Framework-Aware Debugging with Stack Tailoring

Matteo Marra
mmarra@vub.be
Vrije Universiteit Brussel, Belgium

Guillermo Polito
guillermo.polito@univ-lille.fr
Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 , CRIStAL, Lille, France

Elisa Gonzalez Boix
egonzale@vub.be
Vrije Universiteit Brussel, Belgium

Abstract

Debugging applications that execute within a framework is not always easy: the call-stack offered to developers is often a mix-up of stack frames that belong to different frameworks, introducing an unnecessary noise that prevents developers from focusing on the debugging task. Moreover, relevant application code is not always available in the call-stack because it may have already returned, or is available in another thread. In such cases, manually gathering all relevant information from these different sources is not only cumbersome but also costly.

In this paper we introduce Sarto, a call-stack instrumentation layer that allows developers to tailor the stack to make debugging framework-aware. The goal is to improve the quality and amount information present in the call-stack to reduce debugging time without impacting the execution time. Sarto proposes a set of six stack operations that combined hide irrelevant information, introduce missing information, and relate dispersed debugging sources before this is fed to the debugger.

We validate Sarto by applying it to four application cases using inherently different frameworks: unit testing, web server, remote promises and big data processing. We showcase our experiences in using Sarto in the different frameworks, and perform some performance benchmarks to demonstrate that Sarto does not generate noticeable overhead when instrumenting a call-stack. We also show that our solution reduces by half the amount of data stored to debug similar exceptions happening in a parallel setup.

CCS Concepts: • **Software and its engineering** → *Application specific development environments*; *Integrated and visual development environments*; *Error handling and recovery*; **Software testing and debugging**.

DLS '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8175-8/20/11…$15.00
https://doi.org/10.1145/3426422.3426982

Keywords: stack manipulation, domain specific debugging, debugging support

ACM Reference Format:
Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2020. Framework-Aware Debugging with Stack Tailoring. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3426422.3426982

1 Introduction

Applications often use libraries and frameworks to solve common tasks and focus on the application's concern. Libraries and frameworks 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).

Using an online debugger to debug applications that execute within a framework is not always easy: an online debugger presents the developer with a call-stack representing a chain of method activations, also known as contexts or stack frames, containing runtime information (e.g., the values of variables, and the exact methods/classes/functions found during execution). Although for many programs inspecting the call-stack is enough to find the cause of a bug, applications using frameworks find their call-stacks polluted with framework-specific code: application frames and framework frames are interleaved on the stack. Developers then need to understand how to read and navigate such call-stacks, a problem that is worsened by the mix and match of many frameworks in a single application, or the parallel and distributed nature of such frameworks.

Much work in literature proposes debugging solutions tailored to single frameworks. For instance, there are different approaches for debugging actor execution [2, 11, 17], or for debugging distributed parallel execution [9, 13]. Other solutions, such as the moldable debugger [4], provide a more general approach to enable domain-specific debugging by offering different views on the call-stack, that are customizable by developers. This is also a common approach taken by Integrated development environments (IDEs) to offer debugging support to a variety of languages (e.g., Eclipse[1], Chrome DevTools[2]). Modern IDEs typically communicate with a debugging instrumentation layer on top of the runtime system

[1] http://www.eclipse.org/eclipse/debug
[2] https://developers.google.com/web/tools/chrome-devtools

via a debugger protocol. In this way, if a different debugger needs to be attached, the runtime does not have to be changed. This allows IDEs like Eclipse to offer different views to a call-stack and even to filter stack frames based on predefined properties like packages. However, these solutions are either not applicable to different domains, or they are specific to a tool.

In this paper we revisit 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 framework(s) they are using, and has the possibility to dive in the original call-stack in case they are interested in the framework's code. To perform such 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.

We implemented Sarto in Pharo Smalltalk, and we applied it to four different frameworks: HTTP, unit testing, promises, and Big Data parallel execution.

We present a two-fold evaluation of our approach. First, we show our experience in adapting a complex framework to Sarto, and in debugging a tailored application. Second, we conduct performance benchmarks to show that our approach is still practical and efficient. Our benchmarks show that Sarto is also a practical approach, since it does not introduce noticeable overhead during normal debugging, and that using delta stacks instead of full call-stacks improves debugger performance in parallel and distributed applications.

2 Motivation

Despite the fact that stack traces are key in understanding the execution of a program, finding the root cause of bugs remains difficult. 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 such a call-stack because it may be contained in methods that already returned or that were executed in another thread. Finally, when debugging parallel or distributed applications, important debugging information is scattered in different call-stacks that users need to manually relate. This section details such issues by means of four different use cases.

2.1 Case 1: Debugging Web Servers

Web servers are frameworks that listen 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 the Web server's stack frames below the application's frames. Figure 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.

SmallInteger	/
UndefinedObject	DoIt
ZnValueDelegate	handleRequest:
ZnManagingMultiThreadedServer(ZnSingleThreadedSe	authenticateAndDelegateRequest:
ZnManagingMultiThreadedServer(ZnSingleThreadedSe	authenticateRequest:do:
ZnManagingMultiThreadedServer(ZnSingleThreadedSe	authenticateAndDelegateRequest:
ZnManagingMultiThreadedServer(ZnSingleThreadedSe	handleRequestProtected:
BlockClosure	on:do:
ZnManagingMultiThreadedServer(ZnSingleThreadedSe	handleRequestProtected:
BlockClosure	on:do:
ZnManagingMultiThreadedServer(ZnSingleThreadedSe	handleRequestProtected:
ZnManagingMultiThreadedServer(ZnSingleThreadedSe	handleRequest:timing:
ZnManagingMultiThreadedServer(ZnMultiThreadedSer	executeOneRequestResponseOn:
ZnManagingMultiThreadedServer(ZnMultiThreadedSer	executeRequestResponseLoopOn:
ZnCurrentServer(DynamicVariable)	value:during:
BlockClosure	ensure:
ZnCurrentServer(DynamicVariable)	value:during:
ZnCurrentServer class(DynamicVariable class)	value:during:
ZnManagingMultiThreadedServer(ZnMultiThreadedSer	executeRequestResponseLoopOn:
ZnManagingMultiThreadedServer(ZnMultiThreadedSer	serveConnectionsOn:
BlockClosure	ensure:
ZnManagingMultiThreadedServer(ZnMultiThreadedSer	serveConnectionsOn:
BlockClosure	ifCurtailed:
ZnManagingMultiThreadedServer(ZnMultiThreadedSer	serveConnectionsOn:
BlockClosure	newProcess

Figure 1. Call stack when debugging a failing HTTP server in Pharo

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. Debuggers in mainstream IDEs (e.g., Eclipse's debugger) offer filtering operations to hide such framework frames in the stack. Such filters are either based on a number of 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 given packages) or delegated to the application developer, who may not have enough knowledge to do it correctly.

It is the framework developer who has the knowledge about the framework internals, but she has no control on how the stack should be shaped when debugged.

2.2 Case 2: Debugging Unit Tests

Consider a unit test being explicitly called by the developer for debugging purposes. The test fails and the debugger stops on the failing assertion. When debugging this unit

test execution, developers are exposed to a call-stack that interleaves application frames and framework frames: the bottom of the stack has application code calling the testing framework, followed by frames of the testing framework, and finally at the top of the stack there is the failing test frame. While the framework frames are not interesting to the application developer, there is one particular method that could increase the amount of information for debugging: the `setup` method. 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.

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).

2.3 Case 3: Debugging Promise Executions

Promises are a recurrent programming abstraction in concurrent languages to reconcile asynchronous communication with return values [3]. Many mainstream languages have adopted support for promises, *e.g.,* Java, .NET, Scala and more recently JavaScript. A promise represents the result of an asynchronous operation that may, or may not, execute on a different thread. When such execution succeeds, the promise is resolved with a value; if it fails, the promise is marked as failed. Developers add callbacks to handle both succeeding and failing resolutions.

Consider a promise created as in Listing 1. The developer creates a promise from a closure. The promise is executed on a different thread and is resolved later, possibly with a failure. In the example, the promise does a division by the argument `n`, and a callback is added to intercept failures and open a debugger.

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

PromiseRunner new promiseDivision: 0.
```

Listing 1. A failing promise

Let us consider that a promise is created capturing the value `n = 0`. When debugging this failed promise, developers find themselves with a stack that includes only the promise execution, and not the frames that lead to the creation of the promise, so they cannot trace back the origin of the value zero. This happens because the promise execution and the promise creation happen in different threads with different call-stacks. Moreover, the stack frame that created the promise is not available anymore because it returned right after creating the promise.

In this case, the debugger is again missing information that can be **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 lead to different domain-specific debugging techniques whose goal is to reconstruct causal relations in asynchronous communication to offer an *asynchronous track trace* [7, 12, 18]. In this paper, we explore a more general solution to relate information present on different stacks.

2.4 Case 4: Debugging Parallel Executions

Consider again an error happening multiple times when resolving certain HTTP requests in an HTTP server. HTTP servers are classically resilient, and will provide a *500 Internal Server Error* as response, staying available to resolve the next request. When an internal server error happens multiple times, debugging those problems requires understanding how those errors are related, as their cause may be related. 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, combine them, debug them singularly, or to abstract information from all of the different exceptions. The information about the bug is again **dispersed** over different similar call-stacks.

A similar issue happens 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.

In both cases, the debugging information is **dispersed** over many call-stacks. To solve this problem, Marra et al. [13] propose a debugging technique tailored for Map/Reduce that aggregates similar exceptions happening in a parallel Map/Reduce application. In this work, we build on that solution and generalize it to other frameworks.

2.5 Problem Statement

As mentioned in the introduction, modern IDEs are built around the concept of a common protocol to decouple the IDE and runtime systems, facilitating the construction of new tools. In particular, debuggers communicate via a *debugger protocol* with a debugging infrastructure layer which exposes the necessary information from the runtime, e.g., threads, stack frames, etc. However, the current debugging solutions do not tackle all the recurring problems identified when debugging user code executing within a framework.

In particular, we identify three problems: developers are exposed to (i) irrelevant, (ii) missing or (iii) dispersed information about a bug. In the first case, stack traces are overloaded, presenting more information than needed. In the second and third cases, useful debugging information about framework code is not visible, or is missing from the current call-stack. When debugging information is dispersed over many similar call-stacks, due to an error in a parallel or

long resilient execution, developers also need to manually relate the information present in the different call-stacks.

3 Sarto: A Call-Stack Instrumentation Layer for Framework-Aware Debugging

In this work we revise how the debugger and the runtime collaborate 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 propose Sarto: 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. 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 stack operations, and then describe how the different operations are used to tailor the call-stacks of four different frameworks: HTTP server, unit testing, execution of promises, and parallel execution in a big data context. In particular, we describe how they are used to *hide useless information*, *display useful information*, and *relate dispersed debugging information*.

Terminology. Before delving into the specifics of each operation, we provide our definition of a call-stack and of stack frames. 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 the stack frame that generated it (i.e., its sender or caller), its arguments, the receiver (i.e., `self` or `this`), and the local variables. Figure 2 shows a stack frame, with all the references it can hold. A stack frame also references a value stack for intermediate results, not relevant to the scope of this paper, hence not depicted in the figure.

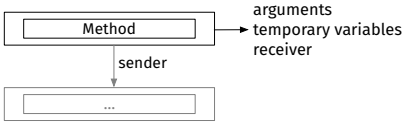

Figure 2. Description of a stack frame.

3.1 The Stack Operations

Sarto presents six stack operations to manipulate (one or more) call-stacks, framework-tailoring the debugging information presented by the call-stack. Table 1 briefly summarizes the proposed operations.

Table 1. Overview of stack operations

Operation	Description
Stack cutting	Produces a new call-stack by filtering out framework code.
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 exception 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 an original stack and a 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. We call *framework entry point* a framework stack-frame that was invoked by application code. Inversely, we call *framework exit point* a framework stack-frame that invokes application code.

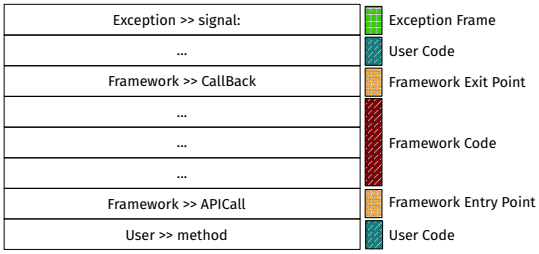

Figure 3. Representation of a call stack, marking each frame as user code, framework entry/exit point or framework code.

Figure 3 illustrates a call-stack where user code invokes framework code, and in turn framework code invokes user code in a callback. This call-stack presents a sequence of stack frames internal to the framework. 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.

Framework entry and exit points are explicitly marked by framework developers in framework code with method annotations (cf. Section 4). Cutting all framework code between exit and entry points hides all framework code, letting developers concentrate on their application code. Still, entry

and exit points are kept in the call-stack to make it explicit that a framework is involved.

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 2.2, 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 debug call-stacks that are augmented with relevant information.

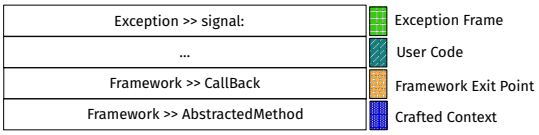

Figure 4. Representation of a call stack, with a crafted context inserted before the framework exit point.

Figure 4 displays the stack of Figure 3, where the exit point was substituted with a custom context. To avoid incompatibility among stack frames, the stack frames under the exit point are automatically excluded from the call-stack.

Concatenating stacks. The *stack concatenation* operation produces a call-stack by concatenating two different call-stacks, thus reconciling two dispersed executions and giving 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 2.3), or more in general, the call-stack of a user code callback within the remote execution of a framework.

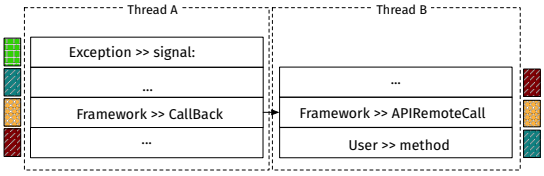

Figure 5. Representation of two call stacks during a failing remote execution of a framework call.

Consider Figure 5, displaying two call stacks related to two threads: thread A, in which the framework exit point calls back user code, that then generates an exception; thread B, that presents the call stack including the original user call to the framework entry point. The *stack concatenating* operation, links the two call stacks using the entry and exit point, present and marked in both the call stacks. In this way, while debugging the developers will see an unique call stack, that includes both the exception raised by the framework

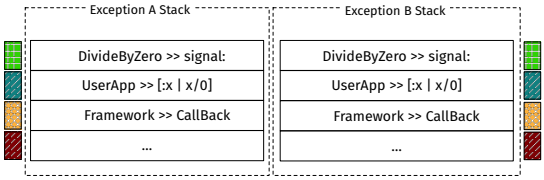

Figure 6. Representation of two call stacks upon two similar exceptions.

exit point, and the user code leading to the framework entry point, implicitly hiding the framework frames that e.g., take care of the network communication.

Stack comparison. The *stack comparison* operation analyzes two call-stacks and determines whether two call-stacks are *similar*. 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. This information can then be displayed to application developers, or be used by the debugger to e.g., optimize network transfers.

Two call-stacks are considered *similar* when they are structurally the same, as in the case of Figure 6. More precisely: two call-stacks are similar when, traversing their frames in pair, they present the same sequence of method calls, and in each pair of stack frames the program counter is the same. Variable values such as receiver or arguments nor their types are taken into account.

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

For example, consider Figure 7 showing two similar call-stacks, where each frame has different values. In the example, the dividend and the application are equivalent, but the two divisors differ. Figure 8 shows the resulting delta stack calculation for the frame analyzed above. Instead of the full frame, only the list of values that make up the delta is kept. The first variable (divisor), that was 1 in one stack and 2 in the other, is included in the delta. The other two variables (dividend and self) were not different, 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.

Applying a delta stack. The *apply delta stack* operation produces a new call-stack from a full call-stack and a delta

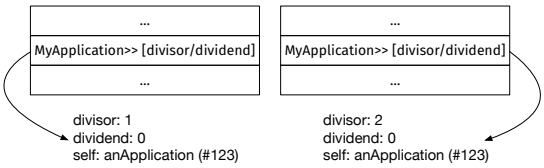

Figure 7. Two stack frames and their variables, in the calculation of the delta stack.

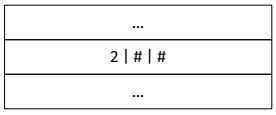

Figure 8. The calculated delta stack frame.

stack. The delta stack must have been created from a similar full call-stack or a copy of it.

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. Applying the delta stack injects each of the values present in the delta into the original stack. A debugger is set-up quickly and, in case of remote debugging, without transferring the entire call-stack many times over the network.

3.2 Debugging Framework Code with Sarto

In the remainder of this section we describe how Sarto's stack operations apply to the four described cases reducing useless debugging information, augmenting with useful debugging information, and relating similar yet dispersed debugging information.

3.2.1 Hiding Useless Debugging Information. To hide useless debugging information, we employ the *stack cutting* operation, and we display its effect when debugging an HTTP framework. Figure 9 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. Application code is isolated from framework code in the debugged execution.

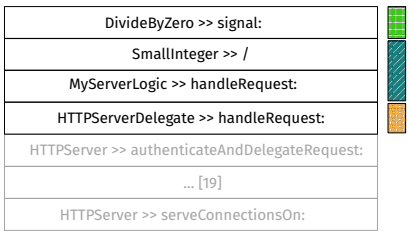

Figure 9. Representation of the stack upon an exception in the HTTP Server.

Another practical example is displayed in Figure 10: a unit test framework scenario. The shown call-stack includes both framework entry and exit points: the unit testing framework is invoked by application code, and in turn the framework code invokes application code. While the call-stack, in the top frames, has a similar structure to the one of Figure 9, at the bottom of this stack, after the framework code frames, there are more application frames before the call to the framework entry point. Using Sarto the call-stack presented to the developer include only application frames, cutting off the framework frames between the entry and exit point.

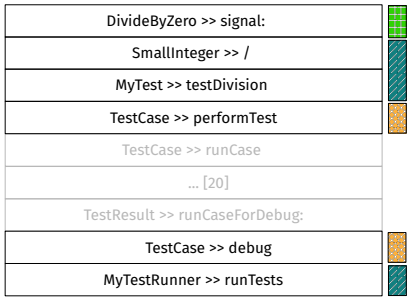

Figure 10. Representation of the stack upon an exception in an unit test execution.

3.2.2 Displaying Useful Debugging Information. To illustrate the usage of *crafting a stack frame*, we extend the previous 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 will thus show 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 under it. The developer sees the setup next to her test, and is able to re-execute the setup and the single test without having to go through any framework code.

3.2.3 Relating Dispersed Debugging Information. Consider again the case of debugging a promise execution as explained in Section 2.3. We use the *stack concatenation* operation to take both call-stacks and link them at the framework entry and exit points, the promise invocation, and the callback to the promise resolution respectively. The result is a single *asynchronous* stack, exposing both the local and remote execution contexts, depicted in Figure 11.

Our approach takes inspiration from Leske et al. [12], 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 execution of the promise through proxies. However, our approach potentially applies not only to promises but also to other asynchronous execution models such as the actor model.

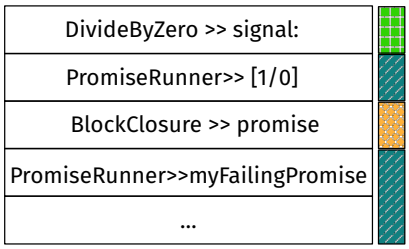

Figure 11. Representation of the instrumented stack of a failing promise, tailored for debugging

Relating similar debugging information. To illustrate the usage of the delta stack operations (*stack comparison*, *delta stack calculation*, and *delta stack application*), we extended Port, our Map/Reduce Big Data framework. Big Data frameworks present a parallel execution, which may generate two or more call-stacks with a similar structure but different values, as shown in Section 2.4. Chances are these similar call-stacks are different instances of a same problem or bug, but it is usually the developer's responsibility to tell if two call-stacks are similar or not, and to debug them singularly.

We extended Port to detect similar call-stacks with stack comparison and create delta-stacks in those cases on the server side. On the client side where the debugger is deployed, we group all similar call-stacks into one, and we show the deltas next to it. A representation of such a debugger UI is shown in Figure 12. When the developer selects a stack frame, the differences in variables are displayed in the debugger, and the developer can select to debug one of them. Starting a debugging session in a delta-stack applies it to the original stack and presents the user with a call-stack that is a copy of the original call-stack generating the problem. This avoids the replay of the executions and minimizes the amount of data that has to be stored (or transferred) for each of the exceptions.

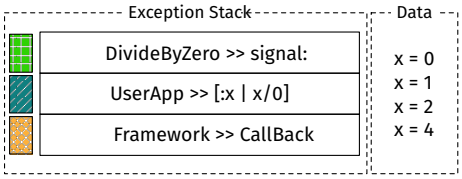

Figure 12. Representation of a debugger showing an instrumented call-stack, with the different possible variables found in the delta stacks.

4 Sarto in Practice

In the previous section we defined the 6 call-stack instrumentation operations provided by Sarto. By using these operations, framework developers provide a simplified debugging experience to application developers. In this section, we elaborate on how these operations are used to enable framework-aware debugging tools.

We implemented our approach as a framework written for the Pharo[3] programming language, a modern implementation of the classic Smalltalk programming language. Smalltalk offers a live programming environment, in which the boundaries between language implementation, runtime, applications and tools are pretty thin. When an exception happens or a breakpoint is hit, a debugger is opened. Just before opening the debugger, the runtime reifies the call-stack into objects in the heap. Sarto inserts itself between the debugger and the runtime call-stack reification step: reifications work on the reified call-stack, producing an instrumented call-stack that is then fed to the debugger. In other runtimes, instrumentations could be applied in the debugging instrumentation layer.

Capturing the call-stack to instrument. To include support for Sarto, framework developers need to insert instrumentation calls in the points where failures may happen. At these instrumentation points Sarto will force a reification of the call-stack and capture it for further manipulation. This is the case of, for example, capturing the call stack of a promise calling thread to concatenate it later with the promise resolution call-stack.

Sarto supports two instrumentation points: eager stack captures at some point in the program, and lazy stack captures at interesting debugging points like breakpoints or errors. Eager stack capturing is done through a Sarto primitive that interacts with the runtime. Lazy stack capturing is done, by means of an exception handler. For instance, a web server developer places an exception handler in the framework exit point before calling application code. In general, the custom exception handler should be placed anywhere where it would handle an exception raised in the user code. In Pharo such exception handler is introduced with the `on:do:` method, equivalent to using a `try/catch` block.

```
[...] on: Error
    do: [:err| StackMachinery manageAndDebug: err].
```

Defining framework entry and exit points. The framework developer defines framework entry and exit points by adding method annotations[4]. 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.

[3] http://pharo.org

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

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

Defining custom stack frames. The framework developer adds a custom stack frame by specifying a mapping between the frame that should be replaced and the method that will replace it in the call-stack. The replacement method can be 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 `performTest` 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: (SubstituteMethod new methodAt: #setUp})
```

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

An optional flag gives the developer enables developers to keep or hide the given framework entry/exit point 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:  (SubstituteMethod new compiledMethodAt:
      #contextForDebuggingSetup ; pcAt: #performTest ;
      keepMethod: false)  }).
```

Using stack concatenation. 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. In the case of concurrent promises, as shown in Listing 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 :=  StackMachinery captureStack.
4   promise := Promise from: self.
5   promise onFailureDo: [:err |
6     machinery := StackMachinery new.
7     combinedStack :=  machinery combineStackOfRemote:
        err withLocal: stack.
8     machinery debugError: err withStack: combinedStack)].
```

Listing 2. Capturing and debugging the stacks of a promise.

Restoring the stack. Our call-stack instrumentation layer stores the original stack before instrumentation in case a developer wants to have access to the original framework frames. When the developer debugs an instrumented call-stack and requests the full call-stack, the original stack of the exception is shown and the instrumented call-stack is discarded.

Debugging with delta stacks. To use delta stacks, the framework developer lazily captures the call-stacks she is interested in and stores them. If a captured call-stack is similar to an already stored call-stack, she can 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. We also have an analogue, but slightly more complex, implementation for a parallel execution framework. When an exception happens, an HTTP server resiliently returns an error as a response, and continues executing. In our approach, we store a copy of such call-stack or a delta-stack if applicable to debug it later.

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

Listing 3. Handling errors using delta stacks in an HTTP server.

Listing 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 `StackMachinery>>#applyDelta: delta toException: exception` to do the inverse.

5 Implementation Details

In this section, we describe some of the implementation details of our approach. Since the practical usage was already described in Section 4, we now focus on how the stack operations are implemented, as well as how some of the operations integrate with other debugging models.

In Pharo, the runtime reifies the call-stack on demand [15], mostly upon an exception, or when the developer explicitly requests to debug an execution. As defined in Section 3, the stack is reified as a linked list of `Context` objects, that are linked through the *sender* variable. A context knows its method and its its current program counter. It also holds a reference to the value of the message receiver, all temporary variables, and arguments. In our implementation, we rely on the Smalltalk stack API, that offers primitives to cut the stack at a certain context, as well as methods to create stack frames and do other kinds of modifications.

5.1 Crafting Stack Frames

Besides the automatic stack frame reification from the runtime, Pharo supports creating stack frame reifications programmatically. A `Context` object is created from a method, a valid program counter (PC) within that method, and, if any, the values of the temporary variables. In our current implementation, we only support crafted stack frames with methods that either do not use temporary variables, or in which the PC is set before temporary variables are initialized. Setting values to temporary variables will be added as future work.

5.2 Copying Call-Stacks

Our call-stack instrumentation layer makes copies of call-stacks to store or serialize them. We implemented the copy of the call-stack as a deep copy that avoids copying some special or sharable objects, such as closures and method objects.

5.3 Serializing Call-Stacks

When remote debugging our parallel execution framework or debugging a failing promise, two call-stacks need to be serialized and transferred between processes. Since stack reifications appear to the language as normal heap allocated objects, they can be serialized and transferred like any other object. To serialize, we use Fuel [6], a standard serialization library in Pharo. For network communication between different threads, we use Zinc[5], the same HTTP framework we presented as use case.

5.4 Applicability in other Languages and Runtimes

Implementing a call-stack instrumentation layer as Sarto in other runtimes requires the instrumentation layer to have access to a reification of the call-stack, or to other instrumentation to manipulate the call-stack. Most languages in the Smalltalk family provide call-stack reifications, which makes easy porting Sarto to them. Moreover, recent work shows that a Smalltalk-like call-stack reification is also efficiently implementable in the Graal VM [16].

On the other hand, Sarto could be also implemented within the debugging instrumentation layers in the IDE such as debugging API of Eclipse or Visual Studio Code: debugger APIs already include some internal representation of call-stacks. Instrumentation then may happen on the consumer of such a representation.

6 Validation

We validate our solution by applying it to four frameworks with different sets of features and requirements. We first present an overview of our experiences using Sarto and subsequently we discuss the performance benchmarks conducted to show that our solution is practical and efficient. In particular, we show that it does not introduce significant overhead to the execution or debugging, and the usage of delta-stacks improves the storage/network usage when debugging Big Data parallel applications.

6.1 Experiences in Using Sarto

To validate that our approach works for a variety of different frameworks, we implemented it to debug the following Smalltalk frameworks: Zinc, an HTTP server framework; SUnit, the classic unit testing framework of Smalltalk; TaskIt[6], a framework for task scheduling (that we adapted for remote execution); and Port [13], a Map/Reduce framework for Big Data applications. Depending on the debugging needs of each framework, a different combination of operations from Sarto was used. Table 2 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 (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. In this work, we applied *ΔStack operations* to the HTTP framework and the Big Data framework.

6.1.1 Debugging Experience for Port applications. To show the benefits of using Sarto for debugging, we describe how it was used for Port applications. Port is a Map/Reduce framework for Pharo Smalltalk that allows developers to program and execute Map/Reduce applications in parallel on different single-threaded machines, deployed locally or

[5] http://zn.stfx.eu/zn/index.html

[6] https://github.com/sbragagnolo/taskit

Table 2. Usage of basic stack operations on the four frameworks.

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

on a cluster. We employ Port because it is (i) the framework that makes use of most of Sarto's operations and (ii) it is the framework that the authors have a prolonged experience in debugging. We now describe how the different operations contribute to debugging experience in Port.

Stack cutting. Debugging Port applications happens remotely on a copy of the failed execution which can be controlled with Pharo's online debugger. When execution pauses (due to a failure or breakpoint), the call-stack presents many stack frames related to the framework called before the actual user code, as in the example described in Section 2.1. This problem is even more evident in Port applications since they are expressed as two simple functions (i.e., a map and a reduced) paralellized by the framework. This means that most stack-frames in the call-stack of a failing map or reduce do not correspond to the application developer's code. Thanks to Sarto's stack cutting, not only the developer got a debugging session focused in the application code, but also the size of the exception's stack decreased strongly. This also reduced the time to show a debugging session to the developer as less information was transferred over the network.

Crafting a stack frame. Besides stack cutting, crafting a stack frame helped us to simulate sequentiality between map and reduce functions for a particular execution (i.e., the current data partition, etc.) enabling step-by-step debugging of failed computation. To explain the importance of crafting while debugging Port applications, consider a simple word count Map/Reduce application. The map function associates a line of text from a given file to key/value pairs with the number of times each word appeared in that line. The reduce function adds up the values with the same keys to obtain how many times each word happened in the whole file. If we debug the map function, the fact that the call was parallelized by the framework, the actual call of the developer (map and then reduce) was never explicitly present in the stack. Crafting a stack frame allows us to improve the debugging session since it adds a frame simulating the call to map, followed by a call to reduce. If the developer steps over the map, the reduce is executed locally, and on the portion of the data available in the copy of the stack. This enables simulating how the execution would continue if the code was correctly fixed, allowing developers to analyze the state of the execution (and all its variables) in a step-by-step manner.

Stack comparison and Delta Stack operations. Stack comparison and delta stack operations are crucial in Port to avoid overwhelming the developer with numerous debugging sessions. When using Port, the same failure may happen at different times during a single parallel execution depending on how a dataset is partitioned, or on the nondeterministic behaviour of the parallel execution. When an exception happens, a debugging session is created and the debugger UI is presented to the developer's machine. Originally, developers would be presented with different debuggers, containing different (but in fact similar) call-stacks, leaving is to the developer to determine how and whether they were related. We employed stack comparison and delta stack calculation to group similar call-stacks and present a single debugger whenever possible. Exception raised by the same method, but on different data, are hence grouped, and the developer can decide which particular execution to debug, without needing to replay. Delta stacks also enable framework-specific debugging, such as debugging on virtual partition, described by Marra et al. [13]. This operation helped us to further reduce at least by half the amount of data that has to be sent over the network when serializing an exception for debugging (cf. Section 6.3).

6.1.2 Framework Developer's Experience. Since Port's applications are written around two user-defined functions, it is easy to clearly identify the entry and exit points to be able to reason about the different Sarto's operations, improving the debugging experience. This was not the case for the other frameworks. To identify user-defined stack frames in the other frameworks, we analyzed a series of exceptions found while implementing Port's framework code (as Port employs Zinc, SUnit, and Taskit). By analysing those exceptions, we extrapolated the framework entry and exit points (as identified in Section 3.2) as those methods that actually performed a call back to user code. To our experience, finding the framework entry/exit point was not particularly time consuming, since it happened gradually while debugging the aforementioned frameworks.

Deciding which stack frame to craft, however, was a more complex task than identifying the framework entry/exit points, because it requires a higher level of knowledge on the framework, and the elements to abstract for debugging.

The interface proposed in Section 4 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. This interface, however, may not be specific enough in some cases (e.g., setting the PC at a message send may not be straightforward when the same message may be sent multiple times in the same crafted method). Thus, our experiences with Sarto's crafting method interface indicate that further application to other frameworks is needed to

identify recurrent patterns to abstract, and to improve the interface.

6.2 Performance of the Operations

In this section, we evaluate the performance overhead of exception handling when using our approach. We do so, by answering three questions:

1. Do the stack operations impact the time of exception handling for debugging?
2. Are the stack operations affected when increasing the size of the stack?
3. Are there advantages when storing a delta stack as opposed to storing all exceptions?

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

6.2.1 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 the special handler), 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.

For the purposes of this validation, we extended the debugger to store a timestamp just before it is opened. We perform the measurement 25 times for each of the frameworks.

Table 3. Time to open a debugger with or without managing the exceptions. Times in milliseconds.

Framework	*Managed*	Err_M	*Default*	Err_D
HTTP Framework	71.5	3.1	103.5	7.74
Unit Testing	28.32	1.5	75.08	4.2

Table 3 shows the results of our benchmark: the managed column represents the average execution when managing the exception with Sarto, the unmanaged 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 maximum 103.5 milliseconds, which we consider small enough for an interactive debugger. Handling an exception with our instrumentation layer resulted 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 less times. From those results, we conclude that our instrumentation layer does not introduce a significant impact in the execution.

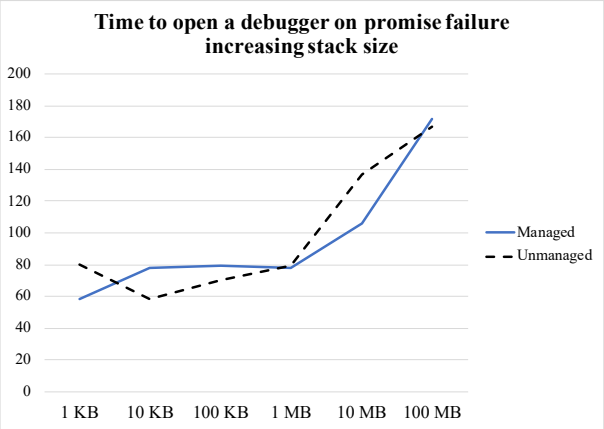

Figure 13. Runtime of a failing promise, when increasing the size of the stack.

6.2.2 Overhead when Increasing Stack Size. This benchmark measures the influence of the stack size on our approach. This benchmark is a variation of the previous one, and we decided to apply it on a third use case: promises using TaskIt. This use case, in fact, requires a copy of the call-stack, hence it gives a measure of the overall overhead of our approach. 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 creates it.

Figure 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: we have to consider that the X axis, i.e., the amount of data, is growing exponentially (by a factor of 10).

6.3 Memory Usage of Delta Stacks

This benchmark determines the advantages of storing a delta stack as opposed to storing full call-stacks. We performed this benchmark on the parallel execution framework Port, integrating our approach into the existent debugging support. We measure the size of the serialized debugging messages that a debugger receives from a remote cluster. A debugging message includes the exception and its full call-stack cut at the framework exit point. In case of a subsequent exception, a delta stack is sent instead of the full call-stack.

We compare the behavior of two different applications: the first one (Simple Error) produces a division by zero during a map; the second one (Kmeans) is a parallel implementation of the k-means clustering algorithm to calculate statistics

on previously recorded tweets. We inject an exception while extracting data from the tweet stream.

The first application presents a short call-stack upon exception, that does not reference heavy data structures, and shows the basic overhead of our approach. The second application, presents a more realistic call-stack that is deeper and references heavy data structures.

We run our benchmarks with an increasing number of parallel workers, so the number of exceptions that happen in parallel increase consequently, and measure the impact on the size of a single delta.

For the k-means application we run not only with different numbers of workers, but also on different amounts of data, increasing the size of the stack. We analyze 250 MB of data with 2, 4, and 8 workers, and 1 GB of data with 4 workers. Please note that the benchmarks were executed on a low-spec processor, so we could not further increase the amount of workers or data analyzed.

Table 4. Size of debugging event using Delta Stacks. Sizes are in KB.

Application	$N_{workers}$	$Exception_{size}$	$Delta_{size}$
Simple Error	2	3.505	1.911
Simple Error	4	4.437	1.860
Simple Error	6	3.254	1.845
Simple Error	8	3.744	1.837
Kmeans 250 MB	2	494.667	263.747
Kmeans 250 MB	4	445.053	214.131
Kmeans 250 MB	8	405.888	232.272
Kmeans 1 GB	4	442.416	211.458

Table 4 shows the results of the benchmark on the different scenarios. The first column depicts the scenario, the second the number of workers, the third the size in KB of the debugging event including a copy of the full call-stack, and the last column the size in KB of the debugging event including only the delta stack. The first scenario error does not seem dependent on the number of workers. Also, the size of the full debugging event is between 3.2 and 4.3 KB, while the size of the delta debugging event is around 1.8 KB. Although the numbers are relatively low for both measurements, our results show two trends: the size of the delta event is from 1.7 to 2.4 times lower than the full debugging event, and the delta event introduces a basic memory usage of less than 2 KB with small stacks.

The first trend is visible also when running the measurements with the more complex k-means application. As for the simple error application, the results do not look dependent on the number of workers. We observe the same ratio between delta event size and full exception: the first is between 1.7 and 2 times lower than the second. The second trend, instead, is less visible because the basic overhead of the delta debugging event is hidden by the bigger size of the call-stacks.

Finally, when increasing the amount of data we can notice that the size of the debugging event does not grow accordingly. This is because of further optimizations to the serialization of the debugging event already present in the framework: big data structures are resized to fit a certain limit, so debugging events (and their deltas) cannot grow unbound.

7 Related Work

In this section we describe some of the closest related work in the field of debugging domain-specific code and domain-specific debuggers for asyncronous execution and Big Data.

Debugging domain-specific code. There exist several work about debugging techniques to help developers debug domain specific code. Classical debuggers, such as GDB or Eclipse's Debugger, offer some primitives to filter stack frames. When debugging Python code in GDB, developers can define different *frame filters* to hide some stack frames from the view. Similarly, Eclipse's debugging support offers a filtering operation, that based on heuristics (e.g., the file or package where the executed method was defined), filters out some stack frames. While these solutions are viable to hide framework stack frames, they are bound by one heuristic that does not let framework developers tailor the stack that will be debugged. Furthermore, these approaches offer no support for relating debugging information.

The moldable debugger [4] provides different visualizations of both the call-stack and the debugged method. Developers extend it to debug different domains such as bytecodes, parsers or notification systems. The moldable debugger has its focus set on the tool: debugger extensions and custom debuggers are mainly implemented by specializing the UI. Our approach shares with this work the domain-tailoring objective, where we focus particularly on framework-aware support in the underlying mechanisms of a debugger. As such, our approach presents 6 operations that, combined differently, can tailor call-stacks not only to display them in a debugger, but also to improve the performance of debugging remote applications.

The Chrome DevTools Protocol [8], is a protocol to communicate information, including debugging information, from a runtime to the IDE. It defines a set of primitives that allow tools to debug different languages, provided that the information passed to a supported tool follows the correct specification. Similarly, Keidel et al. [10] propose an IDE-independent intermediate representation, that can be used by different IDEs, reducing the amount of IDE plugins necessary to debug a single language. While these solutions focus on debugging different languages, or supporting different tools, they do not provide domain-specific solutions to debug user code within a framework execution.

Debuggers for asynchronous execution. Debugging asynchronous executions is difficult, mainly because the execution of a program spans over many different threads. In the case of promises, the thread that creates the promise has a relationship with the thread that is executing the promise, but the second is normally not shown in a debugger, that will allow to debug only the promise execution. Different domain-specific solutions for better debugging of asynchronous execution have been proposed. Dragos et al. [7] focus on capturing stack frames at interesting points (such as future creation), to show them to the developer when debugging. Similarly, Chrome DevTools [8] supports debugging of asynchronous stack traces by storing such information in the stack trace. Other approaches [1, 19] 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. The closest work is Leske et al. [12] which 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 can then debug a single stack that combines both the frames leading to the promise creation, and the ones of the promise execution. While very similar to our approach, they employ proxies to stack frames in order to link the two threads. This means that they do not copy stack frames, but add a proxy to each of the frames. This makes their approach susceptible to debugging interference due to the use of network communication for every operation that needs to be done [14]. Furthermore, their approach focuses specifically on promises, and is not directly adaptable to other frameworks: they present an approach to concatenate stack of promises, while our approach provides five other operations that are combinable and applicable to other scenarios.

Domain specific debuggers for Big Data. Several related work exists in the literature about debugging support for Big Data frameworks. Most solutions can be classified as either replay solutions, such as Arthur [5], or online solutions combined with data provenance, such as BigDebug [9]. The first family of solutions suffer of high replay times. The second offer online debugging capabilities that partially reduce the amount of replay by, e.g., inserting simulated breakpoints. More recently, Marra et al. [13] define a debugging model to deal with such parallel failures remotely, that involves serializing the call-stack over the network for debugging. In their approach, they use different stack operations to reduce the amount of data serialized over the network, including stack cutting and crafting stack frames. Interestingly, they use the concept of composite exceptions to relate exceptions that happen in parallel, specifically in the case of the same exception caused in parallel by multiple data. They devise an approach where, when two *similar exceptions* happen, so two or more exceptions that have structurally exactly the same call-stack, the debugger can calculate some meta-data to identify one of the two exceptions, so only one full stack is sent over the network. Similarly, our approach presents operations applicable to the domain of Big Data, by tailoring call-stacks using Big Data frameworks to minimize the transfer of data and remove the noise of extra stack frames. However, our solution applies not only to Big Data but to a broader spectrum of frameworks.

8 Conclusion

Debugging user code within framework code is not always easy: the call-stack presents irrelevant information, with many stack frames that the developer is not interested in. On the other hand, the call-stack misses important information, with some relevant methods not present at all in the call stack, because they already returned or they executed in another thread. Furthermore, when multiple exceptions happen in parallel or over a sequence of time, limited support is provided to relate the different exceptions in order to ease debugging.

In this paper, we explored introducing different stack operations in the 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 the 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 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 four different use cases: an HTTP web server, a Unit Testing framework, Promise Executions, and Parallel Executions. They all use a different subset of the 6 operations. To show the validity of our solution we presented our experience in both instrumenting and debugging one of the frameworks (Port) using Sarto. Then, we conducted performance benchmarks to show that our approach does not add noticeable overhead in exception handling, and that delta stacks actually improve performance when debugging multiple exceptions, reducing the memory footprint by half when compared to storing the full stack of all the exceptions.

Acknowledgments

We would like to thank the anonymous reviewers for their useful feedback. We would also like to thank Stefan Marr for his early input and Kevin De Porre for his help reviewing the paper. Matteo Marra is a PhD-SB fellow at the Fonds Wetenschappelijk Onderzoek - Vlaanderen - Project number:

1S63418N. We are also grateful to the European Smalltalk User Group (http://www.esug.org) for their financial support.

References

[1] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding Broken Promises in Asynchronous JavaScript Programs. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 162 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276532

[2] Elisa Gonzalez Boix, Carlos Noguera, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt. 2011. REME-D: A Reflective Epidemic Message-Oriented Debugger for Ambient-Oriented Applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11)*. Association for Computing Machinery, New York, NY, USA, 1275–1281. https://doi.org/10.1145/1982185.1982463

[3] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. 1998. Concurrency and Distribution in Object-Oriented Programming. 30, 3 (Sept. 1998), 291–329. https://doi.org/10.1145/292469.292470

[4] Andrei Chis, Marcus Denker, Tudor Gorba, and Oscar Nierstrasz. 2015. Practical domain-specific debuggers using the Moldable Debugger framework. *Computer Languages, Systems & Structures* 44 (2015), 89 – 113. https://doi.org/10.1016/j.cl.2015.08.005 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[5] Ankur Dave, Matei Zaharia, Scott Shenker, and Ion Stoica. 2013. Arthur: Rich Post-Facto Debugging for Production Analytics Applications. *Technical report, University of California* (2013).

[6] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. 2011. Clustered Serialization with Fuel. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST '11)*. ACM, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/2166929.2166930

[7] Iulian Dragos. 2013. Stack Retention in Debuggers For Concurrent Programs. http://iulidragos.com/assets/papers/stack-retention.pdf

[8] Google. [n. d.]. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/

[9] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 784–795. https://doi.org/10.1145/2884781.2884813

[10] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. 2016. The IDE Portability Problem and Its Solution in Monto. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/2997364.2997368

[11] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. 2018. CauDEr: A Causal-Consistent Reversible Debugger for Erlang. In *Functional and Logic Programming (FLOPS'18)*, Vol. 10818. Springer, 247–263. https://doi.org/10.1007/978-3-319-90686-7_16

[12] Max Leske, Andrei Chiş, and Oscar Nierstrasz. 2016. A Promising Approach for Debugging Remote Promises. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies (IWST'16)*. ACM, New York, NY, USA, Article 18, 9 pages. https://doi.org/10.1145/2991041.2991059

[13] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2020. A debugging approach for live Big Data applications. *Science of Computer Programming* 194 (2020), 102460. https://doi.org/10.1016/j.scico.2020.102460

[14] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2018. Out-Of-Place Debugging: a debugging architecture to reduce debugging interference. *The Art, Science and Engineering of Programming* 3, 2 (October 2018), pp. 3:1–3:29. https://doi.org/10.22152/programming-journal.org/2019/3/3

[15] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two Decades of Smalltalk VM Development: Live VM Development through Simulation Tools. 57–66. https://doi.org/10.1145/3281287.3281295

[16] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/3357390.3361024

[17] Kazuhiro Shibanai and Takuo Watanabe. 2017. 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)*. ACM, New York, NY, USA, 50–57. https://doi.org/10.1145/3141834.3141840

[18] Terry Stanley, Tyler Close, and Mark Miller. 2009. *Causeway: A message-oriented distributed debugger*. Technical Report. HP Labs. 1–15 pages.

[19] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. 2019. 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)*. IEEE Press, 61–72.