has been persisted on disk.

Debugging such configuration errors with IDRA$_{MR}$ provides an immediate view of an error in the remote cluster. Furthermore, the code updating capabilities of our approach avoid invalidating a correct execution. The support for library code update avoids the hassle of packaging errors and related re-compiling and re-deployment steps. This is particularly useful, especially when configuration bugs appear only in a late stage of the computation, as in this example.

6.4.3 Discussion

In this section we have evaluated our approach in a realistic scenario, showing both the scalability of Port and a concrete usage of IDRA$_{MR}$. We now discuss our debugging approach w.r.t the different criteria of a debugger for Big Data applications defined in Section 2.5.

In Table 6.1 we revisit Table 2.2 to include IDRA$_{MR}$. Particularly, IDRA$_{MR}$ offers scoped side effects and replay-free debugging. Furthermore, it supports halting the execution to inspect the execution state and the use of classical stepping operations. With the introduction of

Table 6.1: Comparison of IDRA_{MR} with related work w.r.t. the criteria defined in Section 2.5.

Debugger	*Side Effects*	*Replay Point*	*Halt & Inspect*	*Stepwise Exec.*	*Dom.S. Ops.*	*Code Updates*	*Ignore Errors*
Arthur	Both	Start	✗	✓	✗	✗	✗
Graft	Both	Start Rec.	✗	✓	✗	✗	✗
Daphne	Both	Check.*	✓	✓	✗	✗	✗
BigDebug	Global	Check.	✓	✗	✓	✓*	~
IDRA_{MR}	Scoped	No	✓	✓	✓*	✓	✗

the debugging modes, IDRA_{MR} supports domain-specific debugging of Map/Reduce applications but does not include support of domain-specific stepping operations. Regarding code updates, we showed in this section how IDRA_{MR}'s code update capabilities can be used to update a remote application without redeploying. Finally, IDRA_{MR} does not support the systematical ignoring of errors.

During our experiments, we also experienced some problems with the scalability of our approach. When developing and debugging the blockchain indexing application, for example, we choose to include the partition of the failure-inducing record in the debugging session. As a result, while the debugging experience was smooth when analyzing a low number of blocks, the response time of the debugger increased when analyzing more and more data. We also experienced this not only when analyzing increasing data sizes, but also when having multiple parallel exceptions, which produced composite exceptions containing several call stacks. For these reasons, we further investigated more optimizations to make the debugging experience more practical, which we describe in Chapter 7.

6.5 Conclusion

In this chapter, we presented our solution for debugging Map/Reduce applications, based on the concepts of out-of-place debugging, previously introduced in Chapter 5.

Composite events enable the centralization of the debugging experience of debugging events raised during a parallel and distributed execution. To reduce data transfers and improve the debugging experience we make use of Sarto, namely of its stack cutting and crafting of a stack frame operations.

Furthermore, we introduced three different debugging modes for domain-specific debugging of a composite event, allowing developers to debug on virtual partitions of data containing, for example, all the failing records. In this way, the developer can test locally its solution on a portion of the data, before committing the code updates to the remote system.

We integrate the above concepts in a concrete prototype implementation: IDRA_{MR}, a debugger for Port (cf. Section 3.4) in the Pharo Smalltalk IDE. We provide views for the overview of composite events and for code updating, and we rely on the Pharo Debugger for the local debugging of the reconstructed execution. Finally, we assessed the robustness of Port by devising a blockchain analysis application for it and showing a debugging example that highlights both the immediate debugging feedback and code updating capabilities of our solution. More work, however, is necessary to adapt this model to Spark-like applications, and to improve performance and debugging experience of an out-of-place debugging approach for Big Data applications. This will be discussed in the following two chapters.

Chapter 7

Debugging Support for Spark-like Applications

In this chapter, we present our debugging approach for Spark-like applications. As for debugging Map/Reduce applications, we build our approach on top of out-of-place debugging (cf. Chapter 5) using the debugging architecture already presented in Section 6.1. On the one hand, we improve the scalability of out-of-place debugging with several optimizations and we expand our support for domain-specific debugging by introducing different stepping operations tailored to the Spark-like model. On the other hand, we introduce a solution for ignoring errors which extends the failure model of a Spark-like model with the ability to ignore application failures up to a custom threshold, inspired by acceptability-oriented computing [Rin03] and relaxed programs [CKMR12].

In what follows, we first describe our debugging approach by further refining the concepts of debugging session into *dynamic local checkpoints* and introducing the domain-specific stepping operations, complemented with our solution for supporting relaxed computations. Then, we present an implementation of our debugging approach called SpaDebug which is built on Sarto's concepts.

7.1 Debugging Spark-like Applications

As explained in the previous chapters, out-of-place debugging provides local debugging of remote failures. Particularly, when an exception hap-

pens in the cluster, part of the execution environment (i.e., the execution stack with its associated data) is transferred to the developer's machine to create a debugging session that enables local debugging of the remote computation. As a result, developers can debug errors in isolation while avoiding replays.

In this chapter, we build on the debugging model presented in Chapter 6, reusing the concepts of composite debugging events for debugging Spark-like applications. However, we need to consider that the execution model now relies on a distributed data structure with a larger functional API than Map/Reduce. Overall, our debugging approach for Spark-like applications focuses on the following aspects:

Debugging events. We revisit the concept of debugging events to form *dynamic local checkpoints*, i.e., debugging events that store more information about the event to include the lineage of the current execution, at least part of the current data partition, and part of the original partition.

Scalability. We optimize dynamic local checkpoints to reduce their size making our approach more scalable.

Domain-specific Stepping Operations. We propose a combination of fine-grained stepping, i.e., classical stepping operations, and domain-specific stepping operations that allow developers to step through a Spark-like execution based on its functional model, e.g., to step to the next iteration, transformation, and the final result of the debugged execution.

Ignoring errors. We enable different debugging modes to allow developers to ignore errors in applications that can accept a loss of accuracy. This requires revisiting the concept of exception handling of the Spark-like model, since we should now be able to ignore failures according to the selected mode.

In what follows, we first describe dynamic local checkpoints and the optimizations that make them practical, followed by details on how we reconstruct the execution for debugging, and how we augment the debugging with debugging operations specific to Spark-like applications.

7.1.1 Extracting Contextual Information with Dynamic Local Checkpoints

We now describe the concept of *Dynamic Local Checkpoints*, an evolution of the concept of *debugging events* originally devised for the Map/Reduce model and that we now apply to a Spark-like model. Similar to debugging events, at the moment an exception is thrown or a breakpoint is reached, a dynamic local checkpoint is created by the worker through an exception handler, and the call-stack is extracted and cut to exclude framework-related stack frames. In practice, a dynamic local checkpoint includes the call-stack of the exception, the partition of the event-inducing record, and its lineage. Having this information allows a full reconstruction of the execution for debugging, thus a more complete debugging experience for the developer that can step into the execution not only of the event-inducing record, but also of other records in the same partition.

To reduce the size of dynamic local checkpoints, we include two crucial optimizations: *partition windowing*, i.e., cutting the partition of the event inducing record, and *delta composition of events* to reduce to the minimum the amount of information of a composite debugging event and thus the amount of communication between the cluster and the developer's machine. In what follows, we detail the two optimizations.

7.1.1.1 Partition Windowing

Including in a dynamic local checkpoint the event-inducing record as well as at least part of its partition, increases the amount of contextual information on the debugging event, which is available for debugging. However, partitions may include GBs of records, making the dynamic local checkpoint's size potentially very big. To make debugging practical, we perform *partition windowing*, i.e., include a reduced version of a partition by cutting it around the event-inducing record.

In practice, after capturing the data necessary for a dynamic local checkpoint, the worker trims the data of the failing partition around the failure-inducing record to only include a subset of the partition's records. For example, if the runtime was applying a `map:` over a partition of 3000 elements, and the element at index 1234 fails, the failing partition is cut to contain only a window of N elements, with indexes $[1234 - (N/2), 1234 + (N/2)]$, adjusted to respect the bounds of the partition.

The window size parameter N is a customizable parameter that allows the developer to choose a balance between performance and contextual information: setting N to 0, results in the dynamic local checkpoints including no partition data, except the event-inducing record. Setting N to higher values, in turn, will include in the dynamic local checkpoints records that did not cause a failure, i.e., the ones preceding the failure-inducing record, and the ones that can potentially cause one, i.e., the ones following the failure-inducing record.

Partition windowing heavily reduces the size of a dynamic local checkpoint, thus reducing network overhead when transferring a debug session. The impact of partition windowing on the size of dynamic local checkpoints will be assessed in Section 8.1.4.

7.1.1.2 Composite Debugging Events with Delta Stacks

To further reduce network overhead, we revisit the concept of *composite debugging events* to work with dynamic local checkpoints and further reduce their network overhead. Recall that composite debugging events aggregate many similar debugging events that occur in the same parallel execution regardless of their data partition or worker nodes. We consider two dynamic local checkpoints similar if the associated events (i.e., exception or breakpoint) are of the same type, and they happened at the same point of the execution, i.e., they present the same call-stack and program counter.

Upon a composite debugging event, we extract the execution stacks of two similar exceptions and for each pair of stack frames, we compute their *delta stack frame*, i.e., a stack frame presenting only the value of the variables that differ between the two stacks, and placeholders for the ones that do not change. We then serialize only one of the call-stacks of the two exceptions, i.e., the reference call-stack, and, instead of the full-stack of the other dynamic local checkpoint, one only the *delta stack* composed by the different delta stack frames[1]. When other dynamic local checkpoints that are part of the same composite debugging event are detected, a new delta stack is calculated using the reference call-stack as the baseline.

By employing composite debugging events with delta stacks, only one full dynamic local checkpoint needs to be serialized over the network. If

[1]For more details about the delta stack operations please refer to Section 4.2.2, particularly to Figures 4.9 and 4.10

call-stacks reference variables with the same values, that is often the case for parallel iterative Spark-like applications, they will be omitted resulting in lower network usage.

7.1.2 Domain-specific Stepping Operations

Once a debugging session is opened at the developer's machine, out-of-place debugging provides classical online features such as a view of the state of all the variables for each frame of the call-stack, the possibility to inspect and evaluate code in the debugged context, and operations to step into and over the execution of methods. Thus, developers can step into the application of a certain transformation and debug its execution.

Furthermore, as argued in Section 2.5, we also want to provide domain-specific debugging operations to enable debugging across the execution model of Spark-like applications. Namely, a Spark-like execution is modeled to execute a series of transformations followed by an action or a wide transformation that triggered them. Inspired by the stepping over a transformation of BigDebug [GIY+16], we devise a set of three domain-specific stepping operations targeted to Spark-like executions that work in combination with the classical stepping operations offered by online debuggers.

Those operations apply when debugging a particular execution of a transformation within the execution of an action or wide transformation. Below, we describe the three operations:

Step to next record. Steps to the next execution of the same transformation, i.e., to the next record.

Step to next transformation. Steps to the first execution of the next transformation (akin to BigDebug's step over).

Step to action result. Steps until the point in which the next action is applied (locally). At that point, the developer inspects the result to evaluate (i) if the execution finished correctly and (ii) the result of such an action.

7.1.3 A Relaxed Computational Model for Spark-like Applications

As mentioned in Section 2.4.5.2, not all errors are harmful: there are situations where ignoring certain application errors has little or no impact

in the final result of the application. For instance, many times developers spend hours solving errors related to dirty data [MLW+19], which could simply be ignored. This is the case, especially for many data science applications that allow for a certain degree of accuracy loss. For example, in a K-Means algorithm not all data is sensitive and a certain loss of accuracy is acceptable: Chippa et al. [CCRR13] show that a k-means algorithm may run up to 50x faster by giving up 5% of accuracy.

For the applications that do accept a certain degree of accuracy loss, allowing developers to instruct the framework to ignore a certain number of errors leads to a more scalable debugging solution where *only* relevant errors need to be manually debugged. In our approach, developers have control over *when* and *which* exceptions may be ignored by the runtime.

The simplest way to ignore errors is by revisiting the mechanisms of exception handling present in the underlying execution model with new methods to handle and deal with exceptions. If a developer knows where an application error might happen, that error can be ignored by adding an explicit exception handler. The challenge, however, is how to deal with unanticipated errors. This often requires manual creation of exception handlers (or exception handling techniques) scattered around the code base, especially if developers want to keep track of the ignored records. To tackle this problem, we build on the ideas of acceptability-oriented computing [Rin03], which envisions a failure model where errors are accepted automatically by the runtime, avoiding boilerplate exception handling code. This is also referred to as *relaxed computation* [CKMR12].

In particular, we propose a relaxed computation model for Spark-like applications in which developers can instruct the runtime to ignore errors up to a certain threshold, combined with the debugging support for debugging errors that were not ignored. The overall model presents three main features:

1. Developers can explicitly ignore errors happening in both actions and transformations on DDDs.

2. Developers can specify a threshold of the number of ignored exceptions in terms of the original dataset size. This determines the maximum loss in accuracy given by ignoring errors.

3. Developers have access to the ignored records and can debug their execution.

To incorporate the proposed relaxed failure model, we adapt how runtime exceptions are handled. In Big Data frameworks, exceptions across the parallel computation are usually caught by a global exception handler that reports them to the master, which eventually terminates the execution. To enable relaxed computations, the Big Data framework (i.e., Spa) is instructed to catch all exceptions with a *relaxed* exception handler. Those exceptions will be ignored locally, and the execution will be resumed from the next iteration of the same operation. When the number of ignored exceptions across the parallel computation goes over the given threshold, the default handler is invoked to report the error to the master. To enforce the ignoring threshold, each of the operations is initialized on the master with a counter, called the *exception counter*, that is incremented whenever a worker ignores an exception through the relaxed handler.

7.1.3.1 Enabling and using the relaxed failure model

We propose a relaxed failure model that can be activated both globally for all actions or locally to a particular action (and its related transformations in the pipeline). In our implementation in Spa, the global relaxed mode and its threshold are set by a developer through a call to Spa's API or from the UI, in a similar way as launching an application in "debug" mode in a classic IDE, as further detailed in Section 7.2.2. The local relaxed model, instead, is set programmatically for actions and wide transformations (and their pipeline of transformations). To this end, we extended Spa's API with a new action: `ignoreExceptions`. This action forces the ignoring of exception of its preceding pipeline and returns a new distributed data structure (DDD) on which operation can be applied within a relaxed exception handler. Spa also includes an `ignoreExceptions:` variant, where the ignore threshold is specified as a percentage of the size of the entire data set. By default, calling `ignoreExceptions` without parameters is equivalent to calling `ignoreExceptions:100`.

To illustrate our extension, consider Listing 7.1, which applies the local relaxed failure model to the running example on the word count application. Line 1 creates a DDD from a collection of 2001 numbers on which a map operation is applied. To enable ignoring of exceptions for the mapping transformation, the developer adds a call to `ignoreExceptions` before applying the `getCollection` action (in line 3). In this way, the

execution of that action returns an error-free result, that does not contain the result of 1/0.

```
1 data := framework distribute: (0 to: 2000).
2 mapped := data map: [:e | 1 / e ].
3 result := mapped ignoreExceptions getCollection.
4   >> {1 . 1/2 .... 1/2000}.
```

Listing 7.1: Enforcing the relaxed failure model on a particular execution.

As described earlier, when a particular exception is ignored, our approach filters out the failure-inducing record, so that the resulting data structure does not include it (or a transformation of such record). This, however, does not lead to a complete loss of information: a reference to the ignored record, i.e., its index in the data partition, is kept in the returned DDD. To retrieve more information about that record, three operations are made accessible to developers to (1) retrieve how many records were ignored, (2) obtain a collection with all records that were ignored, and (3) obtain a new DDD containing all failure-inducing records. Listing 7.2 shows these three operations in the context of the relaxed operation shown in Listing 7.1.

```
1 count := mapped ignoredCount.
2 ignored := mapped fetchIgnoredValues.
3 ignoredDDD := mapped createDDDFromIgnoredValues.
```

Listing 7.2: Retrieving information from an ignored execution.

7.1.3.2 Enhancing exception handling with ignore awareness

As mentioned in the introduction of this chapter, Spark-like applications will either terminate without errors or fail with a runtime exception after the framework unsuccessfully re-schedules the failed execution in other workers. Recall that we call *failure-inducing record* the record that is being processed when the exception occurs.

In Spark-like frameworks, developers can add an exception handler in the closures passed as arguments to the operations. In Spa, developers can also add their exception handler through a functional call to a certain pipelined transformation.

```
1  mapped := collectionDDD map: [:e | e / 0]
2  mapped handle: ZeroDivide with: [:error | 'infinite' ].
```

The above listing displays how to add an exception handler to a specific transformation, i.e., a map of a division by zero. While this is syntactic sugar for adding the exception handler to the closure execution, treating the addition of the exception handler as a functional call enables the introduction of a pipeline reification to guide the execution.

In Spa, exception handlers on transformations can be used regardless of the global failure mode, and always take priority on the selected failure model: if an exception is handled by a user-defined exception handler, it is not propagated to the default or relaxed handler. Through the pipeline reification, developers can decide to ignore particular exceptions in the exception handler, as shown in Listing 7.3.

Line 1 adds an exception handler to the `map:` transformation for a specific type of exception, i.e., `ZeroDivide`. The closure that is passed to the exception handler takes two parameters: the exception (stored in `error`) and an optional reification of the current execution (stored in `pipeline`). To instruct the runtime to ignore that particular error, the `ignoreError:` method is called within the exception handler on the pipeline reification. In line 2, the call to `getCollection` triggers the execution of the `map:`, and returns the resulting data structure. In the execution, the first element (0) throws an exception that is caught by the exception handler (in line 1) and ignored. As such, the `getCollection` executes without throwing the error, and the result (shown in line 3) contains each of the original elements as denominator except for the failure-inducing record (0) as if the ignore mode was active.

When an exception is ignored in a custom exception handler, the state of the relaxed handler will determine how the exception is processed: if the relaxed handler is active, it will handle the exception, increasing the

```
1  mapped handle: ZeroDivide with: [:error :pipeline | pipeline ignoreError:
     error].
2  result := mapped getCollection.
3    >> result = {1 . 1/2 ... 1/1999 . 1/2000}.
```

Listing 7.3: Ignoring an exception in the exception handler.

exception counter, and taking the ignored exception into account when checking the counter against the threshold; if the relaxed handler is not active, the exception is ignored without checking if the exception counter is lower than the threshold, since it is not set in the selected failure model of the action.

7.2 SpaDebug: an Out-of-place Debugger for Spark-like Applications

We prototype our solution in SpaDebug: an out-of-place debugger for Spark-like applications. SpaDebug is an extension of IDRA_{MR} and thus retains its code update capabilities and domain-specific debugging modes, i.e., the debugging on virtual partitions. It also shares with IDRA_{MR} the UI for displaying composite debugging events.

SpaDebug, however, introduces new optimizations (cf. Section 7.1) and new domain-specific stepping operations available when debugging an execution. Finally, it offers different execution modes to apply relaxed computations.

In what follows, we first detail how we used different stack-tailoring operations from Sarto to implement different features of SpaDebug. Then, we further describe how to activate the different execution modes in SpaDebug and detail how domain-specific stepping and relaxed computations were enabled in the debugger.

7.2.1 Sarto's Operations in SpaDebug

SpaDebug takes advantage of several operations of Sarto (cf. Chapter 4 to tailor the call-stack for debugging, particularly to (i) reduce the size of dynamic local checkpoints, (ii) identify similar debugging events, and reduce the size of composite debugging events, and (iii) enable classical and domain-specific stepping operations.

Reducing the size of dynamic local checkpoints. As IDRA_{MR}, SpaDebug uses the *stack cutting* operation to exclude framework frames from the call-stack, thus reducing the size of a dynamic local checkpoint. To do this, we mark as framework exit point all transformations and actions in the implementation of the DDD partition.

Identify and reduce the size of composite events. As IDRA_{MR}, SpaDebug also uses the *stack comparison* operation to determine whether two debugging events are similar. Furthermore, the *delta stack calculation* operation is used to further reduce the size of composite debugging events. Considering two similar call-stacks related to the same composite debugging events, the delta stack calculation extracts a delta stack that includes only the variables that change among the two similar call-stacks.

Enable classical and domain-specific stepping operations. Before debugging, the *delta stack application* is used to reconstruct the call-stack related to a particular execution. This ensures that all variables are set to the state they had at the moment the dynamic local checkpoint was taken. Furthermore, the *crafting a stack frame* operation is also used to insert in the debugged call stack a method representing a sequential execution of the different transformations that are being debugged. This is further discussed in Section 7.2.3.

7.2.2 Debugging Modes in SpaDebug

By introducing a relaxed failure model, we introduced a novel debugging mode for Spark-like programs in which errors can be systematically ignored. Overall, we provide three global execution modes for SpaDebug:

Debug. When selecting the default debug mode, SpaDebug will effectively use the default error handler, catching errors, reporting the failure to the master, and reporting dynamic local checkpoints to the debugger to enable debugging. Errors are visible shortly after in the UI and are debugged interactively as described earlier in this chapter.

Ignore. When selecting the ignore mode, SpaDebug will use globally the relaxed error handler, catching the errors and ignoring them without reporting a failure to the master. An ignoring threshold can be set, and it applies to each pipeline of transformations: if a distributed collection includes 100 elements, and the threshold is set to 30%, up to 30 failures will be ignored across the parallel execution of a particular action. In case the failures exceed the threshold, the following failures are handled using the default error handler, i.e., as if the debug mode was selected.

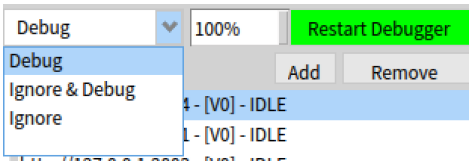


Figure 7.1: The dropdown for the selection of SpaDebug mode.

Ignore and Debug. The *"ignore and debug"* mode combines the relaxed execution with our debugging capabilities. In practice, when using the ignore and debug mode, a special exception handler is used that behaves like the relaxed handler in terms of how the exception is handled, but also performs a call to the default handler to generate a dynamic local checkpoint and report it to the developer's debugger. As for the ignore mode, a threshold can be set to limit the amount of ignored exceptions.

The developers can select the global SpaDebug mode from the Spa UI as shown in Figure 7.1. This is similar to how they would typically select to run a program in debug mode in a conventional IDE.

As already mentioned in section 7.1.3.1, the ignore mode can also be limited to certain action execution by calling `ignoreExceptions` (or `ignoreExceptions: ignoreThreshold`) before calling an action. Analogously, the ignore and debug mode can be activated on single action execution by calling `ignoreAndDebugExceptions` (or `ignoreAndDebugExceptions:ignoreThreshold`). Finally, ignoring exceptions can also be triggered directly in the exception handler as further discussed in Section 7.2.4.

7.2.3 Domain-specific Stepping Operations in SpaDebug

In this section, we describe our approach to correctly reconstruct a remote execution at the developer's machine for debugging, enabling the domain-specific stepping operations.

Let us re-introduce the election pools analyzer first described in Section 3.3. Listing 7.4 shows the implementation of the election polls analyzing application case in Spa, extracted from Listing 3.3.

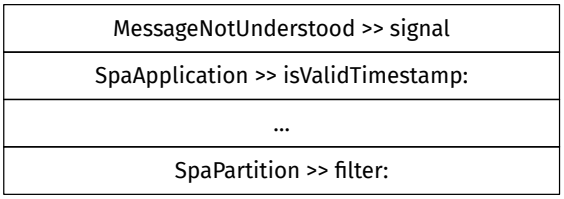


Figure 7.2: The instrumented call-stack when a MessageNotUnderstood error is signalled during a Spa execution.

```
1 parsed := raw map: [:line | line substrings: ','].
2 valid := parsed filter: [:array | (self isValidTimestamp: array third) and: [
      self isValidRegion: array first] ].
3 mapped := valid map: [:array| array second −> 1].
4 result := (mapped reduceByKey: [:value1 :value2 | value1 + value2])
      getCollection.
```

Listing 7.4: Part of the code of the vote counting application in Spa extracted from Listing 3.3.

In this code, line 1 parses the raw contents of a file, and the resulting collection is filtered (line 2) to extract only the records that have a valid timestamp and a valid region. Then each record is mapped to the number 1 in line 3 and reduced by key to count how many times each record has been voted in line 4.

When executing, Spa schedules the execution in two main tasks, similar to Spark's stages. The first task executes the first map, followed by the filter and the second map, and part of the reduceByKey. This is because, after reducing the closure of the reduceByKey locally to each partition in each worker, a grouping operation is called to group the data by key, thus shuffling data so that all key-value pairs of a certain key are in the same partition. At this point, the second task is executed to further reduce the data, and finally return the result to the developer to the call to getCollection.

Figure 7.2 displays the call-stack when an error happens during the filter, thus the second transformation of the first task. Since we cut the framework frames out of the call-stack, it now contains no information about what generated the filter below the bottom stack frame. To debug the full execution of a certain action or wide transformation, there should

be a method below the activation of `filter:`, that makes the sequential call to all the different transformations, as the one shown in Listing 7.5.

```
1 SpaExecution>>execute
2 (((partition map: [:line | line substrings: ','])
3   filter: [:array | (self isValidTimestamp: array third) and: [self
      isValidRegion: array first] ])
4   map: [:array| array second -> 1]) reduceByKey.
```

Listing 7.5: An example of the reconstructed activation method for debugging the execution of Listing 7.4 locally.

Through the *crafting of a stack frame* operation of Sarto, we add a method into the call-stack as the one presented in Listing 7.5. To construct it, we first extract the lineage information about the debugged action/wide transformation and then add each call sequentially to the crafted method. We then add the method at the bottom of the call-stack, setting the PC to the right transformation call.

7.2.3.1 The Domain-specific Stepping Operations in Action

The crafted method enables the domain-specific stepping operations of SpaDebug. To showcase the different operations, let us consider the practical example of debugging a breakpointed execution of the election polls analyzing application. Figure 7.3 shows the view of the debugger halted in the execution of the `filter:`. You can see that several stepping operations are offered to the developer. From the left, the first 5 are the classical stepping operations offered by the Pharo debugger. The three following ones are the domain-specific stepping operations that we introduced with SpaDebug.

In this example, the execution is halted in the `filter:`. Stepping to the *next element* will step to the same `filter:`, but applied to the next record of the available partition. Stepping to the *next transformation* will instead step to the following execution of `map:` on the first record of the local partition. Finally, stepping to the *action result* will inspect the results of the action or wide transformation, `reduceByKey` in this case, allowing the developer to further analyze them. As for the whole debugging session, the execution of these domain-specific debugging operations is local to the developer machine and does not require network communication with the remote execution of Spa (i.e., the cluster).

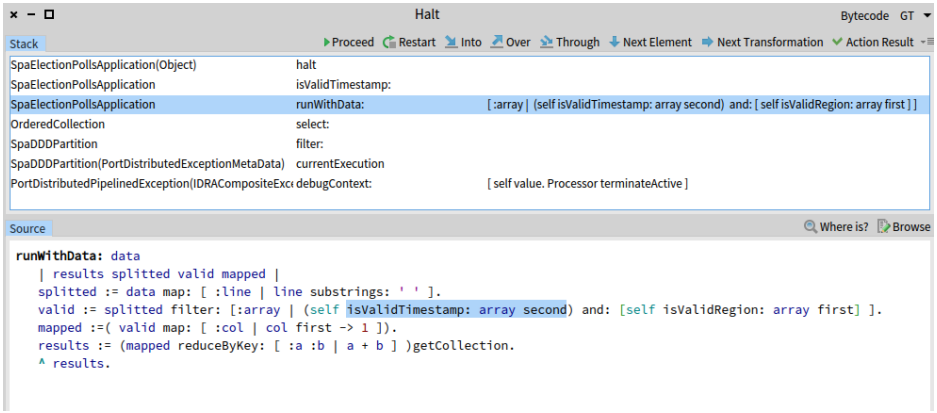


Figure 7.3: The view of the debugger opened on a failing filter in the election polls analyzing application.

7.2.4 Integrating the Relaxed Computational Model in SpaDebug

In this section, we detail how we enable relaxed computations through a relaxed exception handler in SpaDebug. Particularly, we focus on how failure-inducing records are treated and how the ignoring threshold is enforced.

What happens to the ignored failure-inducing record? When a particular exception is ignored, the runtime filters out the failure-inducing record, so that the resulting data structure does not include it (or its transformations). As an example, let us consider again the vote counting application (cf. Section 3.5.1). Its code includes a `map:`, a `filter:`, and another `map:` followed by a `reduceByKey:`.

In the example, an error happens during the filter, within the call of `isValidTimestamp:`, as depicted in Figure 7.4. When that error is ignored, the handler returns a placeholder (represented as `##`) to the first method after the API call, so the call of `select:`, a method of the Pharo Collection API used during `filter:`. If any record was ignored, a flag will instruct the worker to remove placeholders from the collection before feeding the intermediate results to the following transformation, a `map:` in this case.

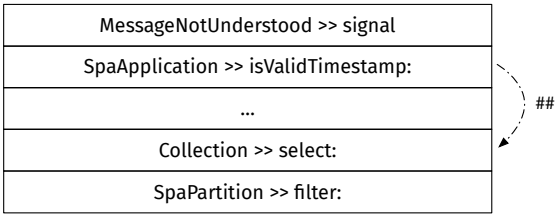


Figure 7.4: The instrumented call-stack when a MessageNotUnderstood error is signalled during a Spa execution, and before the error is ignored.

Enforcing the ignoring threshold To enforce the ignoring threshold, each of the operations is initialized on the master with a counter. This counter is replicated in each of the workers and is incremented whenever the relaxed exception handler ignores an exception. The worker sends the increment of the counter to the master, which then returns the latest updated value of the counter. If the returned counter is higher than the threshold, then the relaxed handler simply forwards the exception to the default handler, which will generate a dynamic local checkpoint to enable debugging.

7.2.4.1 Notes on Related Work

The ideas of acceptability-oriented computing and relaxed computations have been already explored in different contexts. For example, Rigger et al. [RPM18] have used the concept of relaxed programs to prevent buffer overflows in C, avoiding aborting the program on a buffer overflow, through implementing a failure-oblivious system via a recovery logic. In another example, Zhang and Monperrus [ZM19] also use the concept of failure-oblivious system to improve the reliability of Java programs, by testing the execution of a program, and, if a failure is found, injecting empty exception handlers to handle them; if the addition of the handler does not generate a failure in all the application's test cases, then it is kept and signaled to the developer. To the best of our knowledge, this is the first work that explores them in Big Data parallel computations.

7.3 Conclusion

In this chapter, we have described our practical out-of-place debugging approach for Spark-like applications. Particularly, we described how *dynamic local checkpoints* extend *debugging events* with more contextual information of the execution and with two crucial optimizations for reducing their size: *partition windowing* and *composite debugging events with delta stacks*. We then introduced the domain-specific stepping operations tailored to Spark-like applications and we described a relaxed computational model that extends Spa's execution model for ignoring errors. The performance and usability of our proposed approach are validated in the next chapter.

Chapter 8

Validation

In this chapter, we validate our debugging approach for Big Data applications. The goal of this validation is two-fold. First, we assess the scalability and the overall performance of the different features of our debugging approach; second, we assess its usability through an experimental user study.

We perform the benchmarks across different applications written in the Spark-like model, ranging from classical parallel distributed applications such as wordcount and grep, to popular data analysis algorithms such as K-Means.

We then present the details and results of an experimental user study in which 17 participants performed two debugging assignments over two data analysis applications, i.e., K-Means and decision tree learning, using SpaDebug and a reproduction of BigDebug, a state of the Art debugger for Spark (cf. Section 2.4.5.3).

This validation does not only validate the features of our debugger for Spark-like applications but also of the underlying debugging layer shared by both the debuggers proposed in this dissertation.

8.1 Performance Evaluation

In this section we assess the scalability and performance of our debugging approach by answering five research questions:

PQ 1. How does our debugging approach scale to Big Data?

PQ 2. What is the impact in reducing the size of the debugging session of the two optimizations, i.e., partition windowing and composite debugging events with delta stacks?

PQ 3. How much time does out-of-place debugging save in comparison to replay and checkpoint-based debugging?

PQ 4. What is the overhead of the relaxed computational model?

PQ 5. How does the relaxed computational model scale to a big number of exceptions?

To answer these questions, we conducted several performance benchmarks across a benchmark suite including typical Big Data and data analysis applications. In what follows, we first describe the benchmark suite, then the setup of our environment, and finally we present the experiments and their results.

8.1.1 The Benchmark Suite

We run our benchmarks on three different Big Data applications, commonly used to test the performance of Big Data frameworks [WZL+14]: distributed grep, wordcount, and K-Means. In what follows, we describe the three different applications.

8.1.1.1 The Distributed Grep Application

The purpose of the distributed grep is to analyze one or multiple large text files, extracting from the files only the lines that include a certain string.

```
1 SpaGrepApplication >> runWithData: fileDDD
2   ^ (fileDDD filter: [ :line | line includesSubstring: parameter ]) execute.
```

Listing 8.1: The code of the Distributed Grep application.

Listing 8.1 shows the application code. The application expects a DDD representing a file as parameter. Line 1 defines the method, and line 2 applies a filter that accepts only the lines that include a certain substring, `parameter`, that is set as application parameter before running it. `execute` is called to trigger the execution.

8.1.1.2 The Wordcount Application

Wordcount is another classical application to test Big Data frameworks. As the name suggests, the goal of the application is to count how many times each word is repeated in one or more files.

```
1 SpaWordCountApplication >> runWithData: fileDDD
2   |mapped|
3   mapped := (fileDDD flatMap: [:e | e substrings: ' ' ]) map: [ :e | e -> 1
      ].
4   ^ (mapped reduceByKey: [ :a : b | a + b ]) execute
```

Listing 8.2: The code of the Wordcount application.

Each line of the file is flat-mapped to split by the character space, thus extracting each word of the file. Then each word is mapped in a key-value pair to the number 1 using the `map` transformation. The last line reduces by key the mapped data, summing the values associated with each key, thus returning a collection in which each word is mapped to the number of times it was repeated.

8.1.1.3 The Twitter K-Means Application

The purpose of the Twitter K-Means application is to understand which hashtags are more popular based on the number of likes tweets receive. K-Means is also a representative application to evaluate our relaxed computational model, as a lower accuracy does not heavily impact its results [CCRR13].

Listing 8.3 shows the main method of the application executed by Spa. First, line 2 parses the tweets, and line 3 makes pairs for all the tweets by associating each of the hashtags of a tweet with the number of likes the tweet received. Line 4 a `reduceByKey:` is applied to sum, for each hashtag, the number of likes it got. Lines 5,6, and 7 send the message `top:withBlock:` which extracts the trending N hashtags, i.e., the N hashtags with the most total likes. For each hashtag, lines 8 and 9 extract from the pairDDD data structure the values that have that particular hashtag. This is returned as a Dictionary. Lines 11 to 21 correspond to the iterative part of the K-Means algorithm called on each of the trending hashtags (and the subset of tweets that include that hashtag). In a nutshell, the K-Means algorithm runs different iterations to find the optimal clusters of tweets, returning a result for each of the hashtags.

```
1  K−MeansClusteringTweetsExperiment>>runWithData: data
2    parsed := self parse: data.
3    pairDDD := self makePair: (parsed filter: [:e | e isNotNil]).
4    scores := pairDDD reduceByKey: [ :v1 :v2 | v1 + v2 ].
5    trending := scores
6      top: nTags
7      withBlock: [ :v1 :v2 | v1 value > v2 value ].
8    likes := (trending asDictionary keys
9      collect:
10       [ :hashtag | hashtag −> ((pairDDD filter: [ :kv | kv key = hashtag
       ]) map: #value) ])
11     asDictionary.
12   results := Dictionary new.
13   trending
14     doWithIndex: [ :t :idx |
15       rdd := (likes at: t key) execute.
16       centroids := rdd takeSample: nClusters.
17       K−MeansResult := K−Means
18         runK−MeansOn: centroids
19         withData: rdd
20         maxIterations: maxIterations
21         treshold: treshold.
22       rdd deallocate.
23       results at: t put: K−MeansResult.].
24   ^ results
```

Listing 8.3: Main method of the Twitter K-Means application.

Figure 8.1 shows the different stages in the twitter K-Means application that conceptually belong together and that are grouped in pipelined operations [1]. These stages are (i) parsing the JSON dataset in `Tweet` objects (ii) extracting the hashtags and related likes and grouping the tweets by hashtags, and (iii) running the K-Means algorithm. Note that the K-Means stage will be iterated several times for each of the different hashtags, as indicated by the arrow in the figure.

More details about the application are described in the user study assignment material about this application, available at Appendix A. Particularly, the full code of the application is listed in Appendix A.3.

[1] To simplify the figure, the different transformations and actions happening during the iterative part of the K-Means algorithm have been omitted.

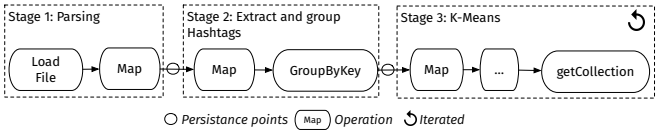


Figure 8.1: Overview of the different stages of the K-Means application.

8.1.1.4 Dataset

We run the applications on different subsets of a dataset containing roughly 100 GB of raw tweets, coming from a recording of a live Twitter stream[2]. In this dataset, tweets are represented using JSON, as detailed by the Twitter API[3]. The great majority of the tweets in the dataset are valid (i.e they have an id, a text, etc.) and are thus correctly parsed by the applications. However, since the dataset was recorded from a live stream, it also presents around 17% of tweets that were malformed or miss information that causes the application to fail if they are not filtered out. Deleted tweets are not valid for our analyses as they do not contain text and thus are excluded in the analysis by our parser. In our experiments, when we say that we inject failures it means that we change the parser to leave some of those tweets in the parsed dataset as `nil` values, which will let the application raise an exception later during the execution.

8.1.2 Setup

We run our experiments on a cluster composed of one *root* node and ten identical *slave* nodes. Each node presents an Intel Xeon CPU E3-1240 @ 3.50GHz, 32 GB of RAM, and 200 GB of SSD Storage. Nodes are connected via a 1 Gb/s local network. Depending on the benchmark, we use a slightly different configuration using different amounts of slave nodes. When this happens, we specify it in the benchmark.

For all the benchmarks, we deploy Port on the cluster using Hadoop Yarn, and we use 1 single-core master, and, depending on the benchmarks, several single-core workers. Recall that Hadoop Yarn takes care of the

[2]Particularly, the dataset used in our experiments includes random tweets recorded on the 30th July 2017

[3]`https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json`

allocation of the master and workers on the cluster (cf. Section 3.6. The cluster runs Pharo 8.0.0 (x64) on a Pharo 8.3.0 Headless VM.

We control the execution and perform debugging from a 2017 MacBook Pro, running an Intel Core i7-7567U @ 3.5GHz CPU, 16 GB of RAM, and 500 GB of SSD storage. This machine uses SSH tunneling to communicate to the cluster. On this machine, we run the same version of Pharo as on the cluster but including UI.

8.1.3 PQ 1: How does our debugging approach scale to Big Data?

To validate the scalability of our debugging approach, we designed an experiment to measure the time to handle an exception when increasing the amount of analyzed data, as well as when increasing the number of exceptions that happen in the parallel execution. To do so, we inject failures in the Twitter K-Means Application described in section 8.1.1.3 and measure the amount of time until Spa reports a failure to the debugger, for increasing amounts of data and parallel exceptions.

For this benchmark, we inject a failure-inducing record (i.e., `nil`) in each partition of the RRD before executing stage 2, i.e., all workers fail just after the tweets are parsed. We then measure the time needed to handle an exception and generate an online debugging session, by measuring in the master how much time passed since the computation has started when the debugging session is sent for debugging. We run the same experiment for datasets of different sizes, from 5 GB up to 50 GB. We run the execution 20 times for each of the portions of the dataset, and each number of parallel exceptions. We report averages ± standard error.

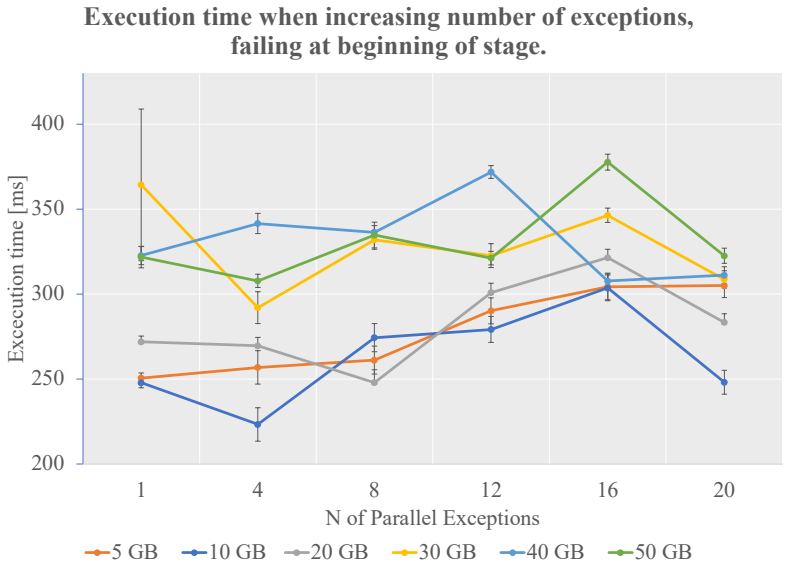


Figure 8.2: Execution time of stage 2 when increasing the number of exceptions happening in parallel, for different sizes of the dataset. Error bars show the standard error.

Results Figure 8.2 shows the average execution time of the failing stage, including the time to generate a debugging session, for each different size of the dataset (shown in different colors). The results show that the execution time of the failing stage is relatively dependent on the amount of analyzed data and of parallel exceptions. When analyzing five to twenty GB of data (gray, orange, and blue line) the execution time is generally lower than when analyzing more data. This difference, however, is always lower than 200 milliseconds, which we believe to be an acceptable amount. When increasing the number of parallel exceptions, however, the execution time generally grows for low amounts of data and stays stable for a higher amount. The differences across the same data size, however, range in around 50 milliseconds, which we again consider to be an acceptable amount.

With these results, we conclude that our debugging approach scales, for what concerns handling an error to create a (composite) debugging session, to both an increasing number of errors and of parallel exceptions. These results also include the overall impact of our debugging optimiza-

tions, i.e., composite debugging events and partition windowing, that we further investigate in the next section.

8.1.4 PQ 2: What is the impact in reducing the size of the debugging session of the two optimizations?

In this section we assess the impact of the two main optimizations introduced in our out-of-place debugger for Spark-like applications (cf. Section 7.1, i.e., partition windowing and delta stacks.

To assess the impact of the two optimizations, we designed two benchmarks using the K-Means application (cf. Section 8.1.1.3). First, we measure the impact of partition windowing by changing the partition windowing size parameter.

Second, we analyze the size of the dynamic local checkpoints when delta stack calculation is enabled and compare it to the size of the dynamic local checkpoints without the use of delta stacks.

Setup. Similar to the scalability benchmark, we let the application fail at the beginning of stage 2, i.e., after the parsing. Since the scalability benchmark showed that the amount of data does not affect the performance of the debugger, we use a fixed amount of data, i.e., 40GB.

8.1.4.1 Impact of Partition Windowing

For this benchmark, we increase the size of the window from 1 (thus a partition including only one record, i.e., the event-inducing record), to 256, increasing by the powers of four (i.e., 1, 4, 16, 6 4, 256). As detailed in Section 7.1.1.1, the window size parameter sets how many records of the partition of the failure inducing record to include in the dynamic local checkpoint.

The total size of the checkpoint is measured at the debugger manager when receiving a dynamic local checkpoint as part of a composite debugging event. We measure the serialized size of the dynamic local checkpoints. All 20 workers fail, hence producing 20 dynamic local checkpoints. For each window size, we repeat the experiment 10 times, and express the average across all the iterations, i.e., 200 single dynamic local checkpoints for each window size.

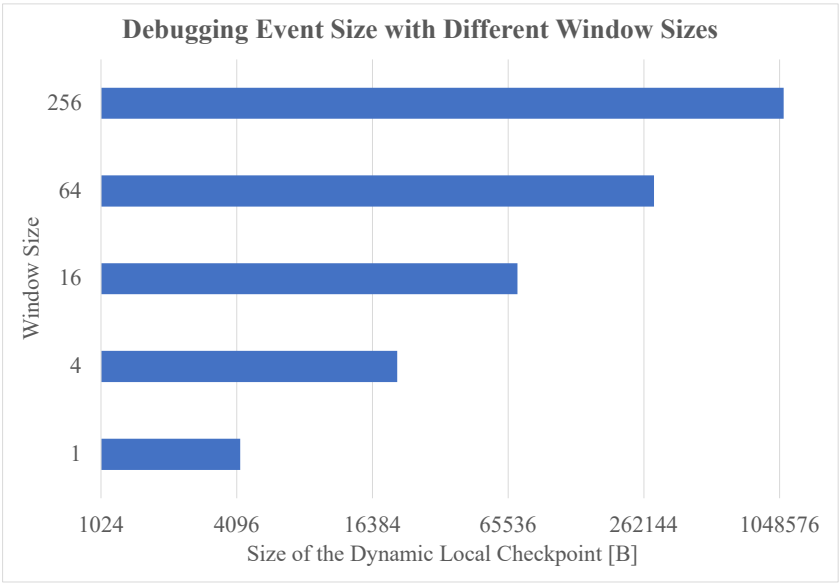


Figure 8.3: Average dynamic local checkpoint size when increasing the size of the windowed partition.

Figure 8.3 shows the result of the benchmark. The Y and X-axis display the different partition sizes and the size of dynamic local checkpoint expressed in bytes, respectively. Both scales are logarithmic.

We do not display standard error, since the differences across the different iterations were measured to be negligible.

As expected, the size of a dynamic local checkpoint grows linearly to the size of the window, i.e., including more data causes bigger dynamic local checkpoints, hence higher network usage.

8.1.4.2 Impact of Delta Stack Calculation

In this benchmark, we assess the impact of partition windowing when toggling the delta stack optimization and measuring the size of the dynamic local checkpoints for different window sizes.When delta stacks are disabled, each dynamic local checkpoint that is part of a composite debugging event carries the full call-stack information, excluding the framework stack frames (cf. Section 6.2.1). When delta stacks are enabled, a dynamic local checkpoint only includes the calculated delta-stack compared to the first dynamic local checkpoints in the debugging event. Recall that delta

stacks are calculated using the delta-stack calculation operation defined in Sarto (cf. Chapter 4).

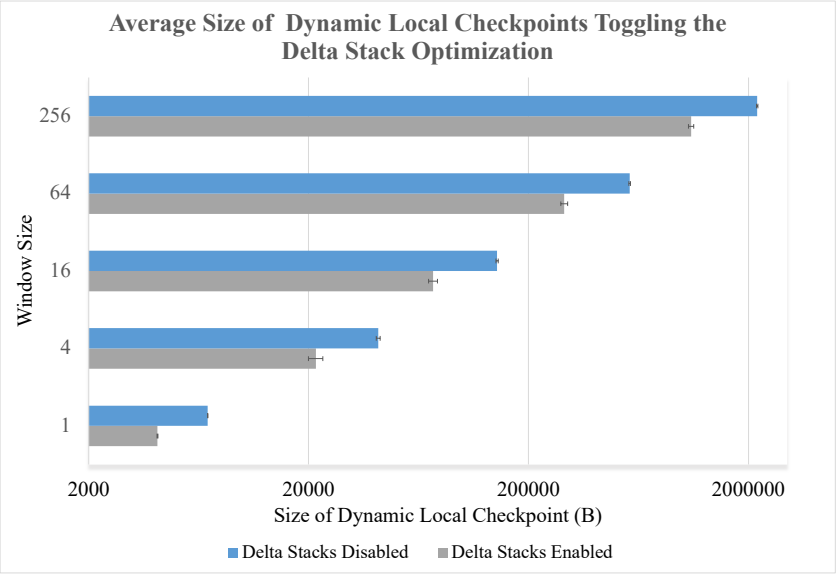


Figure 8.4: Average dynamic local checkpoint size when increasing the size of the windowed partition and toggling the delta stacks optimization.

Figure 8.4 shows the results of our experiment. As for the previous experiment, we run the benchmark 10 times, on composite debugging events with 20 dynamic local checkpoints, 19 of which contain a delta. We report the results as the average across the different iterations, i.e. with 190 samples, and standard error. The horizontal axis scale is logarithmic. Results show that the stack size grows linearly to the size of the partition independently of the usage of delta stacks. The size of the dynamic local checkpoint, however, is drastically lower when using delta stacks: when the partition size is one, the delta stack is 59% the size of the original stack. This number gradually lowers to 50.1% when the window size is 256, sending around 1 MB of data against the 2 of the full dynamic local checkpoint.

To conclude, these two optimizations make our debugging approach practical. In fact, they make it possible to generate compact dynamic local checkpoints, that lead to a fully reconstructed execution debuggable locally.

Dataset Size	Stage 1	Stage 2_{Fail}
5 GB	46 s	< 400 ms
50 GB	448 s	< 400 ms
20 GB*	54 s*	< 400 ms

Table 8.1: Average stage execution times depending on data size. The 20 GB indication of Stage 1 only includes the file read.

8.1.5 PQ 3: How much time does online debugging save in comparison to replay and checkpoint-based debugging?

In this section, we compare the time to open a debugging session of our approach with both replay and checkpoint-based debugging approaches.

Comparison to replay debugging To compare our approach to replay debugging, we analyze the execution time of the KMeans application in the first two stages, injecting a failure at the beginning of stage 2, as in the previous experiments. Table 8.1 in the first two lines shows the execution time of the different stages for the 5 and 50 GB files. With a replay debugger, an error at the beginning of the second stage requires to replay the whole execution from stage 1. The execution time of stage 1 in the previous experiment amounts to 46 seconds for the 5 GB file and up to 448 seconds for the 50 GB file. These results show that generating a dynamic local checkpoint is convenient in terms of runtime, since running again stage 1 takes 100 times more for the 5 GB file and 1000 times more for the 50 GB file.

Comparison to checkpoint-based debugging When comparing to checkpoint-based debugging, the result of the previous experiment in which the error happens at the beginning of stage 2 represents their best scenario if the developer persisted after stage 1, i.e., after loading the data. In this case, a checkpoint-based debugger does not need to replay any execution. However, this quickly changes when an error occurs during or at the end of a long stage, as explicit checkpoints are normally not inserted by developers in the middle of a stage for performance reasons. In those cases, a checkpoint-based debugger would have to replay part of the stage, which

leads to a more realistic comparison. We thus run an experiment in which we inject the failure in the middle of stage 1 by removing an if-test for null.

In this experiment, each worker reads a part of the dataset and immediately starts parsing tweets and iterates the execution 20 times. Since the previous benchmark showed that the size of the file does not impact the time to create a dynamic local checkpoint, we use a 20 GB dataset.

Table 8.1 in its last row shows that reading the file takes on average 54 seconds for the 20 GB dataset. The execution then fails, on average, less than 500 milliseconds after starting the actual parsing and the execution terminates after an average of 55 seconds of total execution time. When using a checkpoint-based solution, this time needs to be replayed to get to the error, since there is no persisting operation between the reading and the parsing. In comparison, as shown in Figure 8.2, the overhead of getting a debugging session with our approach amounts to hundreds of milliseconds.

From these experiments, we conclude that our approach provides *faster* access to a debugging session by avoiding replay operations that both replay and, partially, checkpoint-based debugging approaches for Big Data rely on.

8.1.6 PQ 4: What is the overhead of the relaxed computational model?

We assess the overhead of ignoring exceptions by turning the "ignore" mode on and letting the applications complete their execution without errors, thus measuring the overhead of the infrastructure. In this way, we simulate a setup in which the developer correctly filtered the initial dataset, but still leaves the "ignore" or "ignore and debug" modes on, in case something was missed in the data-cleaning process. We run this benchmark on the three applications using three portions of the dataset of increasing size: 15GB, 30GB, and 45 GB. We use as a baseline the execution when the ignoring and debugging support is switched off and compare it to the execution with it turned on. Each application is executed at least 10 times, and a maximum of 25 times, depending on run-time.

App	$T_{baseline}$	T_{ignore}	Δ (%)
Grep 15G	0,77 ± *0,35*	0,76 ± *0,30*	**-0,60**
Grep 30G	1,09 ± *0,13*	1,16 ± *0,28*	**6,52**
Grep 45G	1,54 ± *0,15*	1,60 ± *0,11*	**3,92**
WC 15G	82,96 ± *5,42*	83,99 ± *3,56*	**1,24**
WC 30G	170,38 ± *8,98*	172,86 ± *10,55*	**1,46**
WC 45G	446,56 ± *18,90*	433,38 ± *22,72*	**-2,95**
KM 15G	171,15 ± *1,08*	180,11 ± *0,69*	**5,24**
KM 30G	263,44 ± *0,84*	273,91 ± *1,06*	**3,97**
KM 45G	385,62 ± *0,93*	392,44 ± *1,43*	**1,77**

Figure 8.5: Runtime (in seconds) and overhead of running Grep (G), WordCount (WC), and K-Means (KM) applications with and without ignore mode active.

Results Figure 8.5 displays the runtime of the different applications for different sizes of the datasets, with the ignore mode turned off ($T_{baseline}$) and on turned on (T_{ignore}). We report the average time of the multiple iterations with the standard error. The last column presents the overhead of the ignore mode in terms of percentage of ($T_{baseline}$).

The table shows that the overhead varies between -2,95% to +6,52%. By further analyzing each application, we observe that Grep, the fastest benchmark, shows a big variation among the different data sizes, starting negative, then growing to 6%, and then lowering to 3,9%. The confidence intervals, however, overlap, which makes us conclude that there is no significant performance degradation. This is probably caused by the short duration of the benchmark. On the other hand, we observed a more consistent overhead pattern in both WordCount and K-Means: the overhead seems to lower when increasing the amount of data.

To conclude, this benchmark showed that the activation of the ignore mode always introduces some overhead, that decreases as the execution time increases. We attribute this overhead to the initialization of the replicated counter that counts the possible amounts of exceptions for each action, since, especially in an iterative algorithm as K-Means, happens hundreds of times during one execution.

8.1.7 PQ 5: How does the relaxed computational model scale to a big number of exceptions?

We now assess the performance impact of the relaxed computational model when exceptions raised during the execution are ignored. In this benchmark, we use the first phase of the K-Means application, in which the tweets are parsed and then grouped by their hashtag. We gradually leave invalid tweets in the dataset to cause an exception when extracting the hashtags. It has to be noted that we did not inject more invalid tweets than the ones already present in the dataset.

We distribute a fixed portion of the dataset and filter out all the failure-inducing records, to then inject them again in a controlled amount and proportionally across the different partitions. We test two different portions of the dataset: the first 5 GB (Dataset 1), and the first 20 GB (Dataset 2). Since we had 17% of failure-inducing records in the original dataset, we set the ignoring threshold to 50%, so we are sure that all of the errors are ignored. It is not set to 100% because in our implementation this disables the increments of the exception counters, since all of the exceptions need to be ignored anyway, thus lowering the effective time to ignore an exception and hampering the results of this benchmark. We test the K-Means application and measure the total time to completion.

Dataset 1 contains 996.576 records that represent valid non-deleted tweets, and we gradually inject at every iteration 20.000 failures, distributed amongst all the workers so that each of the workers will have to ignore the same number of exceptions. Similarly, dataset 2 contains 4.003.315 valid non-deleted tweets, and we inject at every iteration 80.000 failures (corresponding again roughly to 2% of the original records).

Results The boxplot in Figure 8.6 displays our results for dataset 1, using 5 GB of data. More concretely, t shows the amount of time it takes to parse and group tweets with the K-Means application while injecting an increasing number of failure-inducing records. We repeat each measurement 100 times and we report average times in milliseconds. The results show that the runtime increases linearly with the number of ignored exceptions. Particularly, analyzing the data we calculated that ignoring exceptions for this example costs on average between 6 and 10 microseconds for each exception.

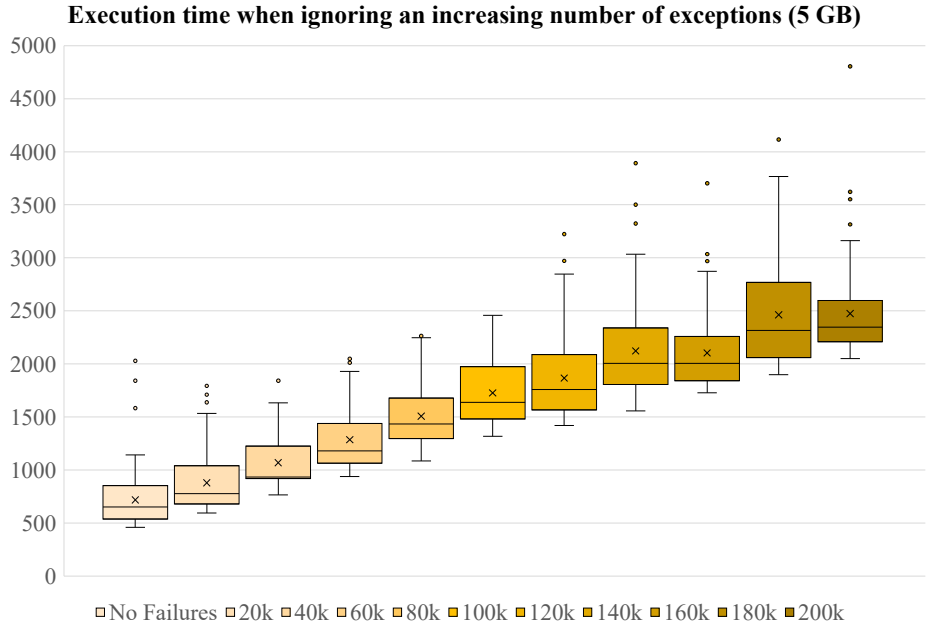


Figure 8.6: Run-time when ignoring an increasing number of exceptions on dataset 1 (5 GB). The color gradient represents the number of exceptions that where ignored, also visible in the legend below.

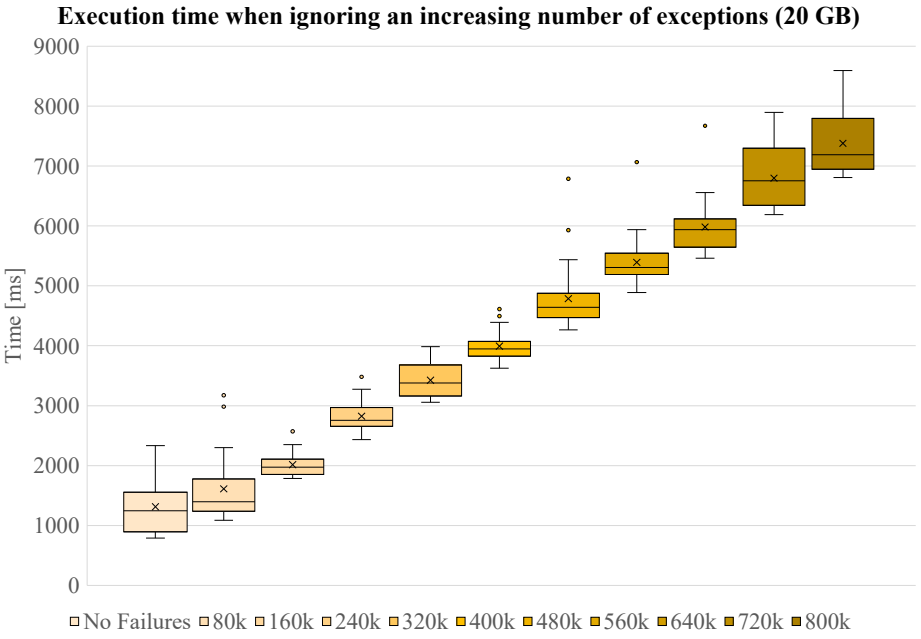


Figure 8.7: Run-time when ignoring an increasing number of exceptions on dataset 2 (20 GB). The color gradient represents the number of exceptions that where ignored, also visible in the legend below.

Figure 8.7 displays the results for the same experiment on dataset 2 (20 GB). Every measurement was repeated 50 times and we report the average time in milliseconds. The result of this bigger dataset confirms the linear increase in runtime when ignoring exception with a slightly lower amount of average time needed to ignore each exception: between 3 and 7 microseconds. This is because updates of the exception counter are sampled, i.e., the replicated counter is only updated when it is incremented by a certain amount about the size of the data. This leads to less frequent updates for bigger datasets, thus producing slightly less network communication for each ignored exception.

8.1.8 Discussion

From these benchmarks, we conclude that our debugging approach is practical and scalable based on the following results:

- The impact on the execution time of our debugging approach remains stable when increasing the amount of analyzed data and parallel exceptions.
- The impact of partition windowing leads to lower sized dynamic local checkpoints linearly to the size of the window, and that delta stacks produce dynamic local checkpoints up to half the size of the original ones.
- The relaxed computational model introduces negligible overhead when the ignore mode is on and there are no errors to be ignored
- Ignoring exceptions impacts the execution time linearly to the number of exceptions that are ignored.

8.2 User Study

In this section, we assess the usability of SpaDebug in a study with 17 subjects that tested SpaDebug and a reimplementation of the closest related work, namely BigDebug [GIY+16], on top of Spa[4]. The goal of the

[4]In this section, Spa refers to the execution framework described in Section 3.5, SpaDebug refers to our debugger for Spark-like applications, and BigDebug refers to the reimplementation of BigDebug [GIY+16] in Spa further detailed in Section 8.2.4.4.

study is to assess whether our debugging solution has an impact on the time to find bugs and on the participants' perception of the features of our debugging solution when compared to the state of the Art. We first detail the design of the user study, then its methodology, and finally the results.

8.2.1 User Study Design

Since the goal of this study is to assess both the impact on time to find bugs and the perception the participants have of the features of our debugging solution, we employ a *mixed method experimental design* [Chr15]. This method entails both a quantitative and a qualitative analysis.

Quantitative Analysis. In the quantitative study, we aim to assess experimentally if there is a cause-effect relationship between the debugger that is used (i.e., the independent variable) and the time to find the bug (i.e., the dependent variable).

Experimental research designs can be of three types: Weak, Quasi, or Strong [Chr15]. Their type is defined by the ability of isolating the effects of the independent variable on the dependent variable. Since we can isolate the effect of the independent variable (i.e., which debugger is used), thus maximizing the internal validity and reducing the external threats to validity, we employ a *strong experimental design*. During the study, we administer two experimental conditions:

1. Solve a debugging assignment using SpaDebug
2. Solve a debugging assignment using BigDebug

Strong experimental designs can be of different types, i.e., within participants, between participants, mixed, and factorial. For this study, we employ a *within participants design* [Chr15, CS63, SCC02] in which all participants are exposed to the two experimental conditions, and fill in a post-test after being exposed to each of the conditions. External threats are limited using a random assignment of participants in two different groups, in which the independent variables are introduced and evaluated in a different order. We further randomize across the two groups the order in which the participants are exposed to the two experimental conditions: the first group is first exposed to debugging with SpaDebug and later with

BigDebug. The second group is exposed to the debuggers in the opposite order. Employing a within-participants design allows us to let every participant test both of the debugging solutions, thus maximizing the number of data points while having a limited number of participants.

Qualitative Analysis. The qualitative part of the study consists of the analysis of the answers to specific questions. These include questions about the debugging experience, evaluation of different features of each debugger, and comparison across the features of the two debuggers. We collect answers to a questionnaire as part of the post-test of the study, which is filled in by the participants after they end their assignment with a particular experimental condition.

Overall, the study is designed so to answer four research questions:

UQ 1. Does the debugger impact the time to find and solve the first bug?

UQ 2. How was the debugging experience evaluated?

UQ 3. Does the debugger influence the number of re-deployments?

UQ 4. How were the features of SpaDebug perceived?

8.2.2 User Study Methodology

Throughout the study, participants have to solve two debugging assignments, always in the same order for both groups: the first one is to debug the Twitter K-Means Application, already described in Section 8.1.1.3, and the second one is to debug the Amazon ID3 Application described more in detail later in Section 8.2.2.2. Each application presents two bugs: the first bug is related to the formatting of the input data; the second bug is related to a logical error in the application, i.e., the final result is not correct.

8.2.2.1 Experimental Setup

We conduct the study with 17 volunteer subjects (9 master students and 8 researchers), randomly split into two different groups. Each participant executes the two assignments with both debuggers but in a different order. The master students had followed (and passed) at least two master-level courses in which they had to program one project in Pharo Smalltalk and

one in Spark. All researchers had experience using Pharo Smalltalk and knowledge of Spark.

Due to COVID-19 restrictions, we run the study in timed sessions in a physical room at the university campus or a virtual one over Zoom. The study, in both online and on-campus versions, has the same structure. First, participants attended a 15-minute presentation and hands-on demo about programming with Spa (cf. Chapter 3) without debugging. Before each assignment, they attended a 20-minute presentation about the debugger they will use and a hands-on demo about debugging an application. At this point, the participants were handed the material and had 45 minutes to complete each assignment. After 45 minutes we stopped the participants from further completing their assignment. After the first assignment, they filled in the first part of the post-test that includes general questions about their experience in debugging and developing software and questions about the debugger they used. Then, after a small break, we repeated the same process for the second experiment, i.e., participants attended a presentation and hands-on demo of the second debugger, performed the second assignment within 45 minutes, and filled in the post-test.

Throughout the study, the host of the experiment is always present in the (campus or virtual) room, answering questions about the debuggers in private, as well as helping with technical issues. All participants received monetary compensation of 10€ at the end of the study.

Group Composition As mentioned before, the 17 participants are divided into two groups, called group X and group Y. Since roughly half of the participants joined online and the other half joined physically, we made sure to have a balance across the two randomly assigned groups. Particularly, half of the online participants were randomly assigned to group X and the other half to group Y. The same randomization was used to split into the two groups the participants that joined the study physically.

Before the study, and after they were already randomly split into two groups, we asked the participants for a time slot that would suit them to do the study. We had several slots for the on-campus participants limited by the availability of the university's computer rooms. By crossing the availability of the on-campus participants with their slots selection, we

created subgroups of X and Y, namely X.1, X.2, Y.1, and Y.2, that would contain participants of group X and Y that had a compatible slot selection.

Similarly, we created groups X.3, X.4, Y.3, and Y.4 for the online participants. The creation of these sub-groups allowed us to have manageable-sized groups (max 3 people), that were required by the university for the use of the physical rooms, due to COVID-19 restrictions.

There is no difference, however, in the treatment given across a group in the different subgroups. Since all presentations were the same and the order of the experimental conditions as well, for our analysis we consider only two groups: X and Y.

The Debugging Tools Participants employed the same IDE, i.e., the Pharo Smalltalk environment including Spa, but they interacted with the two different front-end debuggers corresponding to SpaDebug and a reconstructed version of Big Debug on Spa.

Since Pharo is image-based, we provided participants with a Pharo image already set up for the experiment, with an open Spa UI and easy access to predefined code to run the experiments. This Pharo image was available both on the rooms' computers for the on-campus participants and online for online participants. We compiled two images, one for each group. The only difference across these two images was the order in which the debuggers are used, i.e., the code to launch the experiments that were available in the image performed different calls depending on the right order of the debuggers. Participants deployed Spa locally on the same machine they are using for the experiment. We do not let them execute on the cluster to control the experiment setup, avoid concurrent access to the cluster, and avoid differences in the network communication with the cluster between the on-campus participants and the ones that joined online. Our focus, in fact, is to analyze the participants' feedback over the different debuggers and their features and not their raw performance on big amounts of data.

To keep the participants as unbiased as possible, SpaDebug and BigDebug are randomly renamed respectively Debugger A and Debugger B in all the presentations and in the debugger's UI. Furthermore, both debugger interfaces are accessible through the same *debugger tab*. SpaDebug offers all of the features described in this paper, but the ignore mode is enabled exclusively in global mode. This was necessary to limit the ex-

planation of SpaDebug to a similar length to BigDebug's. The BigDebug reimplementation provides the key features of the original work [GIY+16], namely guarded watchpoints, simulated breakpoints, stepping and resuming, record skipping, record substitution, code patching, and tracing to input. However, instead of being a separated web GUI as in the original work [GIY+16], we integrate it in the Big Data framework's IDE, i.e Pharo Smalltalk. More details of our reimplementation are described in Section 8.2.4.4.

8.2.2.2 Debugging Assignments

As mentioned before, participants have to solve two bugs in each assignment: the first one causing an exception and the second one producing wrong results. Before tackling the second bug, they have to solve the first one. The system gives them feedback when they have solved the first and the second bug (cf. Appendix C). Furthermore, the system automatically logs the activity of the participants while debugging, e.g., time to find bugs, interaction with the UI, etc., as well as the source code of the applications when they finish the experiment.

The first assignment is the debugging of the Twitter K-Means application described in Section 8.1.1. For the first bug, we previously deleted the code that cleaned the data, which leads to data-cleaning errors because of deleted tweets that missed the hashtags. For the second error, we introduced a bug in the application's logic: the original dataset included tweets with non-ASCII characters that could not be displayed correctly within the final result. Those tweets have to be identified and removed.

The second assignment involves debugging an implementation of the popular ID3 decision tree algorithm [Qui86] that analyzes a set of Amazon Reviews to find out which of the features of the review (e.g., length of the text, amount of stars, etc.) makes the review the most helpful, i.e., produced the highest count of the `helpfulVotes` feature. The dataset is retrieved from the Amazon Customers Reviews Dataset[5] and presents amazon reviews per product category, stored in a tab separated value format. For the first bug, we modified the dataset by removing some features from some records, so that a data-cleaning error would appear when the application is executed for the first time. For the second bug, a

[5] `https://s3.amazonaws.com/amazon-reviews-pds/readme.html`

wrong classification of the number of stars makes the algorithm return a final decision tree that is not correct. The wrong classification code has to be identified and corrected.

Experimental Material Before doing an assignment, all the participants are handed several documents to help them in the assignment, listed below.

1. The description of the two assignments, including details and full code of the applications (cf. Appendices A and B).
2. Two cheatsheet containing the same information about the Spa API, Pharo syntax, and a column with information about the two different debuggers (cf. Appendix D).
3. A copy of the slides used by the host to explain Spa and the relevant debugger available at `https://soft.vub.ac.be/~mmarra/userStudy/UserStudyMaterial.zip`.

The Questionnaire After each debugging assignment, participants fill in the questionnaire used for our qualitative analysis. The questionnaire is divided into five parts:

1. **About you.** General questions to collect information about the experience of the participants with debuggers and distributed programming.
2. **General questions (for each assignment).** A set of five questions to collect information over the general debugging difficulty, whether they had enough time to solve the assignment, and how many bugs they found.
3. **Debugging with debugger A.** A set of ten questions to assess their debugging experience with debugger A, i.e., SpaDebug.
4. **Debugging with debugger B.** A set of nine questions to assess their debugging experience with debugger B, i.e., BigDebug.
5. **Overall experience.** A set of four open questions to assess their perception of the differences between the two debuggers and to collect information about technical problems.

After completing the first assignment, participants answer to sections 1,2, and either section 3 or 4 depending on which debugger they used in the first assignment. After completing the second assignment, participants answer to sections 2, 3, or 4, depending on the debugger that was used for the second assignment, and 5. A copy of the questionnaire is available in Appendix E.

8.2.3 Results

In this section, we present the results of the study for each of the four research questions that drive this study, described earlier in Section 8.2.1.

8.2.3.1 UQ 1: Does the debugger impact time to find and solve the bug?

To answer this question, we automatically measured the time that each of the participants took to correctly identify and resolve the first bug. We did this for both applications and debuggers. Figure 8.8 shows a box plot of the time it took participants to find the first bug, aggregated across the two applications and the two debuggers. We show the time to first bug of participants that managed to find the first bug within the set time limit. Particularly, 15/17 participants found the first bug when using SpaDebug and 16/17 found the first bug when using BigDebug.

We execute a Welch Two Sample t-test on this data to check whether there is a statistically significant difference between the two means. The average time to find the bug with SpaDebug is 1136.8 compared to 1766.7 for BigDebug. The test reports a p-value of 0.007252 (t-value: -2.8901, df: 28.765). Since the test's p-value is lower than 0.05 we can say that the two means are significantly different, concluding that participants took less time to find the bug with SpaDebug when compared to BigDebug.

Regarding the second bug, we do not report the results since the success rate was low for both debuggers, making the sample too small to draw conclusions. Only 6/17 participants found the second bug across the two applications when using SpaDebug, and 4/17 when using BigDebug. This may indicate that we underestimated the complexity of the second bug and that the 45 assigned minutes were not enough to solve both bugs.

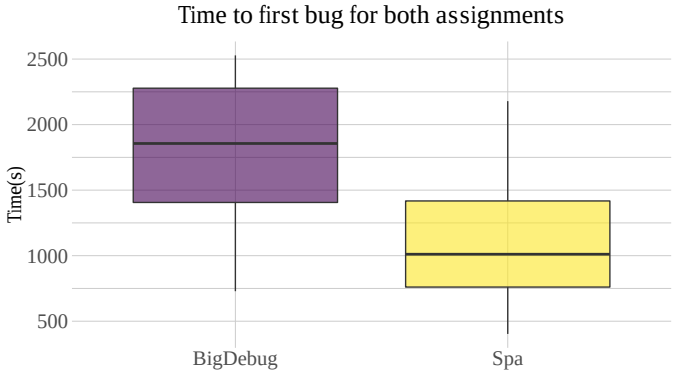


Figure 8.8: Time to find the first bug with both debuggers.

8.2.3.2 UQ 2: Does the debugger impact the evaluation of the debugging experience?

To answer this question, we asked in our post-test how much each debugger helped them in identifying the cause of the bugs, and whether debugging that particular application was difficult. To answer, participants used a Likert scale from 1 to 5 (1=Not at all, ..., 5=Very much).

How much did the debuggers help in solving the bugs? Figure 8.9 presents the results in a violin plot, i.e., a plot similar to a boxplot that shows in its variating width the distribution of the data. We observe that for BigDebug answers cluster around 2, while they cluster around 4 for Spa. The average, shown by the dot, is around 2.5 for BigDebug, and just above 4 for Spa. From these results, we conclude that the participants perceived as advantageous the features of SpaDebug, while they did so less for BigDebug.

How difficult was debugging each application? Figure 8.10 displays the answers of the participants to this question. Each line shows a debugger, colors represent which answer was given, and the percentages show, left to right, the amount of negative, neutral, and positive answers. When debugging with SpaDebug, 29.4% of participants declared

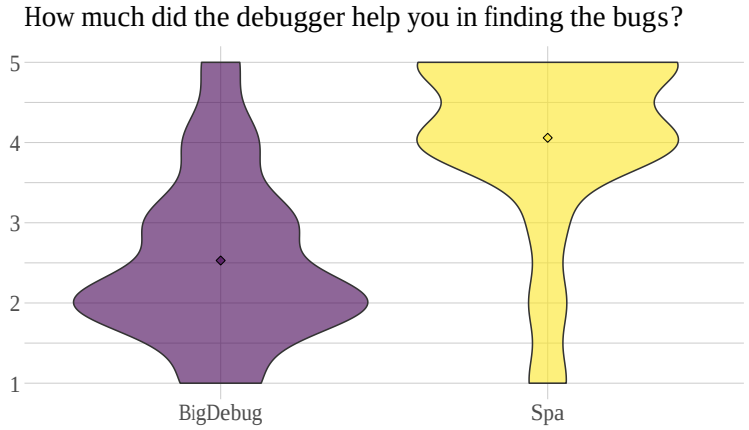


Figure 8.9: Violin plot of the answers to "How much did the debugger help you in finding the bugs?", where 1 is *not at all* and 5 is *very much*.

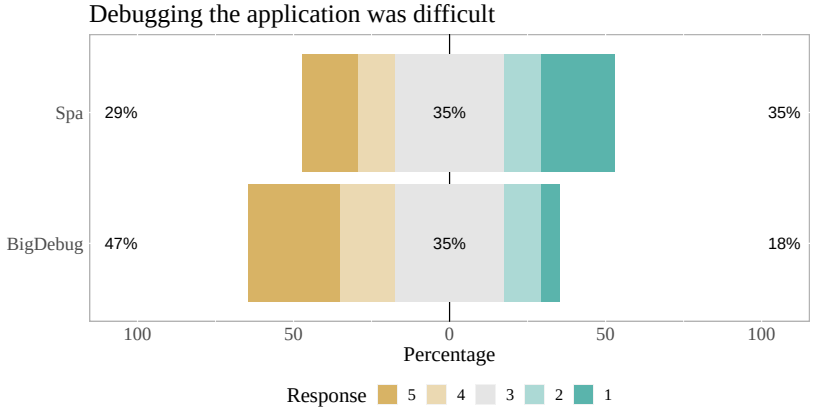


Figure 8.10: Likert plot with the answers to "Debugging the application was difficult", where 1 is *very easy* and 5 is *very difficult*.

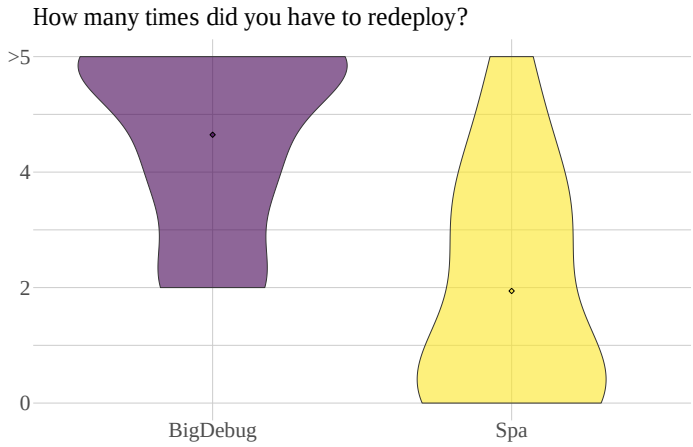


Figure 8.11: Boxplot of redeployment count across the two applications with the two different debuggers.

that debugging the application was difficult/very difficult, against 47% for BigDebug. Accordingly, 35.6% of the participants declared that debugging the application with SpaDebug was easy/very easy, against 17.7% for BigDebug.

Overall, these results make us believe that our debugger was perceived as more useful than BigDebug, leading to an easier debugging of the assignments.

8.2.3.3 UQ 3: Does the debugger influence the number of redeployments?

To answer these questions, we asked the participants how many times they had used the redeploy button, present in the UI of both debuggers. Through these buttons, participants could restart master and workers using the updated code-base, thus performing a full redeployment of the application. Redeploying an application is a relatively slow process, especially in contrast to live updating its code.

Figure 8.11 shows a violin plot with the answers. We observe a clear difference between the two debuggers: with SpaDebug, on average participants had to redeploy 1 time, with most of them being clustered between

0 and 3. For BigDebug, participants re-deployed on average 5 times, being clustered between 4 and more than 5 times. Note that the average of BigDebug might be even higher because the questionnaire only allowed "more than 5" as the maximum value.

This means that the participants had to spend less time waiting for Spa to be redeployed when using SpaDebug. We attribute the number of lower redeployments of SpaDebug to the live code-updating functionality, which allows participants to apply code patches without requiring a full redeployment. We discuss the general appreciation of the live code updating functionality in the next research question.

8.2.3.4 UQ4: How the features of SpaDebug were valued?

To answer this question, we analyze the results of two post-test questions. First, one that asked how useful they perceived the debugging functionalities of SpaDebug[6]. Second, one that asked which feature of each debugger participants missed when using the other one.

How useful are the advanced debugging functionalities of SpaDebug? To answer this question, participants were asked to rate how useful the debugger's features were using a Likert scale from 1 (not at all) to 5 (very useful). Particularly, we asked them how useful they found fine-grained stepping, coarse-grained stepping, ignore mode, and live code updating.

Figure 8.12 shows the answers for each feature. We observe that, except for the ignore feature, the majority of the participants gave a 4 or 5 rating to both kinds of stepping, as well as to live code updating, the best rated feature (as it did not receive any rating below 3). On the other hand, the ignore functionality was considered neutrally useful (with a rating of 3) by most of the participants, although they could have used it to solve the first error of each assignment.

Is there a feature of a debugger that you missed when using the other debugger? This question was asked twice after the participants had completed assignments, i.e., at the end of the study. While no partic-

[6]To avoid bias, we have asked the same question also for BigDebug, but we do not report those results as they are not relevant to this dissertation.

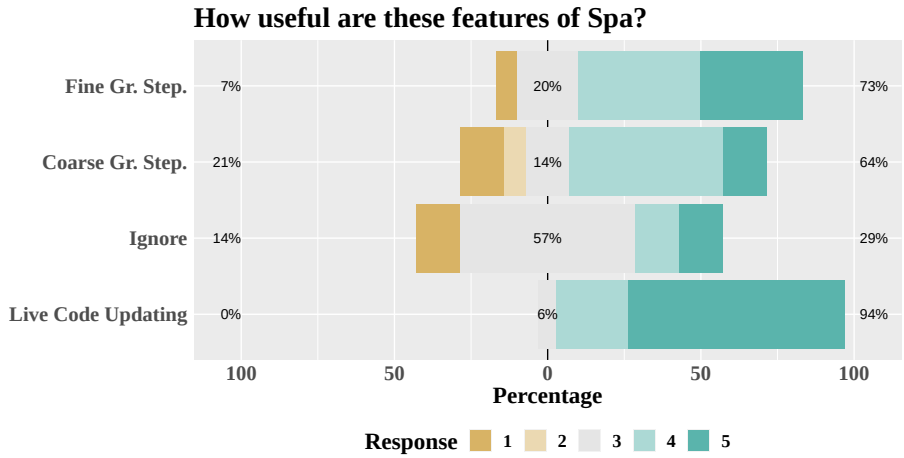


Figure 8.12: Likert scale showing how useful each feature of SpaDebug was rated by the participants. 1 is not at all, 5 is very much.

ipant indicated a feature of BigDebug that was missing in our debugger, different participants indicated several features of our debugger as missing in BigDebug.

Feature	N of participants
Local debugging	11/17
Live code updating	6/17
Full breakpoints	3/17
Abstractions over the exceptions	1/17

Table 8.2: Features of SpaDebug reported as missing in BigDebug.

Table 8.2 shows how many participants indicated in their answer one of the features of SpaDebug that they missed when debugging the assignment with BigDebug. Examples of answers that were classified as the ”local debugging” feature are: ”the interactive debug session” and “Normal stepping, easy browsing and editing code”. The other classifications took into consideration whether they explicitly mentioned the feature in their comment. Interestingly, one participant explicitly answered: ”Abstractions over the exceptions” as a feature they missed in BigDebug.

From the results in this question, we conclude that the participants generally appreciated the features of SpaDebug. This is also confirmed by the answers to the question "Which feature of Spa did you find useful?", in which 16 out of 17 participants (i.e., 94%) selected "Live Code Updating", 15 selected "Debugging locally a remote exception in an interactive way" (i.e., 88%), and 9 selected "Breakpoints on parallel execution" and "Advanced Stepping Operations" (i.e., 53%).

8.2.4 Threats to validity

Although the results of the study are very positive, we have to consider some threats that may limit the validity of our experiments. The main threats to validity are given by the number of participants, their profile, the representativity of the bugs, and their difficulty. Furthermore, another threat to validity is the quality of our BigDebug reproduction. This is discussed more in detail after the discussion over the other threats.

8.2.4.1 Number of Participants and Participant Profiles

Since we required a particular profile of participants (i.e., familiarity with both Pharo and Spark) we were only able to recruit 17 participants. For this reason, we opted for a within-participants study, which allowed us to maximize the data points. Furthermore, within-participants studies also present the advantage that single participants serve as their own control, thus balancing the differences in experience and training that could arise by having two different groups. Within participants studies, however, present different limitations, some of which are part of the threats to validity to our study that we discuss in what follows.

Since participants experience both experimental conditions, they might suffer a carryover effect. We try to limit this by *counterbalancing*, i.e., by randomizing in the two groups the order of the two tools. Having to experiment with both tools, which were both totally unknown to the participants, also led to the unavoidable presence of more explanation of the tools, since the computational model and both debuggers had to be explained to the participants. This made the study longer, and, together with the cognitive tax of having to use two different tools, potentially increased the level of fatigue of the participants. This threat, however, is also limited by counterbalancing.

8.2.4.2 Accidental Bias

In order to avoid accidental bias, we have taken several measures. First, we renamed the two debuggers to Debugger A and Debugger B randomly and we also randomized the order in which participants were exposed to the two debuggers. Second, the front-end of both the debuggers have a similar look and feel, as they are both integrated in the same main debugger tab. Finally, we prepared demos and presentations to be of the same length for the two debuggers.

We also informally asked to some of the participants at the end of the study which they thought was SpaDebug, and we received mixed answers. We did not, however, include this question into the questionnaire and thus we have no scientific way to show that accidental bias was completely avoided.

8.2.4.3 Bugs Representativity and Difficulty

The assignments were both based on popular algorithms for data analysis, but not all participants were familiar with them. Furthermore, the bugs were not taken from public reports but were injected into the code based on common bugs for these data analysis applications as described in Section 2.4.5.1, i.e., one bug caused by dirty data and one by a code defect. This may have led to too difficult bugs to be solved in the time frame of the assignments; while all of the participants except one solved the first bug in both applications, only 6 and 4 participants solved correctly the second bug in the first and second assignment, respectively. We believe, however, that this does not limit the results of our study, since the quantitative experiments are based only on the time to find and solve the first bug, and the qualitative experiments are guided by the overall experience with both debuggers.

8.2.4.4 BigDebug Reproduction

When designing the study, we had to decide how to compare our approach to BigDebug, the closest related work. Since Spa, and thus SpaDebug, runs on top of Pharo Smalltalk, we considered that using the real BigDebug on Apache Spark would have entailed not only a difference in the debugger, but also a cognitive mismatch between the IDEs, tooling, APIs, and programming languages. For this reason, believing in the concepts

of the reproducibility of scientific results, we decided to re-implement the core features of BigDebug. To this end, we employed the details described in the original paper from Gulzar et al. [GIY+16] and other details available on the project website and different publicly available demos. This clearly represents a threat to validity, but we believe that it is a lower threat than letting participants use two different tools in two different environments. Furthermore, technical differences between Spa and Spark, e.g., optimizations, make our reproduction of BigDebug simpler, since it does not have to de-optimize Spark execution. Moreover, it is possibly not as performant as the original one. This is why we do not assess the raw performance of the two approaches in our user study, but focus on studying the usability of their features on debugging simple assignments that we know both debuggers can handle in the same way.

We reimplemented BigDebug in good faith and tried to reproduce to their best the main features of the debugger. However, our reproduction presents some differences with the original implementation which we summarize in what follows. For more details about BigDebug, we refer the reader to our description of BigDebug in Section 2.4.5.3.

8.2.4.4.1 The BigDebug Reproduction UI As mentioned before, we integrated the BigDebug UI into the general Spa UI designed in Pharo Smalltalk.

The participants could access the data causing an exception or watchpoint in the debugger tab, as shown in Figure 8.13. On the left side, there is a representation of the execution pipeline in which the filter is marked with a red exclamation mark, indicating that the error is there. On the right side, under watchpoint, there are all the records that caused the exception. By selecting one record, the rightmost view updates showing the textual form of the record. The three buttons in the bottom right allow developers to modify the record (upon changing its textual form), skip the record, or execute the tracing to input. Finally, the two buttons on the bottom left side allow developers to step (over) a transformation, and to resume the execution.

The stack trace. As in the original BigDebug, the stacktrace is available to the developers. Figure 8.14 shows an example of such a stack trace printed in the Pharo Transcript (i.e., a sort of standard output).

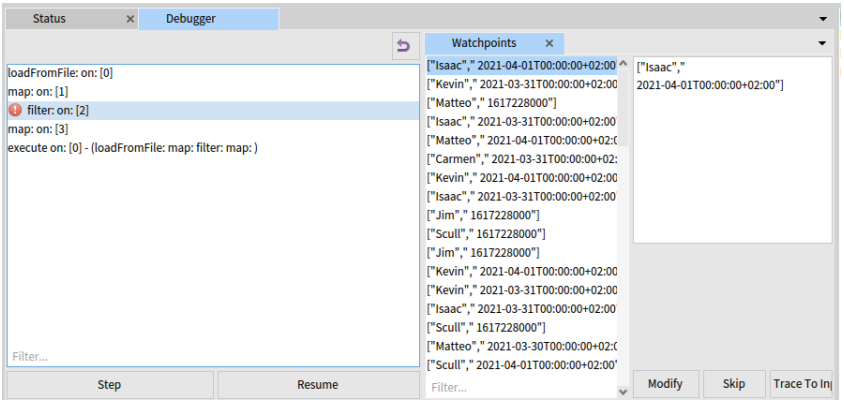


Figure 8.13: The debugger tab of BigDebug.

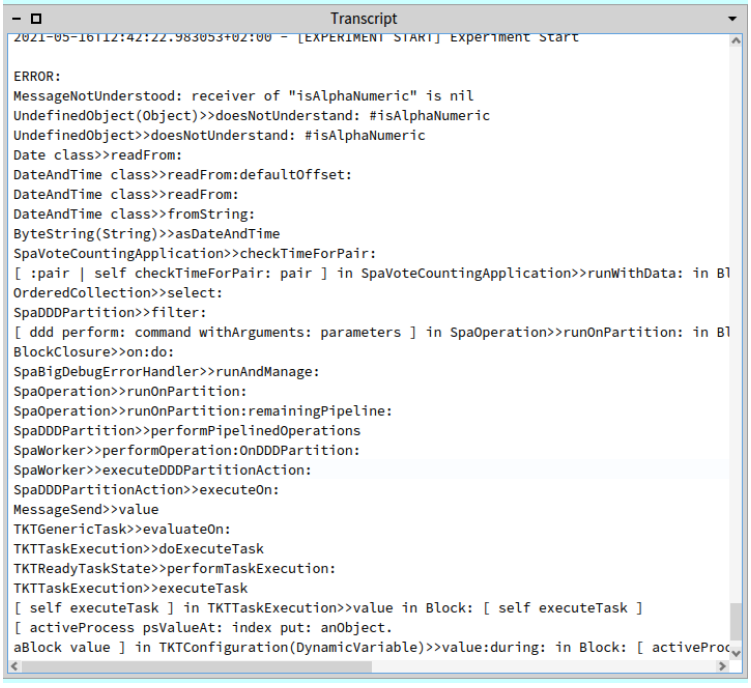


Figure 8.14: Stacktrace of an exception as shown in BigDebug.

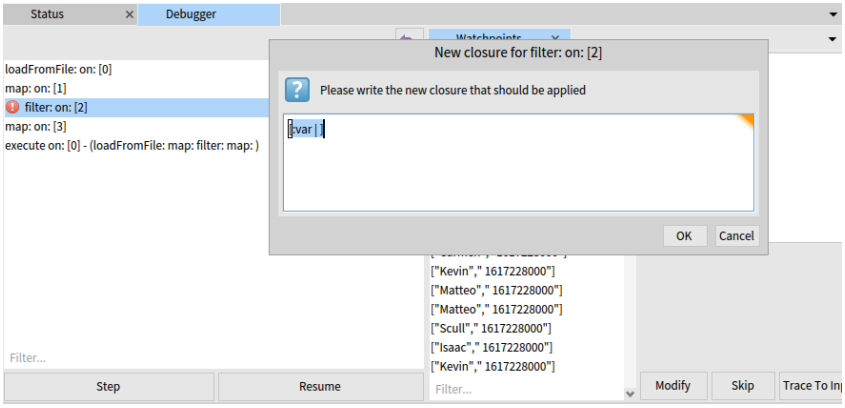


Figure 8.15: Code patching in BigDebug.

Code patching. Finally, patching the code executed by a certain transformation happens similarly to the original BigDebug. By double-clicking on one of the transformations in the left panel of Figure 8.13, the dialog shown in Figure 8.15 pops up, giving the participant the possibility to fill in a closure to be applied instead of the original one.

8.2.4.4.2 The BigDebug Reproduction Details While adding simulated breakpoints and watchpoint was straightforward, only requiring small changes in the Spa API and runtime execution, replaying from checkpoints, tracing, and code patching required some more implementation work, discussed in what follows.

Data Provenance Backend. As described in Section 2.4.5.3, BigDebug relies on Titian, a framework that extends Apache Spark with data provenance. In practice, Titian makes sure to tag and persist data with provenance metadata across stage boundaries, i.e., before data is reduced. The first step to implementing BigDebug on top of Spa was to build a data provenance extension, that could thus persist data at stage boundaries and tag records to track which partition they belonged to before and after shuffling. Our implementation of the data provenance engine powers the checkpoint-based replay of our BigDebug reproduction, as well as tracing to input. However, we only support data provenance on a record across one stage boundary. Although this limits the power of our BigDebug reproduction, tracking down the bugs in the experiments did

not require tracing across more than one stage boundary. As such, the limited data provenance support did not impact the usage of the trace to input functionality during the user study.

Code Patching To implement the code updating features of BigDebug, we compile the changed lambda function in an empty execution context, thus capturing only the input parameters of the lambda function and no other variables from the execution context. The complied lambda function is then substituted in the execution pipeline of the halted execution. This relies on Spa's internal reification of the pipeline, which associates transformation calls to an internal BlockClosure representation.

8.3 Discussion

In this chapter, we evaluated our approach for debugging Spark-like applications through a performance evaluation and a user study. We now discuss our prototype SpaDebug with regards to the different criteria for a debugger to be suitable for Big Data application introduced in Section 2.5.

Table 8.3: Overview of related work compared to IDRA_{MR} and SpaDebug.

Debugger	*Side Effects*	*Replay Point*	*Halt & Inspect*	*Stepwise Exec.*	*Dom.S. Ops.*	*Code Updates*	*Ignore Errors*
Arthur	Both	Start	✗	✓	✗	✗	✗
Graft	Both	Start Rec.	✗	✓	✗	✗	✗
Daphne	Both	Check.*	✓	✓	✗	✗	✗
BigDebug	Global	Check.	✓	✗	✓	✓*	~
IDRA_{MR}	Scoped	No	✓	✓	✓*	✓	✗
SpaDebug	Scoped	No	✓	✓	✓	✓	✓

Table 8.3 revisits Table 6.1 to include SpaDebug. Particularly, SpaDebug offers scoped side effects and replay-free debugging, together with all the other characteristics already supported by IDRA_{MR}, i.e., halting and inspecting the execution, sequential stepping, and live code updates. It also expands the domain-specific debugging support of IDRA_{MR} with domain-specific stepping operations tailored to the Spark-like model. SpaDebug thus adheres to all the criteria defined in Section 2.5. Further-

more, the results of this validation showed that live online debugging is not only applicable to Big Data processing but also scalable tanks to the optimizations introduced in SpaDebug. Moreover, the user study conducted to evaluate SpaDebug shows that its live debugging features were highly appreciated by the participants. Although in this chapter we validated only the performance of SpaDebug, we believe that all the optimizations and the support for a relaxed computational model that we introduced in Chapter 7 are also applicable to the Map/Reduce model and hence to IDRA_{MR}. More concretely, applying them to the Map/Reduce model involves the adaptation of the two employed optimizations for using delta stacks and partition windowing, as well as the adaptation of the domain-specific stepping operations to step to the `reduce:` instead of the next transformation. Finally, also the support for ignoring errors can be extended to Map/Reduce programs by both applying the global execution modes and extending the API of Port with methods for ignoring while mapping and reducing.

8.4 Conclusion

In this chapter, we validated our debugging approach for Spark-like applications, presented in Chapter 7. This includes our out-of-place debugging solution for Spark-like applications which uses Sarto's stack tailoring operations, combined with the concepts of a centralized debugging session through composite debugging events presented in Section 6.2.2.

Particularly, we conducted a quantitative and qualitative evaluation, to assess both the performance and the usability of our debugging solution. By answering five research questions, the quantitative evaluation showed that our approach scales to an increasing amount of data and parallel exceptions. Furthermore, we showed the positive impact of partition windowing and composite exceptions with delta stack in reducing the size of dynamic local checkpoints. Finally, we showed that the relaxed computational model introduces a negligible overhead when no failures are present, and an overhead linear to the number of failures when ignoring them.

For the qualitative part of this validation, we conducted an experimental user study with 17 participants using a within-participants strong experimental design. In this user study, participants had to solve debug-

ging assignments with our debugger and with a reproduction of BigDebug, the closest related work for debugging Spark-like applications. Overall, the results of the study showed that the features of our debugger presented in this dissertation, particularly in Chapter 7, are generally perceived as useful and that our debugger has helped the participants in solving the bug faster than with BigDebug. The live debugging and live code updating functionalities were appreciated by most of the participants, thus showing that a live debugger for Big Data applications is not only possible but also highly usable and functional. On the other hand, the systematical ignoring of exceptions was not particularly used by the participants probably due to the limited explanations and its novelty.

To conclude, we believe that both the performance benchmarks and the positive results of the user study show the potential and validity of a live out-of-place debugger for Big Data applications, particularly for Map/Reduce and Spark-like data science applications.

Chapter 9

Conclusion

This dissertation introduced a novel live debugging approach for Big Data applications with support for relaxed computations. This chapter concludes this dissertation by summarizing its most important concepts and by revisiting our contributions. We also present a discussion over the limitations of our work and indications of possible future work.

9.1 Overview of our Approach

At the beginning of this dissertation, we introduced our research statement, stating that a live online debugging solution could help developers in the process of developing and debugging Big Data applications. After an initial definition of problems in current debugging approaches for Big Data (cf. Section 1.1), we defined in Section 2.5 different criteria that a debugger for Big Data should uphold. Below, we discuss how our approach satisfies the criteria.

No replay times, halt & inspect, stepwise execution. In Chapter 5, we introduced out-of-place debugging, which enables live online debugging of remote applications in isolation by transferring the execution state of an application to a different process for debugging. This avoids replaying the execution while supporting halting the execution through breakpoints. It also offers the inspection and stepwise execution capabilities of an online in-place debugger.

Scalability to Big Data. In Chapter 5, we revisited the architecture of out-of-place for debugging the Master/Worker execution model which lies at the basis of Big Data frameworks such as Map/Reduce and Spark. To make debugging scalable, we centralize the debugging of many parallel exceptions so to limit the communication between the workers and the developer's machine used for debugging. Throughout Chapters 6 and 7, we further introduce optimizations for reducing the size of debugging sessions through cutting the stack, partition windowing, and using delta stacks in composite debugging events. As shown in Chapter 8, thanks to these optimizations the overall model scales to the amount of data and number of parallel exceptions.

Domain-specific operations. In Chapter 6, we describe three different debugging modes to debug a Map/Reduce execution with different data in virtual partitions. This allows developers to debug a composite debugging event on data partitions composed by e.g., all failure-inducing records. In Chapter 7, we defined different domain-specific stepping operations that enable stepping through the concepts of a Spark-like execution, such as step to next transformation, step to next iteration, and step to action result.

Live code updating. Out-of-place debugging introduces live code updating capabilities, that were applied in Chapter 6 to the Master/Worker execution model, particularly for Map/Reduce. We use the same live code updating support in our approach for debugging Spark-like applications.

Support for ignoring errors. In Chapter 7, we introduced a relaxed computational model inspired by the concepts of acceptability oriented computing to add support for ignoring exceptions in the Spark-like model. Particularly, developers can decide to systematically ignore errors up to a predefined threshold in terms of the size of the input data.

This dissertation presented two concrete prototypes of our live debugging approach. In particular, we described IDRA_{MR} and SpaDebug, two debuggers for the Map/Reduce and Spark-like model, respectively. To implement these debuggers, we used different stack-tailoring operations

described in Chapter 4. We applied our debuggers to debug applications for two Big Data frameworks in Pharo Smalltalk, namely Port and Spa, introduced in Chapter 3.

We validated our debugging solution in Chapter 8 through several performance benchmarks and a user study. With the benchmarks, we assessed the overhead and scalability of our approach. With the user study, we compared our debugging solution to a reimplementation of BigDebug. We adopted a mixed-method experimental design, using two within-participants groups. The 17 participants debugged two debugging scenarios using both debuggers.

Overall, the debugging approach and experiments presented in this dissertation validate our initial research statement that an interactive online debugging solution can improve the debugging experience of Big Data applications by not replaying the execution and offering a full online debugging solution, scalable to Big Data. First, through the performance benchmarks (cf. Section 8.1) we show that (i) our approach scales to the amount of data and parallel exceptions, (ii) the optimizations of dynamic local checkpoints are effective in decreasing the network overhead of our debugger, (iii) our debugger saves time in comparison to replay debugging techniques, (iv) the relaxed computational model introduces a low overhead when no failures are present, and that (v) it scales linearly to ignoring an increasing amount of exceptions. Furthermore, the user study (cf. Section 8.2) results show that when using our debugger (i) participants took less time to find the first bug, (ii) evaluated their debugging experience as better when compared to the other solution, (iii) had to re-deploy fewer times thanks to the live code updating capabilities, and (iv) positively evaluated all of the features of our debugger. Finally, we show that applying a relaxed computational model is possible and works correctly to ignore data cleaning errors in our use cases.

9.2 Restating the Contributions

In this section, we restate and summarize our contributions:

Out-of-Place debugging. A debugging model for the online debugging of remote executions, based on the transferring of the execution state from the process of the application to an external process. This enables replay-free debugging in isolation and is complemented with

live code updating capabilities for recording changes during debugging and updating the remote application. We showed that out-of-place debugging is a scalable approach for debugging Big Data applications and that it can be applied not only to Smalltalk runtimes but also to runtimes that do not offer reflective capabilities, such as WebAssembly. The out-of-place debugging model lies at the core of our debugging support for Big Data applications and has been explored for two programming models in a Big Data context: the Map/Reduce and Spark-like models.

Live debugging of Map/Reduce applications. A live debugging approach based on out-of-place debugging through the introduction of *composite debugging events*, domain-specific debugging modes, and live code updating. Through these features the approach is especially designed for to the parallel execution model of Map/Reduce. We prototyped our solution in IDRA_{MR}, our debugger for Port applications that implements all of the described concepts by using some of the operations of Sarto. We showed the different UIs of the debugger and we provided an evaluation showing how the debugger can practically be used to debug a Port application in the domain of blockchain analysis.

Live debugging of Spark-like application. A debugging approach for Spark-like applications, that enables both debugging and ignoring of errors. Our debugging approach extends out-of-place debugging to centralize debugging sessions with *dynamic local checkpoints*. It improves the debugging experience with more domain-specific debugging operations (i.e., step to next iteration, transformation, or to the action result) and is complemented with a relaxed computational model for the systematic ignoring of errors. We validated our overall approach through performance benchmarks and through a user study that showed that our debugger helped the participants to find the bug and revealed the live debugging features were appreciated by the participants.

A stack-tailoring instrumentation layer. An instrumentation layer to tailor call-stacks for debugging framework executions through a set of six operations. We prototyped our solution in Sarto and validated it by assessing the impact of the different operations on the

execution in the context of debugging three execution frameworks. Furthermore, we employed the concepts of Sarto to implement our debugging solution for both Map/Reduce and Spark-like programs, showing in our validation that the stack-tailoring operations helped optimizing the size of debugging sessions, making our approach more scalable.

9.3 Discussion

In this section, we discuss some limitations of our approach. We do not discuss here the threats to the validity of the user study, as they were already discussed in Section 8.2.4.

Performance of the relaxed computational model. As shown in the validation (cf. Sections 8.1.6 and 8.1.7), the relaxed computational model adds minimal overhead to the computation when enabled, which decreases as the duration of the application increases. As already mentioned, we believe that this is caused by the initialization of a replicated counter which is built using an unoptimized ad-hoc solution. Similarly, the overhead of ignoring exceptions is linear to the amount of ignored exceptions, which decreases when we disable the updates of the counter. Thus, the lack of optimizations when ignoring exceptions limits the performance of the relaxed computational model. While sampling the counter updates helps mitigate the overhead, a more performant implementation of a replicated counter or a redesign of the ignoring exception mechanism may positively impact the performance of the relaxed computational model.

Performance of Port and Spa. The two prototypes for Map/Reduce and Spark-like frameworks (i.e., Port and Spa) allowed us to easily experiment with our debugging solution, but they do not feature many optimizations included in production systems such as Apache Spark. For example, in Spa the implementation of pipelining is very limited and the shuffling of DDDs is not optimized and thus is more time-consuming than in Spark. This is because Port and Spa are research prototypes and the access to their code so far has been limited to very few developers. However, we have already tested and improved our implementation of the two frameworks with different

researchers, e.g., to develop the blockchain analysis application described in Section 6.4. The frameworks' scalability could improve provided we had more engineering resources or manpower.

Debugging optimized code. As mentioned above, popular Big Data frameworks perform several optimizations to execute the code in the fastest way possible. This often entails changing the order and the way operations are executed or merging some operations into one, which leads to differences between the actual code that is executed and the one the developer expects to see in the debugger. As Port and Spa do not include such optimizations, our debuggers do not deal with debugging highly optimized executions. Applying our approach to frameworks that do have that optimization may require more engineering work to deoptimize the execution before debugging. As this happens already in many environments, including Pharo and Self in which the virtual machine deoptimizes the call-stack before reifying it for debugging [IMBGB20, HCU92], we are confident that this limitation can be easily overcome with more engineering work.

Debugging speculative executions. Popular Map/Reduce implementations such as Hadoop Map/Reduce apply concepts of speculative execution such as backup tasks to help and deal with stragglers, as already described in Section 2.2. This can lead to *reduce tasks* being scheduled before all of the *map tasks* are finished. Consider an error appearing during one of those speculatively scheduled *reduce tasks*. The debugger needs to choose whether to report the error directly, or to first wait for all the *map tasks* to complete before reporting it. The debugger should also consider that if another error happens during a *map task* concurrent to a speculative *reduce task*, the reduce task should have never been executed. Since Port does not support speculative execution of tasks, we do not present an explicit solution to this problem. In the current implementation, IDRA_{MR} will report all of the errors that appear, possibly misleading the developer by showing both an error in the map and in the reduce of the same execution. IDRA_{MR}, however, could be easily modified to consider the execution of speculative tasks and thus wait that all map tasks are finished before showing an error in a speculatively executed *reduce task*.

9.4 Future Work

In this section, we discuss different research contexts in which our work can be applied and further extended.

9.4.1 Application to Other Big Data Environments

The out-of-place debugging model presented in this thesis and used to build our debugging solution uses the concepts of stack reification in its implementation to transfer the debugging session. Applying out-of-place debugging and thus our debugging solution to other runtime environments may entail changes in the runtime, which are more impacting than an implementation based on metaprogramming like ours. This was already shown in Section 5.2 when discussing the differences between IDRA, our out-of-place debugger, and WOOD, an out-of-things debugger based on out-of-place debugging targeting IoT Webassembly applications.

To apply our debugging approach to other environments, more research is needed to identify the specific choices that need to be taken depending on the runtime and on the execution model. In what follows, we discuss how our approach could be applied to Apache Spark and streaming frameworks.

9.4.1.1 Debugging Apache Spark

As our debugging solution described in Chapter 7 directly targets a Spark-like model, we believe that with the correct adaptations our approach can be implemented also in Apache Spark. This will entail interacting with the runtime, i.e., the JVM (Java Virtual Machine), to make sure that the correct debugging information is made available to the developer. As the JVM already offers support for instrumentation and debugging, we believe that capturing and transferring the debugging session should be feasible. Furthermore, dynamic software updating has already been explored on the JVM [GSJ09, PVH14] and thus it should be possible to update Spark application on a larger scale than what BigDebug already provides. Finally, as already described in the discussion on limitations, the approach should be adapted to allow the debugging of Spark's optimized executions.

9.4.1.2 Debugging Streaming Frameworks

In the latest years streaming frameworks became very popular. While Map/Reduce and Spark are born to analyze files, i.e., data in batch, those frameworks analyze data streams instead of files. Example of popular streaming frameworks are Apache Flink [CKE+15], Storm [Apae], and Kafka [Apab]. Several stream processing frameworks are offered as a service through popular data processing services such as AWS Kinesis [Ama], Google Cloud's Dataflow [Goob], and Microsoft Azure Stream Analytics [Mica]. Streaming frameworks enable the incremental analysis of data as soon as it is available in the stream, thus powering faster decision-making. We believe that our approach can already be applied to those streaming frameworks that use *micro-batching* to enable the analysis of data streams, i.e., collecting data from a stream for a fixed period of time to then analyze it as a batch file. This is, for instance, the approach supported by Apache Spark through its Spark Streaming library. Other approaches, on the other hand, rely on dataflow analysis to model their execution. Thus, it would be interesting to explore how the concepts of dynamic local checkpoints and, more in general, out-of-place debugging can be extended and adapted to such environments.

9.4.2 Application to Other Execution Models

Overall, our debugging approach is built on the concept of debugging a parallel execution in a centralized way, particularly applied to Master/-Worker models. This means that several concepts of our work such as out-of-place debugging in a distributed setting, composite debugging events, and coordination of live code updates could directly be applied to other distributed and parallel models besides Big Data processing.

For instance, we could consider using our debugger in the context of a client-server architecture, possibly including multiple servers. Out-of-place debugging enables the debugging of the different servers in a centralized way, and the optimizations introduced through Sarto, such as stack cutting, can also be used in this context as shown earlier in Chapter 4. Exceptions that happen in parallel could be captured, composed, and debugged through a similar view as the one we offer in IDRA_{MR} and SpaDebug.

With respect to parallel applications, our approach could be applied to debugging parallelized programs through the popular fork/join execution model. This model parallelizes the execution of a certain function/task by forking it in different processes and through a certain degree of parallelization. Errors in the parallel execution could be treated using composite debugging events even deploying a single debugger monitor if all of the forked tasks run on the same virtual machine. Domain-specific operations such as step to the next element could also be used in this context, especially if the forked operations are map-like, and new domain-specific stepping operations could be devised for this and other models.

9.4.3 Offline Out-of-place Debugging

Although out-of-place debugging is designed to provide an online debugging experience for remote applications in isolation, the fact that we extract the execution state and store it in the debugging session could also potentially enable offline debugging. For instance, the same debugging session that is transferred over the network for debugging could be stored in a file or in an image (in image-based environments such as Smalltalk).

Storing debugging sessions enables offline debugging meaning that a developer could decide to debug certain executions post-mortem. However, since the debugged application may not be still running when debugging it offline, we cannot ensure some functionalities of out-of-place debugging, e.g., accessing non-transferrable resources and performing live code updates.

9.4.4 Live Code Updating

Regarding code updating, we have only scratched the surface of the field of dynamic software updates for Big Data applications. More research could be focused on improving the coordination of live code updates in Big Data frameworks, especially in the case of streaming frameworks that continuously analyze data. For example, update modes could be made available to developers to indicate how the application should be updated, what happens to currently running operations, etc. Live code updates could also be further integrated with existing approaches for atomic dynamic software updates such as gDSU [TPB+18]. This would ensure updating the remote applications in safe points, possibly requiring less coordination

by the master and thus a smoother update of running applications. Furthermore, the coordination of live code updates is a more general problem of updating distributed systems, and we could further explore applying our solution to other Master/Worker models, as well as other distributed execution models such as actor-based applications.

9.4.5 Advanced Ignoring of Exceptions

Our application of acceptability oriented computing to Big Data could further be explored to both improve the parameters for ignoring errors and further assess the applicability of the approach. For example, we use a percentage of the input data as ignoring threshold, but this could be defined more dynamically through user-defined functions that have access to application results and global parameters. Furthermore, future work should study both formal and empirical approaches to further assess the impact of ignoring exceptions in more data science applications, possibly enabling a larger use of the relaxed computational model presented in this dissertation. For example, the domain of approximate computing gives already an idea of some applications that allow some degree of accuracy loss [CCRR13], as is the case for the K-Means application that we used to validate our approach.

9.5 Closing Remarks

In this dissertation, we presented an online and live debugging approach for debugging Big Data applications, particularly for the Map/Reduce and Spark-like programming models. The debugger enables full online debugging capabilities (i.e., breakpoints, full state inspection, and step-wise execution) while avoiding replaying the execution for debugging. It also offers live code updating capabilities and domain-specific debugging modes and operations which can reduce the debugging time of an execution. Finally, we augmented our debugging solution with a relaxed computational model which enables the systematical ignoring of errors. This avoids invalidating long executions because of a few records in those applications that allow for some degree of accuracy loss.

Most of the focus of the dissertation was on describing novel debugging models and functionalities to debug the parallel execution of a Big

Data framework in a centralized and live way. Through the presented optimizations we made our approach scalable, and we believe that it could be applied in different execution models and environments for Big Data only with minor modifications.

To conclude, based on the positive results of our user study, especially regarding the features of SpaDebug, we believe that our live debugging approach for Big Data could lead to a more interactive development and debugging experience of Big Data frameworks.

Appendix A

The Twitter K-Means Application Assignment

The first experiment consists of fixing two bugs on an application which analyzes the impact of hashtags on the number of likes of different Tweets employing a k-means clustering algorithm. The goal of this algorithm is to find groups in a dataset, with the number of groups represented by the variable "k". It works iteratively to assign each data entry to one of the "k" groups based on feature similarity.

A.1 The Application

The purpose of this application is to understand which hashtags are more popular based on the likes tweets receive. The analyzed tweets are taken through the Twitter Streaming API, and are a set of random tweets generated during the specific time of the recording from the stream (sometime in 2017). Tweets are stored in a JSON format. The main method of the application is `runWithData:` at the `KMeansClusteringTweetsExperiment` class, presented below.

```
1 KMeansClusteringTweetsExperiment>>runWithData: data
2     parsed := self parse: data.
3     pairDDD := self makePair: (parsed filter: [:e | e isNotNil]).
4     scores := pairDDD reduceByKey: [ :v1 :v2 | v1 + v2 ].
5     trending := scores
6         top: nTags
7         withBlock: [ :v1 :v2 | v1 value > v2 value ].
```

```
8   likes := (trending asDictionary keys
9     collect:
10      [ :hashtag | hashtag -> ((pairDDD filter: [ :kv | kv key = hashtag
      ]) map: #value) ])
11    asDictionary.
12  results := Dictionary new.
13  trending
14    doWithIndex: [ :t :idx |
15      rdd := (likes at: t key) execute.
16      centroids := rdd takeSample: nClusters.
17      kmeansResult := KMeans
18        runKMeansOn: centroids
19        withData: rdd
20        maxIterations: maxIterations
21        treshold: treshold.
22      rdd deallocate.
23      results at: t put: kmeansResult.].
24  ^ results
```

First, line 2 parses the tweets, and line 3 makes pairs for all the tweets by associating each of the hashtags of a tweet with the number of likes the tweet received. Line 4 a `reduceByKey:` is applied to sum, for each hashtag, the number of likes it got. Lines 5,6, and 7 send the message `top:withBlock:` which extracts the trending N hashtags, i.e. the N hashtags with the most total likes. For each hashtag, line 8 and 9 filters the pairDDD data structure to get only the values that have that particular hashtag. This is returned as a Dictionary. Lines 11 to 21 correspond to the iterative part of the k-means algorithm called on each of the trending hashtags (and the subset of tweets that include that hashtag). In a nutshell, the k-means algorithm takes care of running different iterations to find the optimal clusters, finally returning a result for each of the hashtags.

A.2 The Bugs

Bug1 The application fails while parsing the tweets.

Bug2 The final result presents hashtags that cannot be displayed correctly because they are non-ASCII characters. Make sure those hashtags are not considered at all in the analysis.

Please note that the bugs do not reside in the iterative part of the k-means algorithm, i.e. you can consider the calls after line 12 of the `runWithData:` method to be correct.

About the output When it does not fail, the output of this application is a Dictionary, with hashtags as key, and clusters as values.

The full implementation is presented in Appendix A.3 for completeness.

A.3 Code of the Twitter KMeans Application

```
1  KMeansClusteringTweetsExperiment>>parse: data
2  ^ data
3      map: [ :e |
4          TwitterParser createTweet: (NeoJSONReader fromString: e) ].
5
6  KMeansClusteringTweetsExperiment>>makePair: parsed
7      ^(parsed
8          flatMap: [ :tweet | tweet hashtags collect: [ :tag | tag -> tweet likes ]
           ])
9          execute.
```

```
1  TwitterParser>>createTweet: aTweet
2      | text hashtags retweetedTweet likes user id ret |
3      text := aTweet at: 'text'.
4      id := aTweet at: 'id'.
5      hashtags := ((aTweet at: 'entities') at: 'hashtags') collect: [ :dic | dic at: '
           text' ].
6      retweetedTweet := aTweet at: 'retweeted_status' ifAbsent: [ nil ].
7      likes := retweetedTweet ifNotNil: [
8          retweetedTweet at: 'favorite_count'.
9          ] ifNil: [
10         user := aTweet at: 'user'.
11         ((user at: 'favourites_count') / (user at: 'statuses_count')) asInteger.
12         ].
13     ^ Tweet new id: id ; text: text ; hashtags: hashtags ; likes: likes ; yourself.
```

```
1  KMeans class>>runKMeansOn: means withData: aPipelineDDD
           maxIterations: maxIters treshold: treshold
2      | clusters averages converged tweetsPerCluster newMeans |
3      newMeans := means.
```

```
1 to: maxIters do: [ :it |
    clusters := (((aPipelineDDD
    map: [ :likes | (self nearestCluster: newMeans forValue: likes) -> likes ])
    reduceByKey: [ :sum :current |
      sum incrementCount.
      sum addToTotal: current ]
    injecting: TagStats new)
    map: [ :kv |
      kv key
        ->
          (TagInfo new
            count: kv value count;
            mean: (kv value total / kv value count) asFloat;
            yourself) ]) execute.
  averages := (clusters map: [ :e | e value mean ]) getCollection.
  converged := (averages
    collectWithIndex: [ :avg :idx |
      {avg.
      (newMeans at: idx)} ])
    allSatisfy:
      [ :twoAverages | (twoAverages first - twoAverages second) abs < treshold ].
  converged
    ifTrue: [ "we stop"
      tweetsPerCluster := (clusters map: [ :ti | ti value count ]) getCollection.
      ^ averages
        collectWithIndex: [ :val :idx | val -> (tweetsPerCluster at: idx) ] ].
  newMeans := averages ].
tweetsPerCluster := clusters collect: [ :ti | ti value count ].
^ averages
  collectWithIndex: [ :val :idx | val -> (tweetsPerCluster at: idx) ]
```

Appendix B

The Amazon ID3 Assignment

This experiment consists of fixing again two bugs, this time on an application which processes Amazon's reviews employing a machine learning algorithm called ID3 (Iterative Dichotomiser 3)[1]. In a nutshell ID3 constructs a decision tree from a dataset. The algorithm begins with finding the attribute with highest information gain. On each iteration of the algorithm, it iterates through every unused attribute of the dataset and calculates the entropy or the information gain of that attribute. It then selects the attribute which has the smallest entropy (or largest information gain) value. It then creates a decision tree node containing that attribute, and recurse on subsets of the dataset using the remaining attributes.

B.1 The Application

The purpose of this application is to analyze Amazon reviews to see which characteristic of the review (text length, stars, etc.) makes it the most useful (i.e. voted as useful by other users). The analyzed reviews are retrieved from Amazon's open access servers, and represent reviews for a particular category of products (e.g office supplies or software). The main method of the application is `runWithData:` at the `AmazonDecisionTreeLearning` class which implements the main steps of ID3 as shown below:

```
1 | AmazonDecisionTreeLearning>>runWithData: data
```

[1] `https://en.wikipedia.org/wiki/ID3_algorithm`

```
2  | parsed max sample res |
3  parsed := (self parse: data) execute.
4  max := self maxIndexOf: (self calculateIgOnFeatures: parsed).
5  sample := (parsed take: 1) first.
6  res := self
7     id3WithData: parsed
8     target: max
9     attributes: (sample collectWithIndex: [ :s :idx | idx -> s asArray ])
      asDictionary.
10 ^ res.
```

First, the dataset is parsed (line 3). The reviews are stored in a tab separated value format (tsv). This is similar to a csv format but with "/t" instead of commas as separator.

Line 4 finds the attribute with the highest information gain and stores it in the `max` variable. This happens by calculating the entropy for each feature through `calculateIgOnFeatures:` and then selecting the one with the maximum information gain through `maxIndexOf:`.

Line 5 extract a sample from the parsed dataset, and, lines 6 to 9 correspond to performing the iterative part of the algorithm to construct the decision tree as explained before. `id3WithData:target:attributes:` thus returns the constructed decision tree representing the results. Finally, the tree is converted into a dictionary and returned as the result of the `runWithData:` method.

It is important to know that during parsing some of the data is normalised (in the `extractFeatures` method of the class `AmazonReview`), to extract discrete values from otherwise continue ones.

B.2 The bugs

Bug1 The application fails while parsing the reviews.

Bug2 The final result is not correct: one of the features is not correctly normalised, and this produces the wrong final decision tree.

Similar to the first experiment, please note that the bugs do not reside in the iterative part of the ID3 algorithm, i.e., you can consider the calls after line 5 of the `runWithData:` method to be correct.

About the output The output of the application, when it does not fail, is a DecisionTree. A DecisionTree stores the index of one attribute that it refers to, and a set of subtrees, one for each possible discrete value of that attribute.

The full implementation is presented in Appendix B.3 for completeness.

B.3 Code of the Amazon ID3 Application

```
AmazonDecisionTreeLearning>>parse: attributesDDD
    | parsed |
    ^ self extractAttributes: ((attributesDDD filter: [ :l | (l beginsWith: '
      marketplace') not ]) map: [:line | self parseLine: line]) .

AmazonDecisionTreeLearning>>parseLine: aLine
    | splitted |
    splitted := aLine substrings: (Character tab asString).
    ^ AmazonReview new name: (splitted at: 6) ; marketplace: (splitted at: 1)
      ; category: (splitted at: 7); stars: (splitted at: 8) ; helpfulVotes: (
      splitted at: 9); verified: (splitted at: 12) ; reviewSize: (splitted at: 14)
      size.

AmazonDecisionTreeLearning>>extractAttributes: reviewsDDD
    ^ reviewsDDD map: [:review | review extractFeatures ]
```

```
AmazonReview>>extractFeatures
    extractFeatures
    | featuresArray hv ss rs v |
    featuresArray := Array new: 3.
    hv := self discreteHelpfulVotes.
    ss := self discreteStars.
    rs := self discreteReviewSize.
     featuresArray at: 1 put: hv.
    featuresArray at: 2 put: ss.
    featuresArray at: 3 put: rs.
    ^ featuresArray.

AmazonReview>>discreteHelpfulVotes
    | features hv lower higher |
    features := AmazonFeatures new addAll: { 0 . 5 . 10 . 25 . 50 . 100 } ;
       yourself.
```

```
16    hv := helpfulVotes asInteger.
17    lower := ((features select: [:e | e < hv ]) ifEmpty: [ { features first } ] )
        last .
18    higher := ((features select: [:e | e >= hv ]) ifEmpty: [ { features last } ])
        first.
19    features measure: (((hv − lower) >( higher − hv)) ifTrue: [higher
20        ] ifFalse: [ lower ]).
21    ^ features
22
23  AmazonReview>>discreteStars
24    | features |
25    features := AmazonFeatures new addAll: { 1 . 2 . 3 . 4 . 5 } ; yourself.
26    features measure: stars asInteger.
27    ^ features
28
29  AmazonReview>>discreteReviewSize
30    | features rs lower higher |
31    features := AmazonFeatures new addAll: { 0 .100 . 250 . 500 . 1000 } ;
        yourself.
32    rs := helpfulVotes asInteger.
33    lower := ((features select: [:e | e < rs ]) ifEmpty: [ { features first } ] )
        last .
34    higher := ((features select: [:e | e >= rs ]) ifEmpty: [ { features last } ])
        first.
35    features measure: (((rs − lower) >( higher − rs)) ifTrue: [higher
36        ] ifFalse: [ lower ]).
37    ^ features
```

```
1   AmazonDecisionTreeLearning>>id3WithData: data target: target attributes:
        aCollection
2     | igs maxAttribute tree possibleValues filtered subtree |
3     (((data count < threshold) or: [ (self calculateHSample: (data take: 1)
        first idx: target  data: data ) = 0] ) or: [aCollection isEmpty] )ifTrue: [
4       ^ DecisionTreeLeaf new leafValue: (self mostCommonValueForData:
        data attribute: target)
5       ].
6     igs := self calculateIgOnFeatures: data.
7     ((1 to: igs size) select: [:e |( aCollection keys includes: e ) not]) do: [:ee |
        igs at: ee put: 0 ].
8     maxAttribute := self maxIndexOf: igs.
9     tree := DecisionTree new attribute: maxAttribute.
10    possibleValues := aCollection at: maxAttribute.
11    possibleValues do: [:value |
```

```
12      filtered := (data filter: [:d | (d at: maxAttribute ) measure = value ])
        execute.
13      subtree := self id3WithData: filtered target: maxAttribute  attributes:
         (aCollection copy removeKey: maxAttribute ; yourself) value: value.
14      tree addSubTree: subtree value: value.
15       ].
16    ^ tree.
```

Appendix C

Running the Experiments

1. After unzipping the `BigDataUserStudy.zip` file, run Pharo by opening a Terminal in the extracted folder, and executing `./pharo-ui Pharo.image`.

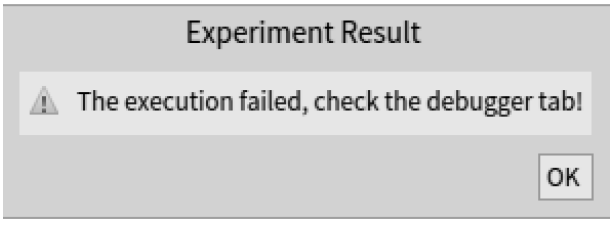
2. You will see a playground containing lines for running the assignments. Select the line for this assignment and right click and select "Do It" (or ctrl/cmd+d).

3. You will see a Spa UI open as well as a Transcript which shows the progress of Spa). Once the transcript shows "Connected to master!" you are ready to start.

4. Execute (ctrl/cmd+d) or inspect (ctrl/cmd+i) the two lines that you find in the Spa Evaluator playground. This will start the experiment.

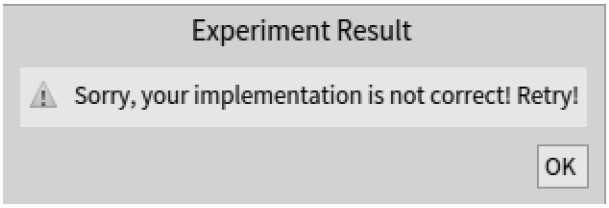
C.1 Expected Results

After the first run of the experiment, you should see an error dialog that looks like this:



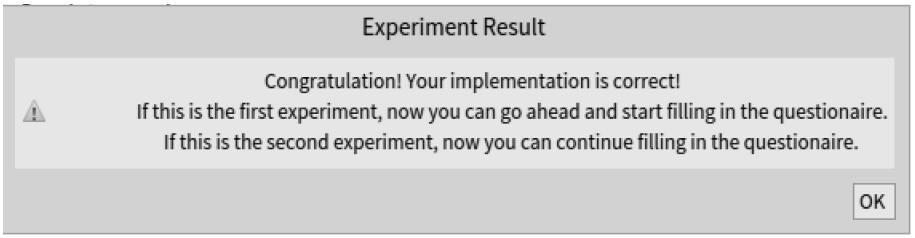
This means that the application thew an exception. At this point you should open the debugger tab, and debug the exception.

To fix the first bug, you may have a look at the corresponding cheat sheet for information on the operations and code updating functionalities of the debugger. When you fix the first bug, the following dialog will appear:



This means that there was no exception, but the final result was not the expected one (because you still did not solve bug 2). You can now inspect the `result` variable to check this.

When you fix bug 2, the experiment is succeeded and the following dialog will appear:



As the dialog suggests, please fill in the questionnaire.

Appendix D

User Study Cheatsheets

In this appendix we show the two cheatsheets (one for Debugger A, SpaDebug, and one for Debugger B, BigDebug) that were given to the participants of our user study.

Spa

As Spark, Spa uses a distributed data structure to represent data, and a set of functional transformations and actions to process data from one or more distributed data structures.

Spa VS Spark	
spa	instance of a SpaRunner, equivalent to a SparkContext
SpaDDD	a distributed data structure, equivalent to a Spark RDD
ddd	instance of a SpaDDD

Distributing Data/Files: Return a SpaDDD	
spa distribute: (1 to: 10)	distributes a collection containing numbers 1 through 10
spa textFile: '/Path/file'	distributes the lines of a file

Transformations are lazy, and are pipelined until an *action* triggers their execution.

Transformations: Return a new SpaDDD	
ddd map: [:e | e + 1]	increases by 1 each element of ddd
ddd filter: [:e | e > 100]	retains only the elements bigger than 100
ddd flatMap: [:e | e + 1]	increases by 1 each element of ddd, flattening collections

Actions: Return a value or collection	
ddd getCollection	returns all the values of the distributed data structure (equivalent to spark `collect`)
ddd execute	returns a new DDD after forcing all the transformations to happen (akin to Spark's `persist`)
ddd count	returns the total size of the distributed collection
ddd reduce: [:e1 :e2 | e1 + e2]	applies a distributed reduce to return the sum all the elements of the SpaDDD
ddd groupByKey	returns a new SpaDDDwhere data is grouped by key
ddd reduceByKey: [...]	groups data by key, and reduces a closure (see reduce:)

Debugger A

When there is an error in the parallel execution, you can use the Debugger A to debug it by accessing the 'Debugger' tab.

How to Debug

Select an exception in the left pane, and select a record in the right pane. Use **Debug Selected** to debug the exception generated by the selected record. The debugger offers the traditional *fine-grained stepping* operations e.g. step into, step over, etc. Recall that one can navigate to any of the stack-frames and inspect state, and restart the execution from there. The debugger also offers *coarse grained stepping* operations:

Step to next element	Steps to the evaluation of the next element in the collection
Step to next transformation	If the current transformation is followed by another one, it steps to its execution
Step to action result	Let's the execution continue and inspects the final result

How to Update Code (Code Updating)

Go to the Code Manager tab. Browse the changes and click on the **Commit Changes** button to propagate them to the workers. Right click on the exception, and run **Replay with new code** to restart the execution with the new code.

Breakpoints

`self halt` opens a debugger when the execution reaches that point. If can be placed within transformations, the remote execution stops and a Spa debugger opens for you.

The Debugging Modes

Debug When an exception happens in a worker, the worker stops and reports the exception to the debugger.

Debug & Continue When an exception happens in a worker, it puts the failure-inducing record aside and keeps executing. The exception is reported to the debugger, and awaits for you to either *skip* the record in the debugger view (right click on the record), or to change the code and use *Resume with new code* from the Code Manager.

Ignore Ignores errors up to a given threshold. The execution continues without the failure-inducing records.

Debugging on Virtual Partitions

The *debug all failed* and *debug on merged collection* buttons enables debugging of the failed execution using a partition with all the failed records, or with all records in all the failed partitions.

Spa: a Spark-like framework for Pharo Smalltalk. User Study Cheatsheet

Spa is a Spark-like framework written and running on Pharo Smalltalk (Version 8.x). In this leaflet, we provide a cheatsheet with basic Pharo syntax and concepts [1], augmented with a description of the Spa API and of the debugger.

Pharo

Minimal Syntax

Reserved Words	
nil	the undefined object
true,false	the boolean objects
self	the receiver of the current message
super	the receiver but for accessing overridden methods

Object constructors & reserved syntactic constructs	
"comment"	
'string'	sequence of characters
#symbol	unique string
$a, Character space	two ways to create characters
12 2r1100 16rC	twelve (decimal, binary, hexa)
3.14 1.2e3	floating-point numbers
#(abc 123)	literal array with the symbol #abc and the number 123
{foo . 3+2}	dynamic array built from 2 expressions
| foo bar |	declaration of two temporary variables
var := *expr*	assignment
exp1. *exp2*	period - statement separator
;	semicolon - message cascade
[:p | *expr*]	code block with a parameter
^ *expr*	caret - return/answer a result from a method

[1] This cheatsheet is based on the original Pharo cheatsheet available at https://github.com/pharo-project/pharo-flyers

Message Sending

When we send a message to an object (the *receiver*), the corresponding method is selected and executed, and the method answers an object. Message syntax mimics natural languages, with a subject, a verb, and complements.

Pharo	Java
aColor r: 0.2 g: 0.3 b: 0	aColor.setRGB(0.2,0.3,0)
d at: '1' put: 'Chocolate'.	d.put("1", "Chocolate");

Three Types of Messages: Unary, Binary, and Keyword

A **unary message** has no arguments.

```
Array new.        ⇝ anArray
#(4 2 1) size.    ⇝ 3
```

`new` is an unary message sent to classes (classes are objects).

A **binary message** takes only one argument and is named by one or more symbol characters from +, −, *, =, <, >, ...

```
3 + 4                ⇝ 7
'Hello' , ' World'   ⇝ 'Hello World'
```

The + message is sent to the object 3 with 4 as argument. The string `'Hello'` receives the message `,` (comma) with `' World'` as the argument.

A **keyword message** can take one or more arguments that are inserted in the message name.

```
'Pharo' allButFirst: 2.      ⇝ 'aro'
 [:x | x + 2 ] value: 7      ⇝ 9
3 to: 10 by: 2.              ⇝ (3 to: 10 by: 2)
```

The second line executes a block. The third example sends `to:by:` to 3, with arguments 10 and 2; this returns an interval containing 3, 5, 7, and 9.

Message Precedence

Parentheses > unary > binary > keyword, and finally from left to right.

```
(15 between: 1 and: 2 + 4 * 3) not    ⇝ false
```

Messages + and * are sent first, then `between:and:` is sent, and `not`. The rule suffers no exception: operators are just binary messages with *no notion of mathematical precedence*. 2 + 4 * 3 reads left-to-right and gives 18, not 14!

Blocks

Blocks are objects containing code that is executed on demand. They are the basis for control structures: conditionals & loops.

```
2 = 2
  ifTrue: [ Error signal: 'Help' ].
```

Send the message `ifTrue:` to the boolean `true` (computed from `2 = 2`) with a block as argument. Because the boolean is `true`, the block is executed and an exception is signaled.

Common Constructs

Conditionals

condition
  ifTrue: [action]
  ifFalse: [anotherAction]
	if (condition)
	  { action(); }
	  else { anotherAction(); }

[condition] whileTrue:
  [action. anotherAction]
	while (condition) { action();
	  anotherAction(); }

Loops/Iterators

1 to: 11 do: [:i |
  Transcript show: i ; cr]
	for(int i=1; i<11; i++){
	  System.out.println(i); }

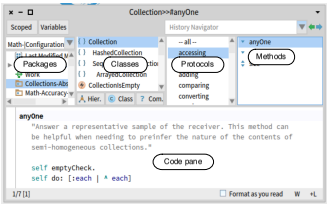
| names |
names := #('A' 'B' 'C').
names do: [:each |
  Transcript show: each, ' , ']
	String [] names ={"A", "B", "C"};
	for(String name : names) {
	  System.out.print(name);
	  System.out.print(","); }

Collections start at 1. `aCol at:i` accesses element at `i` and `aCol at:i put:value` sets element at `i` to `value`.

Collections

#(4 2 1) at: 3	⇝ 1
{4 . 2 . 1} at: 3 put: 6	⇝ #(4 2 6)
{4 . 2 . 1} collect: [:e | e + 1]	⇝ #(5 3 2)
{4 . 2 . 1} select: [:e | e > 2]	⇝ #(4)
{4 . 2 . 1} reject: [:e | e > 2]	⇝ #(2 1)
{4 . 2 . 1} detect: [:e | e > 2]	⇝ #4
(OrderedCollection new) add: 4; add: 2 ; yourself	⇝ #aOC(4 2)
dict := Dictionary new at: #a put: 'Alpha' ; yourself	⇝a Dict(#a->'Alpha')
dict at: #a	⇝ 'Alpha'
dict at: #b ifAbsent: ['Abs']	⇝ 'Abs'

The 5 Panes Pharo Code Browser



- The *packages* pane shows all the packages of the system.
- The *classes* pane shows the class hierarchy of the selected package; the *class side* checkbox allows for getting the methods of the metaclass.
- The *protocols* pane groups the methods of the selected class to ease navigation.
- The *methods* pane lists the methods of the selected protocol; icons are clickable and trigger special actions;
- The *source code* pane shows the source code of the selected method.

Methods

Methods are public and virtual. They are always looked up in the class of the receiver. By default a method returns `self`. Class methods follow the same dynamic lookup as instance methods. Method `factorial` defined in class Integer.

```
Integer >> factorial
    "Answer the factorial of the receiver."
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
```

Executing and Accepting Code

From anywhere, you can select code and run it, inspect it, or print its result. All of these options are visible by right clicking on selected code. When you change a method, for the change to be effective you have to accept the new method by doing right-click and clicking on accept anywhere in the method. All of the above can be also done through the shortcuts presented below.

ctrl/cmd + **i**	Inspect (run the code and inspect the result)
ctrl/cmd + **d**	Do It (run the code)
ctrl/cmd + **p**	Print it (run the code and print the result)
ctrl/cmd + **s**	Accept (compiles and stores the new code)

Spa

As Spark, Spa uses a distributed data structure to represent data, and a set of functional transformations and actions to process data from one or more distributed data structures.

Spa VS Spark	
spa	instance of a SpaRunner, equivalent to a SparkContext
SpaDDD	a distributed data structure, equivalent to a Spark RDD
ddd	instance of a SpaDDD

Distributing Data/Files: Return a SpaDDD	
spa distribute: (1 to: 10)	distributes a collection containing numbers 1 through 10
spa textFile: '/Path/file'	distributes the lines of a file

Transformations are lazy, and are pipelined until an *action* triggers their execution.

Transformations: Return a new SpaDDD	
ddd map: [:e | e + 1]	increases by 1 each element of ddd
ddd filter: [:e | e > 100]	retains only the elements bigger than 100
ddd flatMap: [:e | e + 1]	increases by 1 each element of ddd, flattening collections

Actions: Return a value or collection	
ddd getCollection	returns all the values of the distributed data structure (equivalent to spark `collect`)
ddd execute	returns a new DDD after forcing all the transformations to happen (akin to Spark's `persist`)
ddd count	returns the total size of the distributed collection
ddd reduce: [:e1 :e2 | e1 + e2]	applies a distributed reduce to return the sum all the elements of the SpaDDD
ddd groupByKey	returns a new SpaDDDwhere data is grouped by key
ddd reduceByKey: [...]	groups data by key, and reduces a closure (see reduce:)

Debugger B

When there is an error in the parallel execution, you can use the Debugger B to debug it by accessing the 'Debugger' tab.

How to Debug

You can select an offending record from the right panel, decide to **skip it** or **modify it** . To modify the record, change it in its string representation, and then click on *Modify*. This will restart that particular execution, which will complete or fail (check the Transcript for the output).

How to Update Code (Code Patching)

Double click on a transformation in the left panel, and, through the dialog that appears, you can insert a new closure to be applied instead of the original one. Once you fixed the code, you can use **Resume** or **Step Over** to remotely guide the execution.

Alternatively, you can **Redeploy** to restart your workers with the updated code. If you do this, however, you will have to re-execute the application.

Watchpoints and Breakpoints

If you want to analyze which are the values at a particular transformation, you can call `watchpoint:`, passing a guard as parameter, to get feedback on which values pass the guard.

To add a simulated breakpoint, call `simulatedBreakpoint` **after** a `watchpoint:`. This will generate a simulated breakpoint for each value that passes the guard of the watchpoint.

The Debugging Mode

When an exception happens in a worker, it puts it "aside" it and keeps executing. The exception is reported to the debugger, and the execution awaits for you to *skip*, `modify` that record in the debugger view. You can also change the closure that is applied and resume that particular breakpoint by executing the steps to update the code (see above) and then resume it.

Spa: a Spark-like framework for Pharo Smalltalk. User Study Cheatsheet

Spa is a Spark-like framework written and running on Pharo Smalltalk (Version 8.x). In this leaflet, we provide a cheatsheet with basic Pharo syntax and concepts [1], augmented with a description of the Spa API and of the debugger.

Pharo

Minimal Syntax

Reserved Words	
nil	the undefined object
true,false	the boolean objects
self	the receiver of the current message
super	the receiver but for accessing overridden methods

Object constructors & reserved syntactic constructs	
"comment"	
'string'	sequence of characters
#symbol	unique string
$a, Character space	two ways to create characters
12 2r1100 16rC	twelve (decimal, binary, hexa)
3.14 1.2e3	floating-point numbers
#(abc 123)	literal array with the symbol #abc and the number 123
{foo . 3+2}	dynamic array built from 2 expressions
| foo bar |	declaration of two temporary variables
var := *expr*	assignment
exp1. *exp2*	period - statement separator
;	semicolon - message cascade
[:p | *expr*]	code block with a parameter
^ *expr*	caret - return/answer a result from a method

[1] This cheatsheet is based on the original Pharo cheatsheet available at https://github.com/pharo-project/pharo-flyers

Message Sending

When we send a message to an object (the *receiver*), the corresponding method is selected and executed, and the method answers an object. Message syntax mimics natural languages, with a subject, a verb, and complements.

Pharo	Java
aColor r: 0.2 g: 0.3 b: 0	aColor.setRGB(0.2,0.3,0)
d at: '1' put: 'Chocolate'.	d.put("1", "Chocolate");

Three Types of Messages: Unary, Binary, and Keyword

A **unary message** has no arguments.

Array new.	⇝ anArray
#(4 2 1) size.	⇝ 3

`new` is an unary message sent to classes (classes are objects).

A **binary message** takes only one argument and is named by one or more symbol characters from +, −, *, =, <, >, ...

3 + 4	⇝ 7
'Hello' , ' World'	⇝ 'Hello World'

The + message is sent to the object 3 with 4 as argument. The string `'Hello'` receives the message `,` (comma) with `' World'` as the argument.

A **keyword message** can take one or more arguments that are inserted in the message name.

'Pharo' allButFirst: 2.	⇝ 'aro'
[:x | x + 2] value: 7	⇝ 9
3 to: 10 by: 2.	⇝ (3 to: 10 by: 2)

The second line executes a block. The third example sends `to:by:` to 3, with arguments 10 and 2; this returns an interval containing 3, 5, 7, and 9.

Message Precedence

Parentheses > unary > binary > keyword, and finally from left to right.

(15 between: 1 and: 2 + 4 * 3) not	⇝ false

Messages + and * are sent first, then `between:and:` is sent, and `not`. The rule suffers no exception: operators are just binary messages with *no notion of mathematical precedence*. 2 + 4 * 3 reads left-to-right and gives 18, not 14!

Blocks

Blocks are objects containing code that is executed on demand. They are the basis for control structures: conditionals & loops.

```
2 = 2
  ifTrue: [ Error signal: 'Help' ].
```

Send the message `ifTrue:` to the boolean `true` (computed from `2 = 2`) with a block as argument. Because the boolean is `true`, the block is executed and an exception is signaled.

Common Constructs

Conditionals

condition
  ifTrue: [action]
  ifFalse: [anotherAction]
	if (condition)
	  { action(); }
	  else { anotherAction(); }

[condition] whileTrue:
  [action. anotherAction]
	while (condition) { action();
	  anotherAction(); }

Loops/Iterators

1 to: 11 do: [:i |
  Transcript show: i ; cr]
	for(int i=1; i<11; i++){
	  System.out.println(i); }

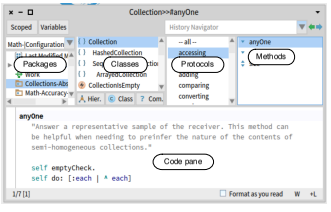
| names |
names := #('A' 'B' 'C').
names do: [:each |
  Transcript show: each, ' , ']
	String [] names ={"A", "B", "C"};
	for(String name : names) {
	  System.out.print(name);
	  System.out.print(","); }

Collections start at 1. `aCol at:i` accesses element at `i` and `aCol at:i put:value` sets element at `i` to `value`.

Collections

#(4 2 1) at: 3	⇝ 1
{4 . 2 . 1} at: 3 put: 6	⇝ #(4 2 6)
{4 . 2 . 1} collect: [:e | e + 1]	⇝ #(5 3 2)
{4 . 2 . 1} select: [:e | e > 2]	⇝ #(4)
{4 . 2 . 1} reject: [:e | e > 2]	⇝ #(2 1)
{4 . 2 . 1} detect: [:e | e > 2]	⇝ #4
(OrderedCollection new) add: 4; add: 2 ; yourself	⇝ #aOC(4 2)
dict := Dictionary new at: #a put: 'Alpha' ; yourself	⇝a Dict(#a->'Alpha')
dict at: #a	⇝ 'Alpha'
dict at: #b ifAbsent: ['Abs']	⇝ 'Abs'

The 5 Panes Pharo Code Browser



- The *packages* pane shows all the packages of the system.
- The *classes* pane shows the class hierarchy of the selected package; the *class side* checkbox allows for getting the methods of the metaclass.
- The *protocols* pane groups the methods of the selected class to ease navigation.
- The *methods* pane lists the methods of the selected protocol; icons are clickable and trigger special actions;
- The *source code* pane shows the source code of the selected method.

Methods

Methods are public and virtual. They are always looked up in the class of the receiver. By default a method returns `self`. Class methods follow the same dynamic lookup as instance methods. Method `factorial` defined in class Integer.

```
Integer >> factorial
    "Answer the factorial of the receiver."
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
```

Executing and Accepting Code

From anywhere, you can select code and run it, inspect it, or print its result. All of these options are visible by right clicking on selected code. When you change a method, for the change to be effective you have to accept the new method by doing right-click and clicking on accept anywhere in the method. All of the above can be also done through the shortcuts presented below.

ctrl/cmd + **i**	Inspect (run the code and inspect the result)
ctrl/cmd + **d**	Do It (run the code)
ctrl/cmd + **p**	Print it (run the code and print the result)
ctrl/cmd + **s**	Accept (compiles and stores the new code)

Appendix E

User Study Questionnaire

After completing each of the assignments, the participants were instructed to complete a questionnaire. Since the study included groups that participated in presence and groups that participated online, we administered the questionnaire in the format of a google form. This allowed us to automatically collect the data for both kinds of groups. Below we list the different questions with the possible answers. Some questions were open, thus are indicated as open questions.

The first part of the questionnaire asks questions about the participant. The second part includes general questions asked after completing each assignment. The two debugging sections (debugging with debugger A and debugging with debugger B) were shown after completing the general questions about the assignment and depending on which debugger they have used during the assignment. Finally, the overall questions section was always asked at the end, after they had experienced both experimental conditions.

About you

What's your experience in developing parallel and/or distributed programs?	1 2 3 4 5
Which programming language do you have the most experience with?	Multiple Choice
Do you normally use a debugger while developing a program	1 2 3 4 5
Which debugger (if any) do you use the most?	Open question
What debugging technique do you normally use?	Multiple choice
Which group are you part of?	X Y

General questions (for each assignment)

Debugging this application was difficult	1 2 3 4 5
I would have liked more time to complete the assignment	Yes No
I found the first bug	Yes No
I found the second bug	Yes No

Debugging with Debugger A

Debugging the application was difficult	1 2 3 4 5
The debugger helped me to identify the cause of the bugs	1 2 3 4 5
Fine grained stepping is useful to debug a parallel execution	1 2 3 4 5
Coarse grained stepping is useful to debug a parallel execution	1 2 3 4 5
I have used the ignore functionality	Yes No
The ignore functionality is useful to avoid data-cleaning bugs	1 2 3 4 5
I have used the live code updating functionality	Yes No
The live code updating functionality is useful to avoid replaying the application	1 2 3 4 5
I had to redeploy the program the following amount of times	0 1 2 3 4 5+
Which of the features did you find useful?	Multiple Choice

Debugging with Debugger B

Debugging the application was difficult	1 2 3 4 5
The debugger helped me to identify the cause of the bugs	1 2 3 4 5
Coarse grained stepping is useful to debug a parallel execution	1 2 3 4 5
I have used the substitute record functionality	Yes No
The substitute record functionality is useful to avoid data-cleaning bugs	1 2 3 4 5
I have used the code patching functionality	Yes No
The code patching functionality is useful to avoid replaying the application	1 2 3 4 5
I had to redeploy the program the following amount of times	0 1 2 3 4 5+
Which of the features did you find useful?	Multiple Choice

Overall experience (Open questions)

- Is there a feature of the other debugger that could have helped you to solve Assignment 1?
- Is there a feature of the other debugger that could have helped you to solve Assignment 2?
- Is there a feature or information that you felt was missing from both the debugger approaches?
- Did you experience some technical issue that hampered the debugging experience?

Bibliography

[Ama] Amazon. Amazon kinesis. `https://aws.amazon.com/kinesis/`. Accessed: 02-2022.

[Apaa] Apache. Apache giraph. `http://giraph.apache.org/`. Accessed: 02-2022.

[Apab] Apache. Apache kafka. `https://kafka.apache.org/`. Accessed: 02-2022.

[Apac] Apache. Apache pig. `https://pig.apache.org/`. Accessed: 02-2022.

[Apad] Apache. Apache spark. `http://spark.apache.org/`. Accessed: 02-2022.

[Apae] Apache. Apache storm. `https://storm.apache.org/`. Accessed: 02-2022.

[Apaf] Apache. Hadoop. `http://hadoop.apache.org`. Accessed: 02-2022.

[AZMT18] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[BH00] S. Bouchenak and D. Hagimont. Pickling threads state in the java system. In *Proceedings 33rd International Conference on Technology of Object-Oriented Languages and Systems TOOLS 33*, pages 22–32, 2000.

[BJC+13] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *University of Cambridge-Judge Business School, Tech. Rep*, 2013.

[BK19] Mehdi Bagherzadeh and Raffi Khatchadourian. Going big: a large-scale study on what big data developers ask. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 432–442, 2019.

[BL07] Jost Berthold and Rita Loogen. Visualizing parallel functional program runs: Case studies with the eden trace viewer. *Parallel Computing: Architectures, Algorithms and Applications. Advances in Parallel Computing*, 15:121–128, 2007.

[BMM+16] Earl T Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for JavaScript/Node.js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1003–1007. ACM, 2016.

[BNDP10] A.P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet. *Pharo by Example*. Open Textbook Library. Square Bracket Associates, 2010.

[BSSZ18] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 572–583. Association for Computing Machinery, 2018.

[CCRR13] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–9, 2013.

[CDGN15] Andrei Chis, Marcus Denker, Tudor Gorba, and Oscar Nierstrasz. Practical domain-specific debuggers using the moldable debugger framework. *Computer Languages, Systems & Structures*, 44:89 – 113, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[Chr15] Larry B. 1941 Christensen. *Research methods, design, and analysis*. Pearson Education Limited, England, 12th ed. edition, 2015.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[CKMR12] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 169–180. ACM, 2012.

[Con] Pharo Contributions. Taskit. `https://github.com/pharo-contributions/taskit`. Accessed: 02-2022.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.

[CS63] Donald T. Campbell and Julian C. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Rand McNally College Publishing, 1963.

[DCD13] Martin Dias, Damien Cassou, and Stéphane Ducasse. Representing code history with development environment events. *CoRR*, abs/1309.4334, 2013.

[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DPDA14] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Fuel: a fast general purpose object graph serializer. *Software: Practice and Experience*, 44(4):433–453, 2014.

[Dra13] Iulian Dragos. Stack Retention in Debuggers For Concurrent Programs. Presented at Scala Workshop 2013, July 2013.

[DZSS13] Ankur Dave, Matei Zaharia, Scott Shenker, and Ion Stoica. Arthur: Rich post-facto debugging for production analytics applications. *Technical report, University of California*, 2013.

[FB88] Stuart I Feldman and Channing B Brown. Igor: A system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988.

[FDCD12] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with Big Data Analytics. *ACM Interactions*, 1072(5220):50–59, 2012.

[Fou] The Python Software Foundation. The python debugger. `https://docs.python.org/2/library/pdb.html`. Accessed: 02-2022.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3):225–233, 1986.

[Gar] Gartner. Big data. `https://www.gartner.com/en/information-technology/glossary/big-data`. Accessed: 02-2022.

[Gil51] Stanley Gill. The diagnosis of mistakes in programmes on the edsac. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 206(1087):538–554, 1951.

[GIY+16] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proc. of the 38th Int. Conf. on Software Engineering (ICSE '16)*, pages 784–795. ACM, 2016.

[GKYL01] Jean-Pierre Goux, Sanjeev Kulkarni, Michael Yoder, and Jeff Linderoth. Master–Worker: An Enabling Framework for Applications on the Computational Grid. *Cluster Computing*, 4(1):63–70, 2001.

[GMMK19] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. White-box testing of big data analytics with complex user-defined functions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 290–301. Association for Computing Machinery, 2019.

[GNU] GNU. The gnu project debugger. `https://www.gnu.org/software/gdb/`. Accessed: 02-2022.

[GNV+11] Elisa Gonzalez Boix, Carlos Francisco Noguera Garcia, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt. Reme-d: a reflective, epidemic message-oriented debugger for ambient-oriented applications. In *SAC'11 The 2011 ACM Symposium on Applied Computing*, volume 2, pages 1275–1281. ACM, 4 2011.

[Gol84] Adele Goldberg. *SMALLTALK-80: The Interactive Programming Environment*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[Gooa] Google. Chrome devtools protocol. `https://chromedevtools.github.io/devtools-protocol/`. Accessed: 02-2022.

[Goob] Google. Dataflow. `https://cloud.google.com/dataflow`. Accessed: 02-2022.

[GSJ09] Allan Raundahl Gregersen, Douglas Simon, and Bo Nørregaard Jørgensen. Towards a dynamic-update-enabled jvm. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, pages 1–7, 2009.

[GSS19] Robbert Gurdeep Singh and Christophe Scholliers. Warduino: a dynamic webassembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 27–36, 2019.

[GWK18] Muhammad Gulzar, Siman Wang, and Miryung Kim. Bigsift: automated debugging of big data analytics in data-intensive scalable computing. In *Proc. of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 863–866, 10 2018.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 32–43, 1992.

[IMBGB20] Daniel Ingalls, Eliot Miranda, Clément Béra, and Elisa Gonzalez Boix. Two decades of live coding and debugging of virtual machines through simulation. *Software: Practice and Experience*, 50(9):1629–1650, 2020.

[IST+15] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in spark. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 9, page 216. NIH Public Access, 2015.

[JYB11] V. Jagannath, Z. Yin, and M. Budiu. Monitoring and debugging dryadlinq applications with daphne. In *2011 IEEE Int. Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1266–1273, Anchorage, AK, USA, May 2011.

[KHL10] Saparya Krishnamoorthy, Michael S Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *2010 19th IEEE Asian Test Symposium*, pages 59–64. IEEE, 2010.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[LCN17] Max Leske, Andrei Chiș, and Oscar Nierstrasz. Improving live debugging of concurrent threads through thread histories. *Science of Computer Programming*, 161, 11 2017.

[Lig] Lightbend. Akka. `https://akka.io/`. Accessed: 2022-01-27.

[LRS+13] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner. Sedge: Symbolic example data generation for dataflow programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 235–245. IEEE, 2013.

[Mar17] Matteo Marra. Idra: an out-of-place debugger for non-stoppable applications, 2017. Vrije Universiteit Brussel. Brussels, Belgium. `http://soft.vub.ac.be/Publications/2017/vub-soft-ms-17-01.pdf`.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[MGBC+17] Matteo Marra, Elisa Gonzalez Boix, Steven Costiou, Mickaël Kerboeuf, Alain Plantec, Guillermo Polito, and Stéphane Ducasse. Debugging cyber-physical systems with pharo: An experience report. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '17, pages 8:1–8:10. ACM, 2017.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, December 1989.

[Mica] Microsoft. Azure stream analytics. `https://azure.microsoft.com/en-us/services/stream-analytics/`. Accessed: 02-2022.

[Micb] Microsoft. Dryadlinq. https://www.microsoft.com/en-us/research/project/dryadlinq/. Accessed: 02-2022.

[Micc] Microsoft. Remote debugging. `https://msdn.microsoft.com/en-us/library/y7f5zaaa.aspx`. Accessed: 02-2022.

[Mit16] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4), March 2016.

[MLW+19] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q. Vera Liao, Casey Dugan, and Thomas Erickson. How data science workers work with data: Discovery, capture, curation, design, creation. In *Proc. of the 2019 CHI Conf. on Human Factors in Computing Systems*, CHI '19, page 1–15. Association for Computing Machinery, 2019.

[MPGB18] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. Out-of-place debugging: a debugging architecture to reduce debugging interference. *The Art, Science and Engineering of Programming*, 3(2):pp. 3:1–3:29, October 2018.

[MPGB20a] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. A debugging approach for live big data applications. *Science of Computer Programming*, 194:102460, 2020.

[MPGB20b] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. Framework-aware debugging with stack tailoring. In *Proc. of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, page 71–84. ACM, 2020.

[MPGB21] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. Practical online debugging of spark-like applications. In *To appear in Proceedings of the 21st IEEE International Conference on Software Quality, Reliability, and Security*, QRS 2021. IEEE, 2021.

[MVB+16] Armando Miraglia, Dirk Vogt, Herbert Bos, Andy Tanenbaum, and Cristiano Giuffrida. Peeking into the past: Efficient checkpoint-assisted time-traveling debugging. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 455–466. IEEE, 2016.

[NHSO06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 72–83. Association for Computing Machinery, 2006.

[Ora] Oracle. Jpda - java platform debugger architecture. https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/. Accessed: 02-2022.

[ORH02] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of java software. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 649–658. IEEE, 2002.

[Pac11] David Pacheco. Postmortem Debugging in Dynamic Environments. *Commun. ACM*, 54(12):44–51, 2011.

[PBFD15] Nick Papoulias, Noury Bouraqadi, Luc Fabresse, and Marcus Denker. Mercury: Properties and Design of a Remote Debugging Solution using Reflection. *Journal of Object Technology*, 14(2):36, 2015.

[PGS+11] Mario Pukall, Alexander Grebhahn, Reimar Schröter, Christian Kästner, Walter Cazzola, and Sebastian Götz. Javadaptor: Unrestricted dynamic software updates for java. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 989–991. Association for Computing Machinery, 2011.

[Por] Perl 5 Porters. The perl debugger. `http://perldoc.perl.org/perldebug.html`. Accessed: 02-2022.

[PT09] Guillaume Pothier and Éric Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26:78–85, 2009.

[PVH14] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Dsu for java on a stock jvm. *ACM SIGPLAN Notices*, 49(10):103–119, 2014.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[RCMBGB21] Carlos Javier Rojas Castillo, Matteo Marra, Jim Bauwens, and Elisa Gonzalez Boix. Extending a WebAssembly VM with Out-of-Place Debugging for IoT applications, 2021. Presented at VMIL '21, Chicago, October 2021.

[Rin03] Martin Rinard. Acceptability-oriented computing. In *Companion of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, page 221–239. ACM, 2003.

[RK13] Ariel Rabkin and Randy Katz. How hadoop clusters break. *IEEE Softw.*, 30(4):88–94, July 2013.

[RPM18] Manuel Rigger, Daniel Pekarek, and Hanspeter Mössenböck. Context-aware failure-oblivious computing as a means of preventing buffer overflows. In *Network and System Security*, pages 376–390, Cham, 2018. Springer International Publishing.

[SBSB19] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. Reasoning about the node.js event loop using async graphs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 61–72. IEEE Press, 2019.

[SCC02] W R Shadish, Thomas D Cook, and D T Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. MA: Houghton Mifflin, 2002.

[SCM09] Terry Stanley, Tyler Close, and Mark S. Miller. Causeway: a message-oriented distributed debugger. Technical report, HP Labs, 2009. HP Labs tech report HPL-2009-78.

[Sen07] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[SSK+15] Semih Salihoglu, Jaeho Shin, Vikesh Khanna, Ba Quan Truong, and Jennifer Widom. Graft: A debugging tool for apache giraph. In *Proc. of the 2015 ACM SIGMOD Int. Conf. on Management of Data*, SIGMOD '15, pages 1403–1408. ACM, 2015.

[SW17] Kazuhiro Shibanai and Takuo Watanabe. Actoverse: A reversible debugger for actors. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, pages 50–57. ACM, 2017.

[Sye14] Basarat Ali Syed. Simplifying callbacks. In *Beginning Node. js*, pages 197–224. Springer, 2014.

[TLBS+17] Carmen Torres Lopez, Elisa Gonzalez Boix, Christophe Scholliers, Stefan Marr, and Hanspeter Mössenböck. A principled approach towards debugging communicating event-loops. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, page 41–49. Association for Computing Machinery, 2017.

[TPB+18] Pablo Tesone, Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, and Luc Fabresse. Dynamic software update from development to production. *The Journal of Object Technology*, 17:1:1, 11 2018.

[TZ18] Wei Tang and Min Zhang. Pyreload: dynamic updating of python programs by reloading. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 229–238. IEEE, 2018.

[Wis97] Roland Wismüller. Debugging message passing programs using invisible message tags. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 295–302. Springer, 1997.

[WPP+14] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108. ACM, 2014.

[WZL+14] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th Int. Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014.

[XZ+10] Xu Xiao, Xiao-song Zhang, et al. New approach to path explosion problem of symbolic execution. In *2010 First International Conference on Pervasive Computing, Signal Processing and Applications*, pages 301–304. IEEE, 2010.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[Zel09] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[Zin] Zinc. Zinc. `http://zn.stfx.eu/zn/index.html`. Accessed: 02-2022.

[ZLZ+15] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. An empirical study on quality issues of production big data platform. In *2015 IEEE/ACM*

37th IEEE International Conference on Software Engineering, volume 2, pages 17–26. IEEE, 2015.

[ZM19] Long Zhang and Martin Monperrus. Tripleagent: Monitoring, perturbation and failure-obliviousness for automated resilience improvement in java applications. pages 116–127, 10 2019.

[ZWG+20] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 722–733, 2020.