Improving live debugging of concurrent threads through thread histories[1] (Preprint[2])

Max Leske[a], Andrei Chiş[b,*], Oscar Nierstrasz[a]

[a] *Software Composition Group, University of Bern, Switzerland*
[b] *Feenk GmbH, Switzerland*

Abstract

Concurrency issues are inherently harder to identify and fix than issues in sequential programs, due to aspects like indeterminate order of access to shared resources and thread synchronisation. Live debuggers are often used by developers to gain insights into the behaviour of concurrent programs by exploring the call stacks of threads. Nevertheless, contemporary live debuggers for concurrent programs are usually sequential debuggers augmented with the ability to display different threads in isolation. To these debuggers every thread call stack begins with a designated start routine and the calls that led to the creation of the thread are not visible, as they are part of a different thread. This requires developers to manually link stack traces belonging to related but distinct threads, adding another burden to the already difficult act of debugging concurrent programs. To improve debugging of concurrent programs we address the problem of incomplete call stacks in debuggers through a thread and debugger model that enables live debugging of child threads within the context of their parent threads. The proposed debugger operates on a virtual thread that merges together multiple relevant threads. To better understand the features of debuggers for concurrent programs we present an in-depth discussion of the concurrency related features in current live debuggers. We test the applicability of the proposed model by instantiating it for simple threads, local and remote promises, and a remote object-oriented database. Starting from these use cases we further discuss implementation details ensuring a practical approach.

Keywords: concurrency, debugging, promises, Smalltalk, domain-specific tools

1. Introduction

According to Pennington [29], developers build a mental model of a program in terms of control flow and data flow. Live debuggers support developers in building that mental model by providing access to concrete values of variables and real-time views of the effects of expressions and statements. For control and sequence bugs [3] in particular, such as erroneous conditional expressions or invalid state transitions, the ability to navigate the call stack helps developers to identify and fix issues. Even developers with a good mental model of the program may need the call stack when they find that their model is wrong or incomplete.

Concurrent programs, however, distribute computation by forking and joining multiple threads. A single thread is in itself a sequential program that can create new threads, which are executed concurrently. We

[1] This work is an extended version of a previous work: A promising approach for debugging remote promises. In: Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies (IWST 2016), http://doi.acm.org/10.1145/2991041.2991059

[2] Max Leske, Andrei Chiş, Oscar Nierstrasz, 'Improving live debugging of concurrent threads through thread histories'. In: Science of Computer Programming, 2017. DOI: 10.1016/j.scico.2017.10.005

[*] Corresponding author
Email addresses: maxleske@gmail.com (Max Leske), chisvasileandrei@gmail.com (Andrei Chiş)
URL: http://andreichis.com (Andrei Chiş), http://scg.unibe.ch/staff/oscar (Oscar Nierstrasz)

call the original thread the *parent* and any thread created by the parent a *child*. Parent threads may await termination of their child threads—which is called "joining"—or they may not. Child threads can themselves be parents of other threads, thus forming a hierarchy rooted in the first thread of the operating system's launch process, which has no parent. In this context, reasoning about control flow in concurrent programs requires developers to investigate different types of relations between threads, such as their histories, how they interact and exchange data, or how they synchronize their executions. We use the term *history* to describe the complete call stack of a thread including the activation records of all its parent threads.

The need for debuggers that show these types of relationships between threads has been recognised as early as 1986 [37, 7]. One possibility for giving developers access to the inter-thread relationships are traces, which have been used for sequential debugging since 1969 [2]. Traces provide serialised records of the events that have occurred during the execution of a program. Trace debuggers simplify search and filtering of traces. Visual trace debuggers use the information from traces to present visualisations to the user that attempt to highlight specific properties of a trace. Specialised (visual) trace debuggers can also display dependencies between processes and threads (Utter and Pancake [37] give an excellent, albeit outdated, overview of parallel visual debuggers). Trace debugging offers a powerful way to analyse issues in a program, especially related to concurrency. However, traces are usually costly to process, due to the large amount of data they contain, and are therefore better suited for postmortem debugging.

Live debuggers offer a faster turnaround and more direct interaction with the program than trace debuggers. Some live debuggers even support manipulation of the program on the fly (also known as "fix-and-continue debugging") so that the program does not need to be restarted or recompiled. Examples include debuggers for Lisp [18], and Smalltalk [15], where this functionality is built into the language environment, as well as debuggers for Java[3] and C++. For all their strengths, live debuggers have not yet reached their full potential when it comes to debugging threads. Developers would certainly profit from being presented with more information about threads and their relationships in live debuggers.

Towards this goal, in this paper we explore how to integrate into live debuggers one type of relationship between threads, namely the complete history of a thread. While approaches exist to improve debugging of concurrent programs, most conventional debuggers still present only a view on the failed child thread, which is independent of the parent thread. This fails to capture the relationship between the parent and the child. For threads running in the same address space it is often possible to reconstruct these relationships from contextual information. This approach is used by the developer tools of the Chrome browser to present asynchronous events in a sequential view [8]. In a remote execution setup however, threads are additionally separated by a communication channel which makes it harder to discover the inter-thread relationships, and raises challenges related to remote communication. Concurrent models like promises or actors that build on top of threads, while simplifying the creation of concurrent programs, introduce further challenges for recovering a thread's history. This happens as live debuggers need to take into account how threads are used internally by those concurrency models.

To improve debugging of concurrent programs using both local and remote threads, as well as other concurrent models, we propose to address the problem of incomplete thread history through a unified thread and debugger model that enables live debugging of child threads within the context of their parent threads. To achieve this, when an exception is raised in the child thread, instead of showing the call stack of either the parent or the child, the debugger presents to the developer the call stack of *a single virtual thread*. This virtual thread merges the child thread with its parents, to give a serial view of events. If the parent thread is waiting for the completion of multiple child threads, the virtual stack only shows the history of the child thread that contained the error. We retain the history of a child thread by creating a copy of the parent's call stack at the point where the child is being created. While this idea is conceptually straightforward, it raises many design challenges. Memory footprint, performance of copying activation records, and ensuring live debugging actions (*e.g.*, step into, step over) in the presence of a virtual thread, are some of the most important.

To investigate the practical applicability of a debugger model that works with a virtual thread, we apply the proposed model to four different contexts:

[3] https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

- *Local threads:* We start with the basic case where a parent thread creates a child thread in the same address space. We present a prototype implementation of a debugger in the Pharo programming environment [5] that extends the thread model provided by Pharo;
- *Local promises:* We extend the base approach by replacing the child thread with a promise. We rely on the TaskIt[4] library for working with promises and build a debugger that takes into account TaskIt promises. As promises represent another type of concurrency model for synchronizing communication in a concurrent application, this shows that the presented approach is not just tied to basic threads;
- *Remote promises:* In the previous two cases both threads exist in the same address space. We explore challenges that arise when the child thread is located in a remote environment by building an implementation of remote promises using Seamless, a framework for distributed computing in Pharo, and designing a debugger that takes into account this aspect;
- *Remote object-oriented database:* The previous three contexts involve threads belonging to one programming language: Pharo. To explore the context in which the virtual thread involves different programming languages we applied our model to GemStone/S, a remote object-oriented database that uses a syntax compatible with Pharo, but has a different thread model that does not have a one-to-one mapping with the one from Pharo. In this case GemStone/S provides an API to access and interact with the remote stack.

We discussed the approach for debugging remote promises in a previous work [20]. We also sketched the approach for debugging local threads in a short two-page paper [19]. The present paper extends our previous work as follows: *(i)* we present a more detailed thread and debugger model for working with a virtual thread, *(ii)* we present in detail the use cases for debugging local threads, local promises, remote promises, and a remote object-oriented database, and *(iii)* we provide an in-depth analysis of existing debugging support for concurrent programs.

The overall contributions of this paper are as follows:

- An in-depth analysis of existing debugging support for concurrent programs (Section 2);
- A unified thread and debugger model for improving debugging of concurrent programs through live debugging of child threads within the context of their parent threads (Section 3);
- The application of the proposed model in the contexts of local threads, local and remote promises, and an object-oriented database (Section 4);
- A prototype implementation for each of the four aforementioned use cases (Section 4);
- A report on the technical challenges for making the approach practically feasible together with current limitations (Section 5).

2. Current state of live debuggers

In this section we clarify the terminology used throughout the paper and examine the current state of live debuggers for different languages. Our analysis focuses on the shortcomings of current live debuggers in enabling debugging of concurrent programs.

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

2.1. Terminology

Threads. Throughout this paper we use the term *threads* to mean *green thread* [34], unless stated otherwise. Green threads are virtual threads that share their memory and are located and scheduled in user space by a virtual machine.

A *remote thread* is a green thread running in a different virtual machine, possibly executing on a different host. A thread consists of a linked list of activation records, called a *call stack*. Newly activated methods are put on the top of the stack, so that the bottom activation record represents the starting point of the thread.

Promises. The concept of *promises* was introduced by Friedman *et al.* [12] for the Lisp programming language. Since then promises have been extended and implemented in a variety of ways so that today the definition depends on the language and the specific use case. Liskov and Shrira for example, extend promises to have a static type and support for exceptions [21].

For this paper we define a promise as an eventual value, the computation of which may execute in a child thread. Our implementation of promises supports exception handling. A remote promise is a promise that is executed by a different virtual machine than its parent thread; the virtual machine may run on a different host.

2.2. Challenges

Live debuggers have to overcome a wide range of challenges to successfully present developers with the complete history of a thread. On the one hand, the POSIX standard [36] specifies that a thread is created in such a way that the first activation record executes a start routine,[5] which the caller must specify, and that the thread is terminated immediately after returning from the start routine (a thread may also terminate earlier explicitly). Building the child thread's history requires access to the activation records of the parent thread. However, as the parent's activation records have been created before the start routine of the child thread they cannot be reached from the child thread. Hence, thread implementations do not need to make these earlier activation records available to the child thread. The consequence is that a live debugger operating on such an implementation can only show the activation records of a child thread up to the point of the thread's start routine. One might argue that a developer could switch to the parent to look at the earlier activation records but that is only true when: *(i)* the parent has not yet exited, and *(ii)* the parent is waiting for the completion of that child in the same activation record in which the child was created. The second point is important because, while some of the activation records that are part of the child's history may still be present in the parent, activation records from which the parent has returned are no longer accessible. For example, if the parent has returned from the activation record that created child A and is waiting for the completion of child B, the activation record that created child A is no longer available in a live debugger.

On the other hand, developers do not always manipulate threads directly. To simplify the development of concurrent programs they can instead rely on concurrent models like promises, actors, active objects, thread pools, *etc.* Debuggers for concurrent programs then need to also take into account the particularities of those concurrent models, as well as of their implementations. This introduces further debugging challenges. For example, a promise [12, 1] hides the complexity of thread synchronisation from the developer by providing a placeholder for the value the concurrent thread will return. The placeholder performs the necessary synchronisation when access to the value of the promise is requested. Promises can execute their computation in a local thread within the same address space as the parent thread that launched the promise or in a remote thread running in another address space, on the same machine or another one. We refer to the latter as *remote promises* [21].

Due to the concurrent nature of promises, debugging them is often as complex as debugging conventional concurrent threads [24]. Furthermore, little debugger support exists for promises in general and for promises involved with remote threads in particular. A developer can proceed to debug promises using generic

[5] http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_create.html

techniques for debugging concurrent threads. One common approach is to simply open a debugger on the child thread when an exception is raised, as would be the case with a conventional thread. To explore ways to improve the debugging of promises, and show that the proposed approach can be used for concurrency models other than those based on threads, we decided to accommodate both local and remote promises.

2.3. Selected debuggers

The space of programming languages today is too large[6] to produce a survey of all the debuggers of these languages in the context of this work. Hence, we had to generate a representative sample of the space of all debuggers. Since debuggers are tied to a single language or only few different ones, we looked at different programming languages to obtain different debuggers.

We started by examining debuggers of popular programming languages, as their widespread use ensures that they are supported by a wide range of tools. The TIOBE Index[7] provides a continuously updated list of the most popular programming languages based on search engine results. The top ten of the index have been composed of the same languages since 2011 (with the exception of Visual Basic .NET), which indicates that these languages are not only popular but important to the industry. In June 2016 the top ten languages, in descending order, were: Java, C, C++, Python, C# , PHP, JavaScript, Perl, Visual Basic .NET and Ruby. To round out the picture we added Objective-C and Swift for Apple's iOS platform since mobile platforms belong to the most important and fastest evolving platforms (the languages used for Google's Android and Microsoft's Windows Phone platforms are already included in the top ten).

Innovation in programming languages and tools for those languages is also driven by research at universities. The languages used in software research are often not popular enough to appear in the top 10 of the TIOBE index. To account for languages used in research, we added Haskell (University of Glasgow [16]), Scheme (MIT [35]), Scala (EPFL[8]), Self (Stanford University[9]), OCaml (INRIA[10]), Prolog (University of Edinburgh, Université d'Aix Marseilles [17]) and Pharo (University of Bern[11], INRIA[12]) to the sample.

From the selected languages we derived a list of debuggers by including the standard debugger for each language (if one exists) and possibly others that are used by a significant number of developers (*e.g.*, debuggers used by popular development environments). The debuggers covered by our survey are: Java Debugger Interface[13] (Java), Visual Studio debugger[14] (C# , C++, Visual Basic .NET, JavaScript), PyDev[15] (Python), pdb[16] (Python), perldebug[17] (Perl), GDB[18] (C), Chrome development tools[19] (JavaScript), XDebug[20] (PHP), Zend Debugger[21] (PHP), debug.rb[22] (Ruby), LLDB[23] (Objective-C, Swift), GHCi debugger (Haskell)[24], Concurrent Haskell debugger [6] (Haskell), SISC debugger[25] (Scheme), Scala asynchronous debugger[26]

[6] Wikipedia (https://en.wikipedia.org/wiki/List_of_programming_languages) for example listed 694 programming languages on July 8th 2016
[7] http://www.tiobe.com/tiobe_index
[8] http://www.artima.com/weblogs/viewpost.jsp?thread=163733
[9] http://www.selflanguage.org
[10] http://ocaml.org/learn/history.html
[11] http://scg.unibe.ch/research
[12] http://rmod.inria.fr/web/research
[13] http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/
[14] https://msdn.microsoft.com/en-us/library/sc65sadd.aspx
[15] https://github.com/fabioz/PyDev.Debugger
[16] https://docs.python.org/2/library/pdb.html
[17] http://perldoc.perl.org/perldebug.html
[18] https://www.gnu.org/software/gdb/
[19] https://developer.chrome.com/devtools
[20] https://xdebug.org
[21] http://files.zend.com/help/Zend-Studio/content/remotely_debugging_a_php_script.htm
[22] http://ruby-doc.org/stdlib-2.0.0/libdoc/debug/rdoc/DEBUGGER__.html
[23] http://lldb.llvm.org
[24] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html#the-ghci-debugger
[25] http://www.sisc-scheme.org/manual/html/ch04.html
[26] http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html

(Scala), Self debugger[27] (Self), ocamldebug[28] (OCaml), XPCE debugger[29] (Prolog), Pharo debugger.[30]

2.4. Debugger features

By studying how the debuggers we selected handle debugging of concurrent processes and threads we can observe that they all adopt a similar approach:

1. when a breakpoint is reached, suspend all running threads (or processes);
2. present the user with lists of all current threads;
3. let the user choose the thread she is interested in;
4. present details of the selected thread to the user, isolated from other threads.

To gain more insight into this strategy we identify the features each debugger provides to support threads and promises. For every feature we composed a question to determine whether a given debugger supports it. The following list shows the questions we asked, the answers are listed in Table 1:

- *Thread hierarchy information:* Does the debugger provide some kind of information (visual or textual) about the parent-child relationship of threads?
- *Thread switching:* Does the debugger allow developers to change the focus to another thread?
- *Thread isolated breakpoints:* Can breakpoints be isolated to specific threads, *e.g.*, through conditional breakpoints?
- *Promise history:* Can the debugger show the history of promises across different threads?
- *Thread history:* Can the debugger display the original call stack of the parent thread for a given child?

Thread hierarchy information. The only debugger that provides any kind of information about the hierarchichal relationship between parent and child threads is the Visual Studio debugger (for so-called "managed code").

Thread switching. In this case it is interesting to note that Perl and Python implement threading but do not provide adequate debugger support. It is even more interesting that PyDev allows developers to switch threads even though it is not the standard debugger for Python. It would be interesting to find out why debugger support for such a fundamental feature is not being provided by the standard tools.

The debuggers for Haskell, Scheme and OCaml provide only implicit thread switching because they are functional languages in which threads are also implemented with functions. In these implementations only the active thread can be displayed in the debugger and switching to a different thread requires a context switch. Scala, which is also partly functional, by contrast uses the thread implementation of the HotSpot virtual machine, and therefore its debugger can provide explicit thread switching.

Thread isolated breakpoints. In all of the languages we looked at, threads are exposed as first class objects that allow access to their internal identification ("thread-ID"). Many of the debuggers satisfy the property of isolated thread breakpoints simply because they offer conditional breakpoints where the condition can include a comparison with the current thread's thread-ID. Not all debuggers provide conditional breakpoints out of the box however. The SISC debugger for Scheme for example does not include a facility for conditional breakpoints but can be extended with a custom function that provides it. Of the debuggers that we looked at for Haskell, Scheme and OCaml, only the GHCi debugger for Haskell supports breaking on different threads in the same session.

[27] http://handbook.selflanguage.org
[28] http://caml.inria.fr/pub/docs/manual-ocaml/debugger.html
[29] http://www.swi-prolog.org/pldoc/man?section=guitracer
[30] http://pharo.org/documentation

	thread hierarchy information	thread switching	thread isolated breakpoints	promise history	thread history
Java Debugger Interface	·	✓	✓	·	·
Visual Studio debugger	✓	✓	✓	·	·
PyDev	·	✓	✓	·	·
pdb	·	·	✓	·	·
perldebug	·	·	✓	·	·
GDB	·	✓	✓	·	·
Chrome development tools	n/a	n/a	n/a	✓	n/a
XDebug	·	✓	✓	·	·
Zend Debugger	·	✓	✓	·	·
debug.rb	·	✓	✓	·	·
LLDB	·	✓	✓	·	·
Concurrent Haskell debugger	·	·	✓	·	·
GHCi debugger	·	✓	✓	·	·
SISC debugger	·	·	✓	·	·
Scala asynchronous debugger	·	✓	✓	✓	·
ocamldebug	·	·	✓	·	·
Self debugger	·	·	✓	·	·
XPCE debugger	·	✓	✓	·	·
Pharo debugger	·	✓	✓	·	·
Pharo thread debugger	·	✓	✓	✓	✓

Table 1: Thread related features of live debuggers

Promise history. Only the Scala asynchronous debugger and the Chrome development tools can show stacks for promises across multiple threads. Both debuggers can show the user the activation record where a promise was created and the activation record where it was resolved, even if the resolution occurred in a different thread as the creation.

Thread history. None of the debuggers we looked at implements our idea of augmenting the call stack of a thread with the call stack of its parent.

2.5. Notes on debuggers

In this section we discuss particular aspects of the analyzed debuggers with respect to the aforementioned features.

perldebug. perldebug includes an experimental thread debugging option that identifies the current thread and can list all running threads. It is not possible to switch between different threads in perldebug.

Visual Studio debugger. The Microsoft Visual Studio debugger (C# , C++, Visual Basic .NET) can visualise thread hierarchies (*i.e.*, the relationship between parent and child threads within a single process) and show activation records shared among different active threads (*e.g.*, two threads created at the same point will share at least some activation records). An additional "task view" provides the same information for tasks, a concept related to promises.[31]

[31] https://msdn.microsoft.com/en-us/library/hh873175(v=vs.110).aspx

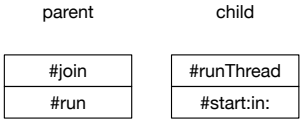

Figure 1: Stacks of parent and child threads at the moment where the parent is waiting for the child to exit (line 3) and the child is executing the method #runThread.

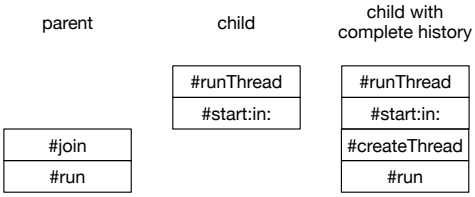

Figure 2: On the right the call stack of the child is shown as it would appear in a live debugger if the child and its parent were executed as a single sequential program. Note that the frame of #createThread is accessible in neither the child nor the parent.

Chrome development tools. JavaScript is single-threaded but the event mechanism uses threads or a mechanism amounting to the same. For that reason only "promise history" is an applicable question for JavaScript debuggers.

The "Async" option in the Chrome development tools makes the call stack list display a contiguous stack for events, promises and asynchronous network requests (XMLHttpRequest).[32] Traditionally, these constructs are displayed only with the stack of their current context *i.e.*, the place where an event, promise or XMLHttpRequest was created cannot be determined from the visible stack. The idea is very similar to the one used in the Scala asynchronous debugger.

Concurrent Haskell Debugger. This is an experimental debugger for Haskell that employs the same scheme as other debuggers but also visualises relationships between threads through lines between threads and shared resources.

Scala asynchronous debugger. The Scala asynchronous debugger is an extension to the Scala debugger that adds support for debugging of actors and futures. To that end the debugger creates a continuation at the point where a future is being created or a message is being sent to an actor. The continuation is later used in the debugger to present the context of the future or message to the developer as a call stack, separate from the thread's call stacks.

Pharo debugger. With regard to threads and promises the Pharo debugger does not distinguish itself from other debuggers as it only supports thread switching (although only implicitly) and isolated thread breakpoints (via named threads and conditional breakpoints). We mention this explicitly because we use the Pharo debugger as the basis for the implementations introduced in Section 4.

3. A unified approach

To improve debugging of concurrent programs we propose that when an interrupt occurs in the child thread, *e.g.*, a breakpoint or unhandled exception, the developer be presented with a debugger that gives her access to the stacks of the child thread and its parents. Next we present what we want to achieve using a basic example, to illustrate the idea of a virtual thread and introduce a thread and debugger model supporting it.

3.1. A basic example

Let us start from the following code where a developer creates a child thread and then waits until it completes:

[32] http://www.html5rocks.com/en/tutorials/developertools/async-call-stack/

```
1 run
2   self createThread.
3   thread join.
4   ↑ self readThreadResult
```

Listing 1: The method #run asks for a new thread to be created, waits for it to exit, and returns the result from the computation performed in the thread.

```
5 createThread
6   thread := Thread new.
7   thread
8     start: #runThread
9     in: self
```

Listing 2: The method #createThread creates and starts a new thread.

The method #run sends #createThread, waits for the child thread to exit and returns the result that the thread has written to a shared variable. The child thread is initiated in the #start:in: method and executes the #runThread method. The frame for the method #createThread will no longer be on the stack at the point where the parent waits for the child to exit (line 3), since the parent will have returned from the method. Figure 1 shows this scenario.

Nevertheless, if the same scenario would occur in a sequential execution a developer would have access to the method #createThread. Figure 2 shows how the stack would look in a sequential scenario, where #createThread remains on the stack, instead of immediately returning. This is the view of full thread history that we aim to enable.

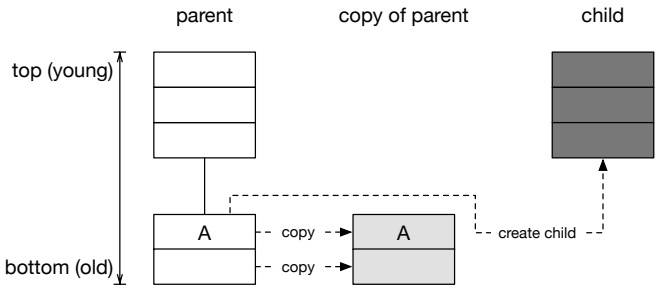

Figure 3: Stacks of two concurrent threads with a parent-child relationship after both threads executed several method calls concurrently.

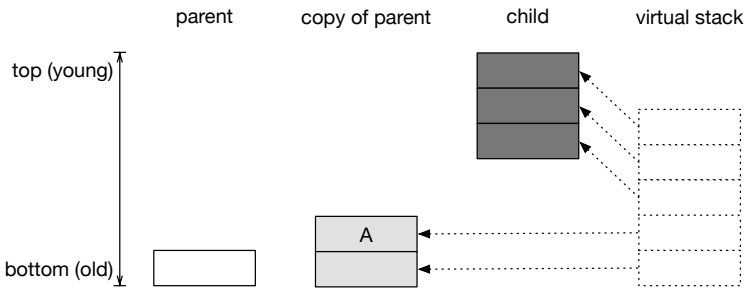

Figure 4: The virtual thread references a copy of a frame in the parent instead of the original, as the original may be returned from at any time.

3.2. Virtual thread

To give the developer a serialised and unified view the child and its parent threads we propose to save the call stacks of threads and merge them into a *virtual thread*. For a given thread the debugger can then display the call stack as if the child and its parents were a single sequential program. The virtual thread merges the call stacks of two actual threads at the place where the child thread was created in the parent.

Figures 3 and 4 show the call stacks of a parent thread and its child. *"A"* marks the activation record in the parent in which the child has been created. Figure 3 depicts the situation in which both parent and

child have performed some method calls concurrently, as illustrated in the example from Section 3.1. While creating the child, the parent also created a copy of itself, which contains copies of all the activation records that are part of the history of the child, *i.e.*, all activation records up to and including *"A"*.

When the child thread is interrupted, *e.g.*, due to a breakpoint or unhandled exception, we can construct a virtual thread from the child and the copy of its parent as shown in Figure 4. The bottom part of the virtual thread references the activation record copies from the parent thread, while the top part is comprised of the references to the activation records from the child. The bottom activation record of the child thread immediately follows the activation record *"A"*, in which the child was created. A debugger operating on such a virtual stack now enables users to navigate the activation records from two threads sequentially. This method is applied recursively so that the virtual thread consists of all the threads that are part of the thread history.

The operation that produces a copy of the call stack of a thread creates shallow copies of the activation records, *i.e.*, the objects referenced by a call stack and its copy are identical. However, as objects referenced from the original activation records are modified as part of the regular program execution, these changes will also be visible through the copies of the activation records (assigning a different reference to a field, however, will not be reflected in the copy). The call stack copy, therefore, does not represent the actual call stack at the time of the copy operation. While this may seem to be a point of confusion for users of the debugger, it is important to note that this behaviour is identical to the behaviour of a regular live debugger: every instruction executed in a live debugger can modify the object graph, and that change will be visible regardless of the activation record that is being viewed.

3.3. Interaction with the virtual thread

For debuggers, the interaction with a virtual thread is more complex than the interaction with a single thread. The actions developers can perform on activation records or threads must all be implemented to take into account that more than one thread is part of the call stack. To provide support for interacting with a virtual thread, we start from the observation that when the debugger displays the virtual thread, the state of a parent thread is not determined. The following states are possible:

- running,
- suspended,
- exited,
- blocked (waiting on another thread, or for an interrupt),
- blocked while waiting on the child that is currently being debugged.

For simplicity, we assume in our model that the copy of the parent's activation records can be inspected but cannot be acted upon. Consider the case from Figure 3 where the parent and the child have executed some calls concurrently. The activation record A from the virtual thread is then no longer on the stack; the parent, however, may still be executing, be waiting for another thread, or already have finished. In this case debugging actions on activation record A, (*e.g.*, restart, step into) are not defined. If we knew, however, that a parent was blocked and waiting for the child that is open in the debugger, we could use the active activation records of the parent in the virtual stack and the debugger would be able to operate on those activation records. In all other cases, letting the debugger perform actions on active activation records of the parent (or on copies of those actviation records) can lead to undefined behavior. The child thread however is not affected by this and we can perform debugging actions on it as if a developer had opened that thread in a standalone live debugger.

For completeness we assume that some activation records of the parent threads may be live, *i.e.*, not returned from, and safe to operate on, *i.e.*, the associated thread is waiting on the child thread. We use the term *actionable* to describe an activation record from which the thread has not yet returned, and that it is safe to operate on.

While we will describe the following debugging actions in terms of actionable and not actionable activation records, our current implementation of this model only works on actionable activation records belonging to the child thread. We took this decision as in the language in which we implemented the model we could not find a reliable and general solution for determining the state of the parent thread. A consequence of this decision is that the virtual thread will contain only copied activation records for the parent thread and these copies are not actionable.

Set breakpoint. Any of the methods referenced by the activation records on the stack may contain breakpoints. However, for activation records that are not actionable it is better to ignore the action and inform the user that the breakpoint would have no effect if she were to resume execution.

Step to next instruction. Stepping to the next instruction in an actionable frame is always possible but care must be taken when the next instruction is part of a different frame, as that frame may not be actionable. In addition, when crossing thread boundaries it may be necessary to perform additional actions to properly terminate a thread. This also applies to the related actions like "step to next instruction in current method" or "run to selected instruction".

Resume. Resumption requires the debugger to ensure that every thread that is part of the virtual thread and has been suspended because of the debugger is resumed. This can be achieved by resuming the topmost live thread, usually the child thread, which will automatically resume threads waiting for that thread. Special care may be necessary when dealing with remote threads that are part of the vitual thread (*e.g.*, such a thread may require explicit resumption).

Restart. For debuggers that support a restart action the action may be associated with a frame other than the top frame. For the frame that is associated with the action the debugger must check whether the frame is actionable.

Return value. Some debuggers support explicitly returning from a method with a user supplied value, which is only possible when the frame selected for this action is actionable. Additional care may be necessary when crossing thread boundaries.

3.4. Implementation details of the virtual thread

We assume the host infrastructure (programming language or IDE) provides a way to model the call stack of a thread as a list of activation records (Figure 5, *Part a)*). On top of this we introduce a UserThread class designed to store a copy of the active thread's call stack in an instance of ThreadHolder (Figure 5, *Part c)*). This could have been implemented on the class representing threads in the target environment, *e.g.*, Thread, but we wanted to guarantee a clean separation from the existing environment and prevent the creation of call stack copies for system threads, or other situations when no performance delay introduced by stack copying is acceptable. Through ContextHolder we provide a way to treat both live and dead activation records in a uniform manner. It further provides an explicit mapping between an activation record, its copy and the executing thread. ThreadHolder stores the stack of the virtual thread as a list of activation records that does not depend on explicit links between activation records.

Parent and child threads form a hierarchy in which every thread is a child thread (with the exception of the launch process thread) and can itself be the parent of other threads. If we indeed want to preserve the complete history of a thread we must therefore construct the virtual stack recursively so that the call stacks of all parents are represented in the virtual stack. To support this a ThreadHolder can have a link to a another ThreadHolder storing the parent of the current thread. Through this chain it is possible to construct a virtual thread containing the complete history of a thread, *i.e.*, the stack includes the activation records from all the parent processes of the child thread. The exceptions to this rule are instances of Thread for which we do not want to create call stack copies, meaning that the chain of ThreadHolder instances terminates when the parent is an instance of Thread.

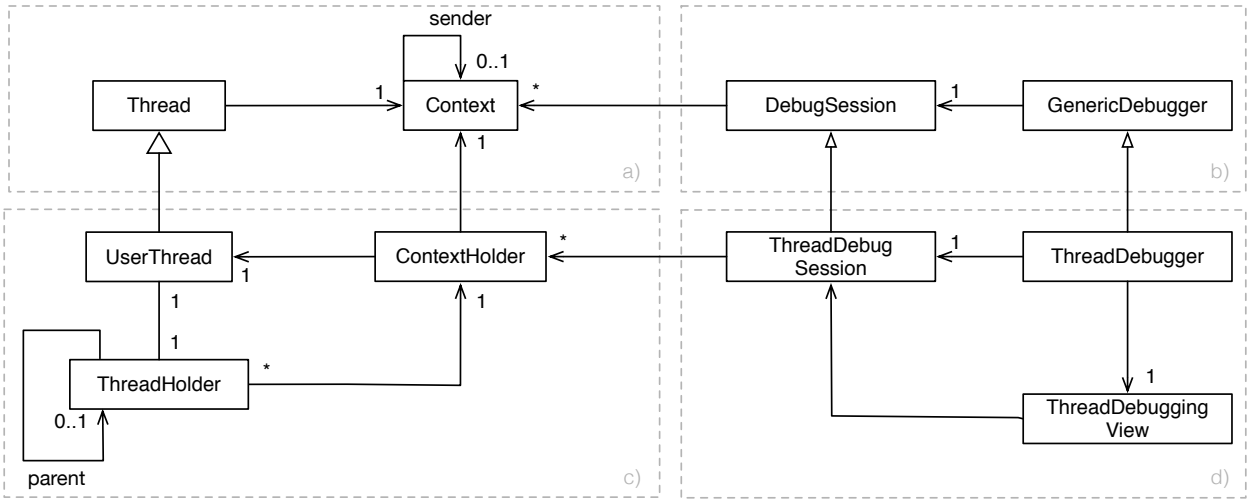

Figure 5: A debugging model for working with a virtual thread: *a)* and *b)* are elements that need to be present in the host infrastructure; *c)* is a model for representing a virtual thread; *d)* is an extension to the Moldable Debugger framework that can provide a debugger working with a virtual thread.

3.5. Debugger Model

To construct a debugger based on the virtual thread we rely on the Moldable Debugger framework [9]. The Moldable Debugger decomposes a debugger into a *Debugging Session*, encapsulating the logic for working with threads and activation records, and a *Debugging View*, consisting of a set of debugging widgets that operate on activation records. The Moldable Debugger framework allows us to propose a specialized concurrent debugger as a domain-specific extension, and plug it into the default debugger of the system.

To create a debugger that works with a virtual thread we require a custom ThreadDebugSession that allows the debugger to operate on a collection of ContextHolder instances, *i.e.*, the stack of the virtual thread. ContextHolder instances belong to a ThreadHolder instance. The stack of the virtual thread can contain ContextHolder instances of different types belonging to different types of ThreadHolder instances. This is necessary if the host system has different kinds of threads or communicates with external systems that have a different thread model. The debugging session also controls every operation on the stack to guarantee that activation records are being mapped correctly to the thread they belong to.

Debuggers working with virtual threads need to support dynamic thread switching and activation record selection, as opposed to hard coding a parent and a child thread. We achieve this by having a *Debugging View* that selects the right widgets (*e.g.*, code editor, object inspector) by dispatching through ContextHolder instances. For example, when selecting an activation record representing a dead activation record from the parent in the debugger, the debugging view can select an object inspector widget that knows how to display and interact with that type of activation record.

4. Exercising the model

In this section we take the thread and debugger model presented in Section 3 and exemplify four different use cases in which we apply the model to improve debugging. For each use case we describe the implementation that is required to address a running example. The running example illustrates a static analysis where part of the computation is performed concurrently in a separate child thread.[33]

[33] A version of the four prototypes, including installation instructions and code for benchmarks can be found at http://scg.unibe.ch/download/moldabledebugger/concurrentdebuggers.zip

4.1. Running example

Static analysis of source code is an expensive task which, to improve performance, can be run in another thread, either local or remote. Through this example we simulate the situation in which part of the analysis is done concurrently. Listings 3 through 5 show the example code for launching a task using a worker (child thread) that is executed concurrently. The method #runAnalysisOn: must receive as parameter a relative or absolute path to an existing directory containing the source code that needs to be analized, such as *"sourcecode"* or *"/private_repositories/sourcecode"*. The program performs the following steps:

1. Create an absolute path based on the *inputPath* parameter (line 12). When the input path is already absolute, *absolutePath* will have the same value as *inputPath*.
2. Initialize the analysis in a concurrent worker with the absolute path to the directory as input. The new worker will attempt to access the directory at the given path.
3. Inform the user that the analysis is executing (line 14).
4. After informing the user, the parent thread, which launched the worker, returns from the method #runAnalysisOn: and waits for the worker to complete. The activation record in which the worker was created will no longer be on the call stack of the parent and the variable *absolutePath* cannot be inspected.

```
10 runAnalysisOn: inputPath
11     | absolutePath worker |
12     absolutePath := self absolutePathFrom: inputPath.
13     worker := self runAnalysisConcurrentlyOn: absolutePath.
14     self informUserToWait.
15     ↑ worker
```

Listing 3: The method StaticAnalysisService>>#runAnalysisOn: is the entry point to the static analysis service.

```
16 runAnalysis
17     | worker |
18     worker := self runAnalysisOn: self getDirectoryFromUser.
19     self checkResultFromWorker: worker
```

Listing 4: #runAnalysisOn: starts the analysis in a worker thread and waits for the worker to complete in #checkResultFromWorker:. The activation record that created the worker is no longer on the call stack after returning from #runAnalysisOn:.

```
20 self runAnalysisOn: '/sourcecode'.
21 self runAnalysisOn: 'sourcecode'.
```

Listing 5: Possible ways to invoke StaticAnalysisService>>#runAnalysisOn:.

When the concurrent worker attempts to access a directory that does not exist it will fail. The question that a developer debugging this problem must answer is, why did the directory not exist? Assuming that a directory exists at *"/private_repositories/sourcecode"* there are two locations for possible failure in the example:

1. The original input was erroneous, as in line 20. #absolutePathFrom: will not modify *inputPath* since '/sourcecode' is already an absolute path. The value of *absolutePath*, which is being passed to the worker, is '/sourcecode'.
2. The original input was correct (line 21), however, the invocation of #absolutePathFrom: returned an erroneous path, *i.e.*, '/repositories/sourcecode' instead of '/private_repositories/sourcecode'. The value of *absolutePath*, which is being passed to the worker, is '/repositories/sourcecode'.

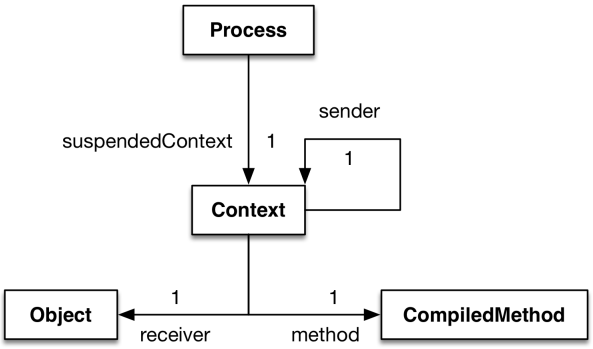

Figure 6: UML diagram of threads (Process) and their chain of activation records (Context) in Pharo.

4.2. Case Study 1: Local threads

We first discuss the application of a virtual thread for when the parent and the child worker are Pharo threads. We start by clarifying the terminology used in Pharo to refer to a thread and then show how to address the running example.

4.2.1. Threads in Pharo

In Smalltalk-80, and hence in Pharo, the class Process represents green threads, not processes as its name suggests. We will refer to instances of Process as threads to avoid confusion with the POSIX definition of process. The UML diagram in Figure 6 shows a simplified schematic of how the threads in Pharo are connected to their activation records. Activation records are represented by instances of class Context, which is the equivalent of the class MethodContext in Smalltalk-80, and are linked to each other through the *sender* field of Context. Every activation record holds a reference to the *method* whose activation it represents (CompiledMethod). The *self* pseudo-variable is bound to the object the method was invoked on, its *receiver* [15]. This thread model satisfies the requirements needed for constructing a virtual thread, as described in Figure 5 *a)*.

4.2.2. Instantiating the model

For the implementation of the debugger we follow the model presented in Figure 5 and create a *Thread* debugging extension to the Moldable Debugger framework. When instantiating the model we can use the same type of ThreadHolder and ContextHolder for both the parent and the worker thread as they are both standard Pharo threads. ContextHolder instances from both holders will also point to standard Context instances.

4.2.3. Addressing the running example

To address the running example we first need to provide an implementation of the methods that work on a local thread as the worker. Listing 6 shows the method #runAnalysisConcurrentlyOn:, which creates the worker thread, and #checkResultFromWorker:, where the parent awaits the completion of the worker thread using #joinWithin:. The worker thread is allowed to run for one hour; a failure in the worker will open a debugger.

```
22 runAnalysisConcurrentlyOn: absolutePath
23     ↑ [ self privateRunAnalysisOn: absolutePath ] newUserProcess resume
24
25 checkResultFromWorker: aWorkerThread
26   aWorkerThread joinWithin: 1 hour
```

Listing 6: #runAnalysisConcurrentlyOn: delegates the actual computation to a concurrent worker thread. #checkResultFromWorker: waits for the worker to complete using #joinWithin:

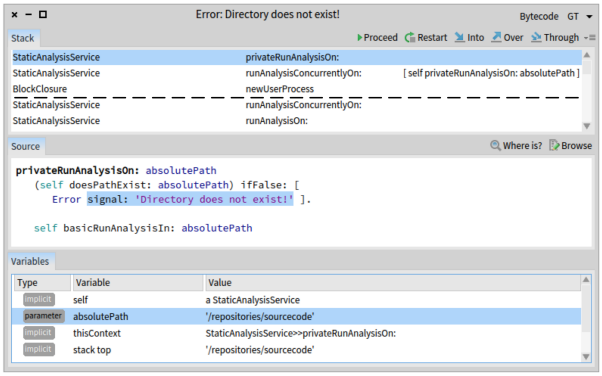

Figure 7: The top activation record is part of the worker (child thread) and shows where the error was signalled.

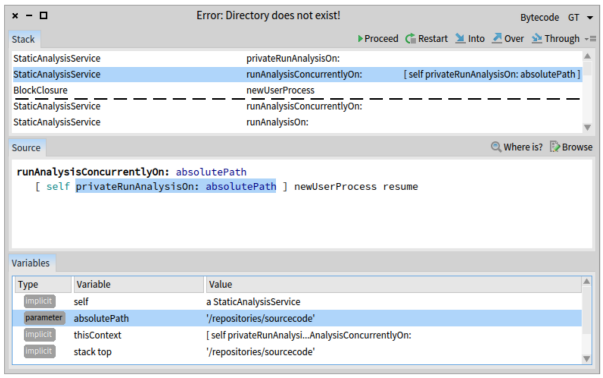

Figure 8: The selected activation record shows the method #runAnalysisConcurrentlyOn:, the activation record is part of the worker (child thread). The variable *inputPath* is not visible.

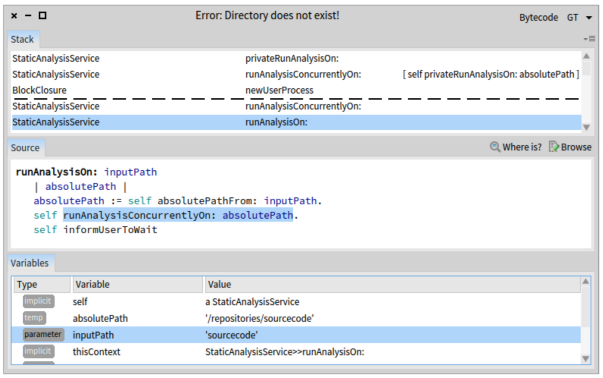

Figure 9: Only in the activation record of #runAnalysisOn:, which is part of the parent thread, is the variable *inputPath* visible and can be compared to the value of *absolutePath*.

Figures 7, 8 and 9 show the thread debugger after an exception has been signalled in the worker (child thread). The point where the child thread has been joined to its parent in the virtual call stack is highlighted by a dashed line (child thread above, parent below the line). In Figure 7 the activation record in which the exception was signalled is shown. The activation record selected in Figure 8 is the last activation record of the child thread. The variable *inputPath* is not part of this activation record and cannot be compared with the method argument *absolutePath*. Figure 9 shows the activation record that receives the method argument *inputPath* that the user supplied. This activation record is part of the parent thread and has access to both the *inputPath* and *absolutePath* variables. Thus, using this joint view of the two threads a developer investigating this problem can easily navigate both stacks to investigate the source of the problem.

4.2.4. Implementation details

In this section we emphasize the technical details of how the debugger copied the stack of the parent thread. We first need to offer developers an API for creating a thread that can copy the stack of its parent. For this we extend the BlockClosure class modeling anonymous functions with the API method #newUserProcess (Listing 7), which creates a new unscheduled user thread. Developers can then create a user thread as shown on line 23. We then need to store the copy of the current thread's call stack. We add this functionality in the method UserProcess>>#initialize that is sent upon creation of the instance. This method creates and stores an instance of ProcessHolder.

```
27 newUserProcess
```

```
28      ↑ UserProcess
29        forContext:
30            [self value.
31            Processor terminateActive] asContext
32        priority: Processor activePriority
```

Listing 7: The method BlockClosure>>#newUserProcess creates instances of UserProcess. UserProcess is the holder that wraps an instance of Process

```
33  initialize
34      super initialize.

36      masterProcessHolder := ProcessHolder for: Processor activeProcess
```

Listing 8: UserProcess>>#initialize is executed during creation of UserProcess instances and stores a reference to the active thread, which will become the parent of the new thread.

The ThreadHolder will then create a copy of the stack inside the constructor #initializeWithProcess: by calling the method Context>>#copyStack on line 40 (Listing 9). The method #copyStack is part of the Pharo language. The pseudo variable thisContext, provided also by the Pharo language, refers to the activation record of the current method, which is also the topmost activation record of the current thread (the method #stack simply creates an array from the linked list of activation records). In addition, #initializeWithProcess: stores a reference to the thread and creates an instance of ContextHolder for every activation record (lines 41 to 46).

```
37  initializeWithProcess: aProcess
38      | stackCopy |
39      process := aProcess.
40      stackCopy := thisContext copyStack stack.
41      contextHolders := thisContext stack withIndexCollect: [ :context :index |
42          ContextHolder
43              forProcess: aProcess
44              context: context
45              andCopy: (stackCopy at: index)
46              withIndex: index ].
```

Listing 9: ProcessHolder>>#initializeWithProcess: is executed during instance creation of ProcessHolder and creates a copy of the call stack of the thread passed as argument.

4.3. Case Study 2: Local promises

We have shown how our model works with local threads that employ simple synchronisation. Next we apply our model for local threads that use a different synchronisation mechanism, *i.e.*, promises. This shows that even more complex thread synchronisation is covered by the proposed model.

4.3.1. TaskIt

TaskIt is a library for Pharo that abstracts execution and synchronisation of concurrent tasks. In particular, it provides an implementation of promises, called *futures* in TaskIt. Based on our model, we have implemented a debugger for TaskIt that retains the history of a promise so that the debugger can present the user with a virtual call stack in which both the creation of the promise and its resolution are visible.

4.3.2. Instantiating the model

We instantiate the model by creating a dedicated type of UserThread and ThreadHolder, *i.e.*, PromiseThread and PromiseThreadHolder (Figure 10), instead of reusing the same as for local threads. Hence, for the running example, the virtual thread will have at the top activation records of type PromiseContextHolder that are aware that the thread they belong to is a promise (Figure 11). We took this decision as not in all cases the thread that creates the promise is also the thread that executes the promise. For example, TaskIt uses a thread pool and this allows us to handle that case.

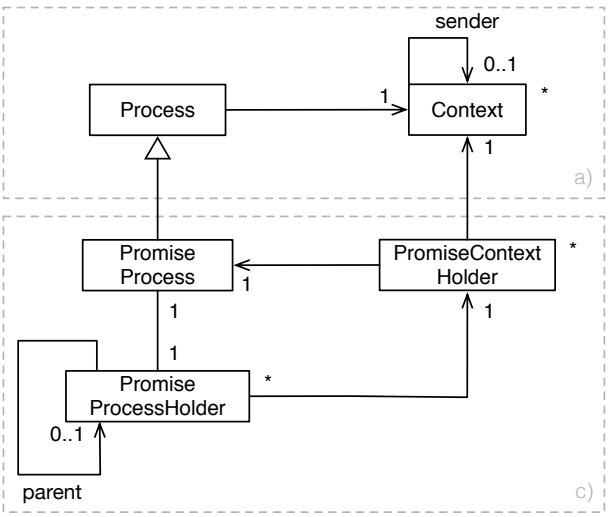

Figure 10: A model for building a the part of a virtual thread that references a promise.

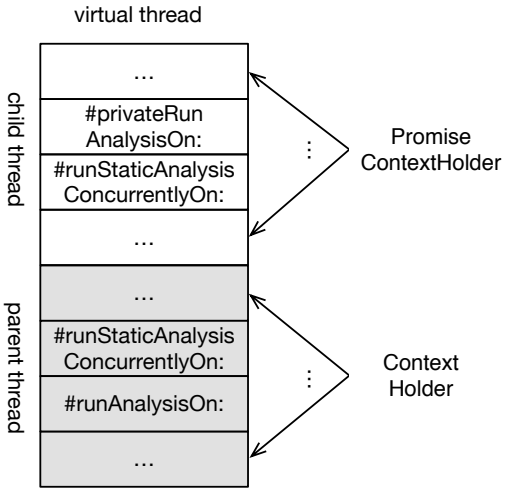

Figure 11: Forming the virtual thread for a local promise for the running example. The top activation records belong to the child thread and are modeled using a different type of context holder than the activation records of the parent thread.

4.3.3. Addressing the running example

The method #runAnalysisConcurrentlyOn: is implemented as shown in Listing 10. The TaskIt promise is assigned to the local variable *worker* on line 18 (Listing 4) and #checkResultFromWorker: waits for the worker to complete the analysis.

```
47 runAnalysisConcurrentlyOn: absolutePath
48   ↑ [self privateRunAnalysisOn: absolutePath] future.

50 checkResultFromWorker: aWorker
51   aWorker
52     onFailureDo: [ Error signal: 'Analysis failed' ];
53     waitForCompletion: 1 hour.
```

Listing 10: The message #future creates a promise that is returned as the worker on line 18 (Listing 4). The promise is set up to launch a debugger in case of failure and to time out after one hour.

4.3.4. Implementation details

The implemenation for TaskIt then differs in only one main aspect from the general implementation described in Section 4.2. The thread in which the promise is being executed is not created by the thread that instantiated the promise but by a scheduler thread that manages a global task queue. Therefore, the parent-child relationship between the thread executing the promise and the thread waiting for the promise to complete has to be established by the debugger.

To construct this mapping we use FutureExecution provided by TaskIt that represents the execution of a promise and is instantiated within the thread that creates the promise. Upon initialization of an instance of FutureExecution we create a PromiseThreadHolder instance that copies the active thread. When the future is being executed in the execution thread, we set the PromiseThreadHolder instance on the active thread, thus linking the two threads in a parent-child relationship.

4.4. Case Study 3: Remote promises

Remote promises are functionally equivalent to local promises but add the challenge of dealing with remote communication. We show that our model can also deal with this additional complexity.

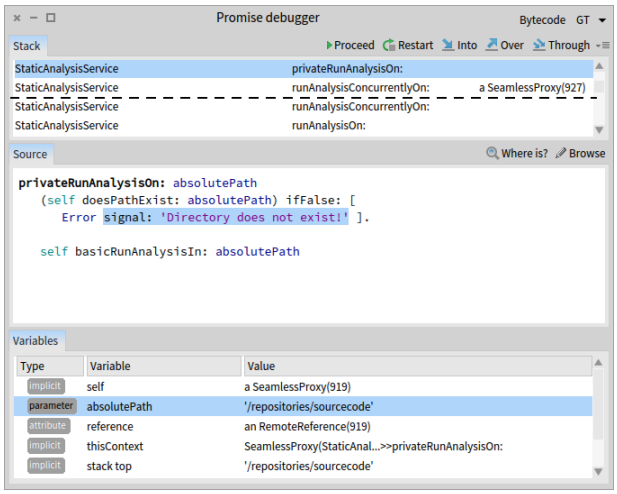

Figure 12: Debugging a remote promise executed using Seamless. The top activation record is part of the worker (remote child thread) and shows where the error was signalled.

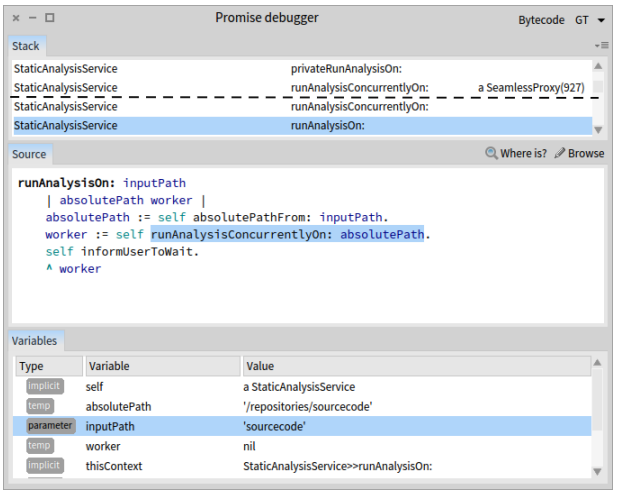

Figure 13: Debugging a remote promise executed using Seamless: only in the activation record of #runAnalysisOn:, which is part of the parent thread, is the variable *inputPath* visible and can be compared to the value of *absolutePath*.

4.4.1. Seamless

In previous work [20] we presented the prototype of a promise library for Pharo that is capable of executing remote promises using Seamless [27], a framework for distributed computing. To provide transparent communication with remote threads, Seamless can use proxies, which is what our model relies upon. This aspect of Seamless was leveraged in Mercury [28], a tool for providing remote debugging solutions using reflection, which, however, did not look into concurrent aspects of debugging. In this section we apply our previous work on the model presented in this paper and use it to show that our model covers the case of remote execution: the debugger can show a virtual stack comprised of a local thread, which created the promise, and a failed remote thread.

4.4.2. Instantiating the model

We instantiate the model as in Section 4.3. The main difference in this case is that the worker thread is in a remote address space. To account for this fact we rely on proxies. Hence, a PromiseContextHolder instance will reference a proxy pointing to a remote Context instance. The thread holder will also store a proxy to the instance of the remote child thread that is debugged.

4.4.3. Addressing the running example

We develop an implementation of a remote promise that can take as a parameter a Seamless connection. The computation that will be performed remotely is expressed on line 55 using an anonymous function. This function directly accesses the parameter *absolutePath* belonging to the context of the containing method, as well as *self*, *i.e.*, the object that received the message #runAnalysisConcurrentlyOn:. We can express the remote computation this way as Seamless automatically detects and serializes attributes, local variables and parameters defined in the outer context of the anonymous function. Figures 12 and 13 show the debugger after an exception has been signalled in the remote worker.

```
54 runAnalysisConcurrentlyOn: absolutePath
55   ↑ [self privateRunAnalysisOn: absolutePath] remotePromiseOn: self connection.
56
57 checkResultFromWorker: aWorker
58   aWorker
59     openDebuggerOnError: true;
60     timeout: 1 hour;
61     value.
```

Listing 11: #runAnalysisConcurrentlyOn: creates a remote promise that is returned as the worker on line 18. The promise is set up to launch a debugger in case of failure and to time out after one hour.

4.4.4. Implementation details

The parent of the remote thread executing the promise is not the local thread that created the promise. Hence, as we did for TaskIt, we need to construct the parent-child relationship between the local parent and remote child thread manually. Upon an exception in the remote thread we serialise a reference to the failed thread and pass it back through the communication channel.

While our promise implementation is independent of the location of execution (the promise itself is always a local construct) working with remote objects requires extra effort. There are two locations where the details of remote communication become relevant to us. The first is the case where an exception occurs in the remote thread. We must ensure that all the information we need will be retained and that we send back the correct proxies. The second location is the setup of the debugger where we have to create local objects for certain proxies for performance reasons. The proxies concerned are those that receive many messages due to UI interaction.

4.5. Case Study 4: Remote object-oriented database

In the previous three use cases both the parent and the worker were Pharo threads. Hence, the debugger can always treat the worker as consisting of a collection of local or remote instances of type Context. This is not the case if the worker is being executed in a language that has a different model for representing the call stack. To verify that the proposed model addresses this aspect we also applied it for the use case of a remote worker thread executing in GemStone/S, a system with a model of threads and call stack that differs from the model used by Pharo.

4.5.1. GemStone/S

GemStone/S is a distributed, object-oriented database. It uses Smalltalk syntax to express computations over the database, however, it has its own independent virtual machine, implementation of the language, and model for working with threads.[34] GemStone/S aims to make it transparent to developers that the objects they are working with could be distributed over a cluster of machines.

The process and thread architecture of GemStone/S is described in its programming guide,[35] however, the details of that architecture are not relevant for this work. We rely only on the fact that threads are reified on the remote system as instances of type GsProcess, and that there is an API, provided by the virtual machine, for accessing the call stack and the activation record details of a given thread. Nonetheless, activation records have a different representation in GemStone/S. Hence, a one to one mapping is not possible. Also, Seamless, the library we used to enable communication between a local and a remote Pharo image in Section 4.4, does not support GemStone/S.

4.5.2. Instantiating the model

We follow the same idea as in the case of Section 4.4, however, as Seamless does not work in GemStone/S we needed an alternative communication library. We selected to use the communication facilities provided by the project gt4gemstone.[36] To link this library into our prototype we provide both a custom implementation for a GemstoneContextHolder that knows how to communicate with an activation record from a remote GemStone/S system, and a thread holder specific for GemStone/S. We needed to specialize the model this way as the implementation discussed in Section 4.4 is tied to the Seamless communication library.

[34] The GemStone/S virtual machine can further execute Ruby code
[35] https://gemtalksystems.com/products/gs64/
[36] https://github.com/feenkcom/gt4gemstone

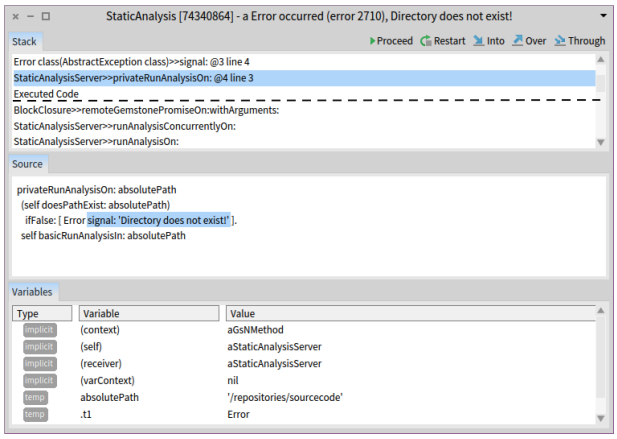

Figure 14: Debugging a remote promise executed on GemStone/S: the top activation record is part of the child thread and shows where the error was signalled in the remote thread. The variable *inputPath* is not accessible in the remote child thread.

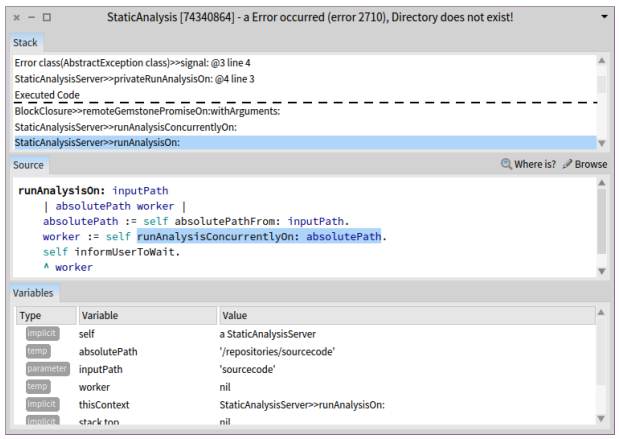

Figure 15: Debugging a remote promise executed on GemStone/S: the variable *inputPath* is only visible in the activation record of #runAnalysisOn:, which is part of the parent thread, and can be compared to the value of *absolutePath*.

4.5.3. Addressing the running example

As in Section 4.4 we develop an implementation of a remote promise that can take as parameters a GemStone client and a set of arguments. The computation is expressed using an anonymous function. The promise serializes the arguments and executes the computation using the provided client, which handles the communication with the GemStone/S system. Listing 12 shows the code that creates the worker client. The code for waiting for the results of the computation (Listing 13) is the same as the one from Listing 11.

```
62 runAnalysisConcurrentlyOn: absolutePath
63   | gemstoneClient |
64   gemstoneClient := GtGsMinimalClient forConfig: 'StaticAnalysis'.
65   ↑ [ :analyser :targetPath |
66       analyser privateRunAnalysisOn: targetPath ]
67           remoteGemstonePromiseOn: gemstoneClient
68           withArguments: {self, absolutePath}.
```

Listing 12: The message #remoteGemstonePromiseOn:withArguments: creates a promise that will be executed on GemStone/S and is returned as the worker on line 18 (Listing 4).

```
69 checkResultFromWorker: aWorker
70   aWorker
71     openDebuggerOnError: true;
72     timeout: 1 hour;
73     value.
```

Listing 13: The worker is allowed to run for at most one hour. Upon failure of the worker a debugger will be opened.

Given that the analysis is being executed in a database we assume that the database stores a set of files and folders and the analysis checks whether an entry with the given path exists in the database. When no such entry exists we obtain the debugger shown in Figure 14. Here, we see that, as in the other use cases, the parameter *absolutePath* has the value '/repositories/sourcecode'. Figure 15 shows the activation record of the parent that includes both the user supplied parameter *inputPath* and the computed absolute path *absolutePath*.

4.5.4. Implementation details

To enable communication between a Pharo image and a remote GemStone/S database, gt4gemstone relies on a shared C library that is part of the GemStone/S database.[37] In Seamless the communication between Pharo instances uses the TCP protocol. When an error then occurs in the remote database the communication infrastructure sends a notification to the Pharo client. The client then opens a debugger, which in turn retrieves the call stack of the failed thread through the remote session.

In the debugger based on Seamless remote activation records are modeled as proxies: when a UI widget needs any kind of information it uses a proxy to make a request for that information. In the case of GemStone/S, to minimize the number of requests to the database, all widgets register the information that they need and every time an activation record is selected all the necessary information is retrieved in a single request. We chose this design as the operating system process running a GemStone/S database is by default in idle mode, and is only activated to perform a computation when a request is received.

Instead of passing the arguments required for the computation in the remote promise explicitly, they can be passed as part of an anonymous function, *i.e.*, a closure. This is supported in GemStone/S, however, the current implementation of the communication library can only reference local variables and parameters. Hence, the current example could be written as shown in Listing 14; all referenced variables would be serialized and passed by value.

```
74 runAnalysisConcurrentlyOn: absolutePath
75   | analyzer |
76   gemstoneClient := GtGsMinimalClient forConfig: 'StaticAnalysis'.
77   analyzer := self.
78   ↑ [ analyzer privateRunAnalysisOn: absolutePath ] remoteGemstonePromiseOn: gemstoneClient.
```

Listing 14: Expressing a remote promise for GemStone/S by referencing variables outside of the context of the anonymous function (closure) containing the computation.

4.6. Summary

In this section we showed how to apply the model for a virtual thread in four different use cases. These ranged from simple threads to remote promises. In each case we instantiated the presented model resulting in a debugger that could display the entire history of a child thread. Especially in the case of local promises (Section 4.3.1) and remote promises for GemStone/S (Section 4.5.1) the instantiation of the model required implementation details specific to the used libraries (*i.e.*, TaskIt for local promises, gt4gemstone for remote promises for GemStone/S). Differences appeared due to how libraries chose to fork a thread or enable communication with a remote image. Abstracting these aspects would require changes to the underlying libraries. Given the wide range of libraries for supporting concurrent programming we conclude that a model for a virtual thread should allow for flexibility in specifying holders for threads and activation records.

5. Performance overhead and memory consumption

The manipulations we perform on threads come at a cost. Additional costs when debugging are acceptable as long as the tools remain subjectively usable. It is important however, that these additional costs do not lead to a performance degradation in the general case. In this section we investigate the performance overhead of copying activation records, as well as the memory overhead incurred by the copied records in our current prototype implementation.

5.1. Context reification

Creating a copy of a call stack means creating a copy of each of its activation records. We consider the impact of this operation with respect to performance and memory, exemplified by our implementation in Pharo. Our analysis focuses on the 32-bit version of the virtual machine; a 64-bit virtual machine is currently in development.

[37] https://github.com/GsDevKit/GsDevKit_home

Activation records in Pharo are represented by instances of Context, equivalent to the MethodContext in Smalltalk-80. The virtual machine for Pharo, however, only creates contexts on demand [10], except for cases where a Context instance is needed anyway (Miranda [25] provides a list of such situations). Hence, creating a copy of an activation record in general entails the creation of two Context instances. To calculate how much memory is needed per instance we need to know how many bytes are needed to represent the object structure of contexts and what information is being stored.

Every regular object requires a header of 8 bytes, which is also true for contexts [4, 26]. Context defines the fields *stackp*, *method*, *closureOrNil* and *receiver*, and inherits *sender* and *pc*, each field being four bytes long. Context is also a so-called "variable class", meaning it has a fixed number of unnamed, indexed slots. In contexts, these slots represent a stack with fixed capacity that is used to implement a stack machine for the activated method. Elements consumed by the bytcode instructions are pushed onto the stack, while values produced by the instructions are popped from it. The number of slots, *i.e.*, the stack capacity, depends on the "large context flag" bit [15] of the CompiledMethod object header that is being activated, the value of which depends on the number of method arguments, literals, local variables and temporary values of the method. The flag is set to "large" by the compiler when the stack needs to hold more than a certain amount of elements at a time.

In the current virtual machine for Pharo the two states of the "large context flag" represent contexts of 16 (small context) or 56 (large context) variable slots. In a fresh Pharo 6 image (build 60143) the number of CompiledMethod instances is 94824, 344 (0.36 %) of which have the large context flag set. Using a single bit flag instead of the exact number of variable slots, which would require multiple bits, reduces the memory consumed by instances of CompiledMethod.

We can now determine the size of small Context instances:

$$\begin{aligned} & 8 \text{ bytes for object header} \\ + \; & 6 \times 4 \text{ bytes for instance variables} \\ + \; & 16 \times 4 \text{ bytes for variable slots} \\ \hline = \; & 96 \text{ bytes} \end{aligned}$$

Large Context instances have 56 instead of 16 indexed slots:

$$\begin{aligned} & 96 \text{ bytes} \\ + \; & (56 - 16) \times 4 \text{ bytes for variable slots} \\ \hline = \; & 256 \text{ bytes} \end{aligned}$$

The contents of contexts do not require additional space when copied, as references are already accounted for in the field and variable slot sizes, and immediate values, such as integers, do not require additional memory allocation. In general, creating a copy of a context therefore requires 2×96 bytes = 192 bytes of additional storage in the best case, 2×256 bytes = 512 bytes in the worst one. Knowing that typical call stacks have sizes of tens or hundreds of activation records [33, 11], we can estimate an upper bound of 1000×512 bytes = 512 kB of additional memory required for large call stacks with large methods. Even 512 kB are not much in comparison with the memory bound by the associated object graph.

5.2. Bound memory

Memory bound by call stack copies may pose a bigger problem than the additional memory required for new contexts as the amount of memory occupied by the object graph rooted in a given context is potentially many times larger than the memory needed for the context itself. Contexts usually have a short life and objects referenced only from a context, such as those referenced by temporary variables, will be recycled by the garbage collector after a short time. This is not the case when we create copies of contexts. The memory occupied by objects that would normally no longer be needed can only be reclaimed when the child thread terminates. This is a problem for Pharo in particular because the 32-bit virtual machine can only allocate

garbage collection time	small context			large context		
	average [ms]	median [ms]	max [ms]	average [ms]	median [ms]	max [ms]
included	2272.1	2267	2366	2351.4	2265.5	2898
excluded	592.7	592.5	615	586.5	586	594

Table 2: Benchmark for copying a call stack of 100 000 activation records, small (16 variable slots) and large (56 variable slots). The times are given in milliseconds and shown for bare computation time and computation including time needed for intermittent garbage collection.

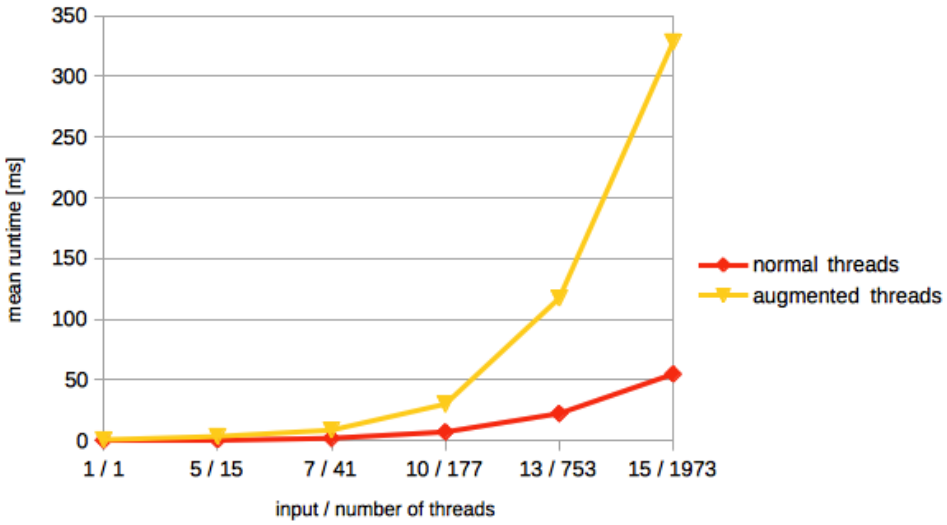

Figure 16: Mean runtime of computing Fibonacci numbers with normal threads and augmented threads.

a certain amount of memory (around 2 GB, depending on the operating system).[38] The memory bound by context copies is a problem even when the debugger is not being used, as we cannot know in advance whether or not a thread will be opened in a debugger and we therefore must create call stack copies proactively.

5.3. Computational overhead

The creation of a UserProcess instance incurs a performance penalty due to the copy operation on the active thread. With a stack of 100 000 activation records the average and median times needed to copy the complete stack are in the range of 600 milliseconds without garbage collection and between 2300 and 3000 milliseconds with garbage collection, as shown in Table 2.[39]

It is interesting to see that the size of the context does not seem to have any effect on the execution time even though it is clear that a larger context will require more operations to create a copy. It is possible that other operations mask the actual time it takes to create the copies, which would also explain the uniformity of the obtained results. Miranda [25] for example, describes how context reification in the VisualWorks virtual machine[40] can cause large portions of memory to be rearranged.

Given that stacks typically contain fewer than 1000 activation records [33, 11], performance clearly is not a problem for single threads. For programs that make heavy use of threads, however, even a small

[38] Note that this limitation is inherent to 32-bit memory management
[39] The complete output from the benchmark runs can be found in Table B.3, Table B.4 and Table B.5
[40] VisualWorks is Smalltalk implementation whose virtual machine is related to the Pharo virtual machine.

overhead may be prohibitive. To investigate the impact of the computational overhead in relation to the number of threads we performed benchmarks on a parallel implementation of the algorithm for computing Fibonacci numbers. The implemented algorithm computes the Fibonacci number $fib(n)$ for the input n by creating a thread for every Fibonacci number. Hence, the number of threads t is given by the equation $t = 2 * fib(n) - 1$. We selected five inputs for the program ($n = 1, 5, 7, 10, 13, 15$). For example, for input $n = 15$ the computation creates 1973 threads. This implementation of the algorithm[41] is relevant for measuring the performance overhead of our implementation, as the actual child threads only perform additions. With algorithms that perform more computations in the child threads the overhead of copying those activation records can be hidden by the actual cost of the computation.

To perform the benchmarks for each input value, we run the algorithm both with and without our prototype for recording the history of parent threads. The benchmarks were performed on an Apple MacBook Pro (early 2011), with a 2.2 GHz Intel Core i7 CPU and 8 GB of 1333 MHz DDR3 memory. The operating system was macOS version 10.13 Beta (17A291j). We obtained ten data points, each of which was computed as the average of five runs (after two warm up runs) during a single virtual machine invocation, following the recommendations of Georges *et al.* [14]. We used ReBench (`https://github.com/smarr/ReBench`) to make the benchmarks reproducible. The ReBench scripts for reproducing the benchmarks can be found in AppendixB.

Figure 16 shows the results of the experiment. We can observe that the overhead of the stack copy operation becomes a large part of the total computation time when many threads are used. For 177 threads the overhead is already larger than the total computation time with normal threads. For 1973 threads recording their histories slows down the execution by a factor of 6.5.

6. Future work directions

In this section we discuss directions for future work in improving the practical performance of our approach, as well as ways to further improve debugging of concurrent programs.

6.1. Challenges for a practical solution

As shown in Section 5.3, especially for applications that use more than 100 threads, the computational overhead of copying activation records is no longer negligible. For these cases virtual machine support is one possible solution for improving the performance of the proposed approach.

Operations executed directly by a virtual machine take less time than interpreted instructions. The most expensive—because it is the most frequent—operation in our implementation, is Context>>#copy, which already is a primitive. However, contexts are copied individually and the repeated sends of #copy occur in the interpreted space. Moving the complete stack copy procedure into the virtual machine would certainly improve performance. The following ideas could be used by an implementation in the virtual machine:

- the call stack could be duplicated lazily (only copy activation records that are being returned from);
- operations could be split into chunks to ensure responsiveness;
- context reification could be optimised, for example by delaying reification or reifying only the copies.

A second challenge in our current implementation is the memory bound by the call stack copies. A possibility to mitigate this problem may be to compress the memory of objects that are only being referenced from copied call stacks. Since these objects will only be accessed from the debugger, the additional time to decompress the memory should not be of concern. The additional time required for compression on the other hand may have a negative impact on performance, so that moving the stack copy procedure into the virtual machine may indeed become necessary. A similar alternative is to use Marea [23], a library that can be used to write graphs of objects to secondary memory and reload them on demand. Another option might be to

[41] The code of the algorithm is given in Listing 16.

reclaim memory from copied call stacks when necessary, starting with the oldest copies. Finally, a promise whose resolution was successful, *i.e.*, no unhandled exceptions ocurred during its execution, could discard the references to the copies of its parents activation records as soon as it has completed. This would free memory early in the case of a promise whose value is not being accessed immediately after its resolution.

6.2. Supporting other concurrent models

In the current work we presented the model of a virtual thread in the context of basic threads. We further showed how to apply it to local and remote promises. Nonetheless, in practice there exist many other concurrent models as discussed by Marr and D'Hondt [22]. Examples include processes, actors and active objects. We assume that the presented model for a virtual thread can also be applied to these concurrent models. Nonetheless, future work is needed to confirm that the actual implementation can fit the presented model or to detect concurrent models that need specialized concepts.

6.3. Showing more types of relations between parent and child threads

As mentioned in Section 1 there are more types of relations between threads useful during debugging apart from thread histories. For example, a parent thread can fork more child threads (either as direct threads or using another concurrency model like promises) and then wait for those threads to finish using a synchronization mechanism (*e.g.*, semaphores, locks, mutexes). Child threads can in turn do the same resulting in a hierarchy of threads. Concurrency bugs in this case could require a global view of this hierarchy. However, the virtual thread model that we discuss in this work only shows the relationship between a child and its parents. A future work direction consists in investigating whether or not this model can be extended to debug parents with multiple child threads. Debugging a thread that creates and joins two or more threads would bring new challenges due to activation records that join multiple treads.

7. Related work

Section 2 already presented a discussion of features for debugging concurrent programs in several debuggers. In this section we look at other related approaches, as well as alternative approaches for debugging concurrent threads.

Zhang *et al.*, for their *Directive-based lazy futures* implementation in Java, proposed a serialised view of the events of both the child thread and its parent [38]. In contrast to our approach, the serialised view is not created on demand but is a side-effect of their stack-splitting strategy: the virtual machine (a customised Jikes VM) creates a future by copying the current thread and marking the current activation record as the bottom of the stack. The child thread thus contains all the activation records from before its creation (the mark is reversible). Their implementation of futures is based on the idea that the virtual machine can decide to create threads for long running computations, hence a future is implicit and not guaranteed to run in a separate thread at all. It also means that their implementation does not support remote futures because the location of execution cannot be configured.

A different approach to debugging concurrent threads is *omniscient debugging*, which provides a way to navigate backwards in time within a program's execution trace [30]. This enables developers to record a scenario exhibiting a specific problem and replay it until the point where the child thread is created. For example, Jockey [31] is a record/replay tool for debugging Linux programs that records invocations of system calls and CPU instructions. It further supports checkpointing to diagnose long-running programs efficiently. Once a trace has been recorded, Jockey enables the developer to replay each process within a traditional debugger. Jockey does not support deterministic replay of a system, nor does it aim to provide developers with a unified view of multiple related threads.

Another debugger that allows record and replay is rr.[42] Developed to ease the debugging of Mozilla Firefox, rr records a group of Linux user-space processes and captures all inputs to those processes from the

[42] http://rr-project.org/

kernel, as well as any nondeterministic CPU effects performed by those processes. Hence, it can support a deterministic replay of a system, instead of just replays of single threads. rr is designed to run on top of GDB, and does not come with its own dedicated user interface.

In the context of distributed C++ applications, liblog [13] can log the execution of deployed application processes and replay them deterministically. Hence, it is not limited to processes running in the same user space. As rr, it leverages GDB and has the same limitations at the level of the user interface.

Schulz *et al.* [32] describe the following extensions to the thread debugging facilities of GDB: thread-specific breakpoints, status inquiries, scheduling control, scheduling breakpoints. The first three points are a given in Smalltalk-like languages. Scheduling breakpoints, however, could be an interesting addition to our debugger, as they would enable breaking upon context switches, *i.e.*, when the thread scheduler preempts one thread to run another one.

8. Conclusions

The goal of this paper was to improve debugging of concurrent threads in live debuggers. We started from the observation that not showing the complete history of threads in a debugger increases the difficulty of debugging concurrent threads. To solve this problem of incomplete thread history we proposed to create a copy of the parent thread at the point where a child thread is being created and to make that copy available to the debugger. We proposed a model for creating a virtual thread, comprised of activation records from multiple threads, and for a debugger that can work with a virtual thread and present to developers the complete history of a given thread.

To validate the practical applicability of the proposed model we applied it to four different use cases consisting of local threads, local promises, remote promises, and a remote object-oriented database. In each case we showed that it is possible to reconstruct the relationship between a child and its parent thread and present developers with a unified view of these threads in the debugger.

To explore the downsides of our implementation we performed an analysis of the memory overhead caused by the copied activation records together with benchmarks to measure the overhead of copying those activation records on a parallel algorithm. We observed that while the memory overhead of the copied activation records is low, the performance overhead of copying activation records becomes significant in applications using hundreds of threads. To make the approach practical in those situations a deeper investigation into leveraging VM support to copy activation records is needed. Furthermore, the issue of memory consumption requires further investigation, as the amount of memory bound by the object graph that is referenced solely by the copied activation records may strongly increase the total memory consumption.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

[1] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, Aug. 1977. ISSN 0362-1340. doi: 10.1145/872734.806932. URL http://doi.acm.org/10.1145/872734.806932.

[2] R. M. Balzer. Exdams: Extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, AFIPS '69 (Spring), pages 567–580, New York, NY, USA, 1969. ACM. doi: 10.1145/1476793.1476881. URL http://doi.acm.org/10.1145/1476793.1476881.

[3] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990. ISBN 0-442-20672-0.

[4] C. Béra and E. Miranda. A bytecode set for adaptive optimizations. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST'14)*, 2014.

[5] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL http://pharobyexample.org.

[6] T. Böttcher and F. Huch. A debugger for concurrent Haskell. In *Draft Proc. 14th Intl. Workshop on Implementation of Functional Languages (IFL'2002)*, pages 129–141, 2002.

[7] T. Cargill. Pi: A case study in object-oriented programming. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 350–360, Nov. 1986.

[8] P. Chen. Debugging asynchronous JavaScript with Chrome DevTools, July 2014. http://www.html5rocks.com/en/tutorials/developertools/async-call-stack/.

[9] A. Chiş, M. Denker, T. Gîrba, and O. Nierstrasz. Practical domain-specific debuggers using the Moldable Debugger framework. *Computer Languages, Systems & Structures*, 44, Part A:89–113, 2015. ISSN 1477-8424. doi: 10.1016/j.cl.2015.08.005. URL http://scg.unibe.ch/archive/papers/Chis15c-PracticalDomainSpecificDebuggers.pdf. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[10] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, Salt Lake City, Utah, Jan. 1984. doi: 10.1145/800017.800542. URL http://webpages.charter.net/allanms/popl84.pdf.

[11] D. R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. *SIGPLAN Not.*, 17(4):48–56, Mar. 1982. ISSN 0362-1340. doi: 10.1145/960120.801825. URL http://doi.acm.org/10.1145/960120.801825.

[12] D. P. Friedman and D. S. Wise. Aspects of applicative programming for file systems (preliminary version). *SIGSOFT Softw. Eng. Notes*, 2(2):41–55, Mar. 1977. ISSN 0163-5948. doi: 10.1145/390019.808310. URL http://doi.acm.org/10.1145/390019.808310.

[13] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267359.1267386.

[14] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297033. URL http://doi.acm.org/10.1145/1297027.1297033.

[15] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0. URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[16] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[17] R. A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[18] D. K. Layer and C. Richardson. Lisp systems in the 1990s. *Commun. ACM*, 34(9):48–57, Sept. 1991. ISSN 0001-0782. doi: 10.1145/114669.114674. URL http://doi.acm.org/10.1145/114669.114674.

[19] M. Leske. Improving live debugging of concurrent threads. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2016, pages 61–62, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4437-1. doi: 10.1145/2984043.2998544. URL http://doi.acm.org/10.1145/2984043.2998544.

[20] M. Leske, A. Chiş, and O. Nierstrasz. A promising approach for debugging remote promises. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST' 16, pages 18:1–18:9, 2016. doi: 10.1145/2991041.2991059. URL http://scg.unibe.ch/archive/papers/Lesk16b.pdf.

[21] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.54016. URL http://doi.acm.org/10.1145/53990.54016.

[22] S. Marr and T. D'Hondt. Identifying a unifying mechanism for the implementation of concurrency abstractions on multi-language virtual machines. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 171–186, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-30560-3. doi: 10.1007/978-3-642-30561-0_13. URL http://dx.doi.org/10.1007/978-3-642-30561-0_13.

[23] M. Martinez Peck, N. Bouraqadi, M. Denker, S. Ducasse, and L. Fabresse. Marea: An Efficient Application-Level Object Graph Swapper. *The Journal of Object Technology*, 12(1):2:1–30, Jan. 2013. doi: 10.5381/jot.2013.12.1.a2. URL https://hal.inria.fr/hal-00781129.

[24] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, Dec. 1989.

[25] E. Miranda. Context management in VisualWorks 5i. Technical report, ParcPlace Division, CINCOM, Inc., 1999.

[26] E. Miranda and C. Béra. A partial read barrier for efficient support of live object-oriented programming. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 93–104, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754186. URL http://doi.acm.org/10.1145/2754169.2754186.

[27] N. Papoulias. *Remote Debugging and Reflection in Resource Constrained Devices*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2013.

[28] N. Papoulias, N. Bouraqadi, L. Fabresse, S. Ducasse, and M. Denker. Mercury: Properties and design of a remote debugging solution using reflection. *Journal of Object Technology*, page 36, 2015.

[29] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295 – 341, 1987. ISSN 0010-0285. doi: 10.1016/0010-0285(87)90007-7. URL http://www.sciencedirect.com/science/article/pii/0010028587900077.

[30] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, 42(10):535–552, 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297067.

[31] Y. Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 69–76, New York, NY, USA, 2005. ACM. ISBN 1-59593-050-7. doi: 10.1145/1085130.1085139. URL http://doi.acm.org/10.1145/1085130.1085139.

[32] D. Schulz and F. Mueller. A thread-aware debugger with an open interface. In *ISSTA'00*, Portland, Oregon, 2000. ACM.

[33] K. Srinivas and H. Srinivasan. Summarizing application performance from a components perspective. *SIGSOFT Softw. Eng. Notes*, 30(5):136–145, Sept. 2005. ISSN 0163-5948. doi: 10.1145/1095430.1081730. URL http://doi.acm.org/10.

1145/1095430.1081730.
[34] M. Sung, S. Kim, S. Park, N. Chang, and H. Shin. Comparative performance evaluation of Java threads for embedded applications: Linux thread vs. Green thread. *Information Processing Letters*, 84(4):221 – 225, 2002. ISSN 0020-0190. doi: 10.1016/S0020-0190(02)00286-7. URL http://www.sciencedirect.com/science/article/pii/S0020019002002867.
[35] G. J. Sussman and G. L. Steele. The first report on Scheme revisited. *Higher-Order and Symbolic Computation*, 11(4): 399–404, 1998.
[36] The Open Group. International standard—information technology portable operating system interface (posix)base specifications, issue 7. *ISO/IEC/IEEE 9945:2009(E)*, pages 1–3880, Sept. 2009. doi: 10.1109/IEEESTD.2009.5393893.
[37] P. S. Utter and C. M. Pancake. *Advances in Parallel Debuggers: New Approaches to Visualization*, volume 18. Cornell Theory Center, Cornell University, 1989.
[38] L. Zhang, C. Krintz, and P. Nagpurkar. Supporting exception handling for futures in Java. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 175–184, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-672-1. doi: 10.1145/1294325.1294349. URL http://doi.acm.org/10.1145/1294325.1294349.

AppendixA. Installation instructions

1. Download a copy of Pharo version 6 (image and changes) from https://pharo.org/download.
2. Download the stable virtual machine for your platform from https://pharo.org/download.
3. Download the sources file from http://files.pharo.org/get-files/60/sources.zip, unpack it and place it in the same folder as the virtual machine.
4. Open the Pharo image with the virtual machine.
5. Download and install the code by pasting the following snippet into a playground and evaluating it. This will also install Seamless for the remote execution examples.

```
Metacello new
    baseline: 'MLThesis';
    repository: 'github://theseion/master-thesis:master/mc';
    load
```

AppendixB. Benchmarks

Listing 15 shows the ReBench configuration to reproduce our results. Follow the steps in AppendixA, then install ReBench into the same directory as the image and execute sudo rebench augmented_threads.conf in a terminal.

```
standard_experiment: augmented_threads
standard_data_file:  'augmented_threads.data'

runs:
    number_of_data_points:  10

quick_runs:
    number_of_data_points:  3
    max_time:               60

# definition of benchmark suites
# settings in the benchmark suite will override similar settings of the VM
benchmark_suites:
    FibonacciLargeNumberBenchmark:
        gauge_adapter: RebenchLog
        # argument format: #processes #iterations #problem size
        command: ReBenchHarness FibonacciLargeNumberBenchmark.%(benchmark)s 1 7 %(input)s
        input_sizes: [1, 5, 7, 10, 13, 15]
        warmup_iterations: 2
        benchmarks:
```

```
            - benchFibonacciWithNormalProcesses
            - benchFibonacciWithUserProcesses
        max_runtime: 300
    StackCopyBenchmark:
        gauge_adapter: RebenchLog
        command: ReBenchHarness StackCopyBenchmark.%(benchmark)s 1 7 %(input)s
        input_sizes: [100, 1000, 10000, 100000]
        warmup_iterations: 2
        benchmarks:
            - benchCopyLargeContextsStack
            - benchCopySmallContextsStack
        max_runtime: 300
    StackCopyBenchmarkWithoutGC:
        gauge_adapter: RebenchLog
        command: GCExcludingHarness StackCopyBenchmark.%(benchmark)s 1 7 %(input)s
        input_sizes: [100, 1000, 10000, 100000]
        warmup_iterations: 2
        benchmarks:
            - benchCopyLargeContextsStack
            - benchCopySmallContextsStack
        max_runtime: 300

virtual_machines:
    Pharo6:
        path: /Users/theseion/devel/Pharo6/latest
        binary: pharo Pharo.image

experiments:
    augmented_threads:
        description: >
            Run all benchmarks for testing performance between
            normal threads and augmented threads.
        benchmark:
            - FibonacciLargeNumberBenchmark
            - StackCopyBenchmark
            - StackCopyBenchmarkWithoutGC
        executions:
            - Pharo6
```

Listing 15: ReBench configuration file for reproducing the benchmarks presented in this work

```
benchFibonacciWithAugmentedProcesses
  1
    to: self problemSize
    do: [ :n |
      self
        fibonacciOf: n
        processSelector: #newUserProcess ].

benchFibonacciWithNormalProcesses
  1
    to: self problemSize
    do: [ :n |
      self
        fibonacciOf: n
        processSelector: #newProcess ].

fibonacciOf: anInteger processSelector: aSymbol
  | result semaphore |
  semaphore := Semaphore new.

  result := 0.
  ([ result := anInteger < 2
      ifTrue: [ 1 ]
      ifFalse: [
```

```
165         (self fibonacciOf: anInteger – 1 processSelector: aSymbol) +
166         (self fibonacciOf: anInteger – 2 processSelector: aSymbol) ].
167     semaphore signal ] perform: aSymbol) resume.
168
169     semaphore wait.
170     ↑ result
```

Listing 16: This code is run for the FibonacciLargeNumberBenchmark, for augmented threads (lines 141 through 147) and normal threads (lines 149 through 155 respectively. Lines 157 through 170 show the method invoked by both benchmark types. The message #resume creates a new thread for each calculation of a Fibonacci number (line 167).

AppendixB.1. Benchmark results

input	number of samples	min [ms]	max [ms]	mean [ms]	median [ms]	standard deviation
			large contexts			
100	10	0	3	0.8	0	1.2
1000	10	6	11	8.1	8	1.4
10 000	10	190	223	204.1	201.5	11.1
100 000	10	2244	2898	2351.4	2265.5	187.8
			small contexts			
100	10	0	3	1.3	2	1.1
1000	10	6	9	7.2	7	1
10 000	10	191	248	203.5	200.5	15.9
100 000	10	2180	2366	2272.1	2267	53.6

Table B.3: Results of StackCopyBenchmark

input	number of samples	min [ms]	max [ms]	mean [ms]	median [ms]	standard deviation
			large contexts			
100	10	0	3	1.5	2	1
1000	10	5	8	6.4	6	1
10 000	10	55	64	59.6	59.5	3.1
100 000	10	579	594	586.5	586	5.5
			small contexts			
100	10	0	3	0.7	0	1.1
1000	10	5	8	6.7	7	1.3
10 000	10	57	64	60.3	60	2.7
100 000	10	573	615	592.7	592.5	12.5

Table B.4: Results of StackCopyBenchmarkWithoutGC

input	number of threads	number of samples	min [ms]	max [ms]	mean [ms]	median [ms]	standard deviation
			normal threads				
1	1	10	0	2	0.2	0	0.6
5	15	10	0	3	0.5	0	1
7	41	10	0	3	1.8	2	1
10	177	10	4	9	7	7	1.3
13	753	10	20	27	22.2	22	2.1
15	1973	10	51	63	54.7	54.5	3.1
			augmented threads				
1	1	10	0	2	0.6	0	0.9
5	15	10	2	5	3.3	3.5	1.2
7	41	10	8	9	8.5	8.5	0.5
10	177	10	27	36	30	29	2.5
13	753	10	113	122	117.3	117	2.8
15	1973	10	295	432	328.5	316.5	38.7

Table B.5: Results of FibonacciLargeNumberBenchmark