Improving Live Debugging of Concurrent Threads

Max Leske
University of Bern, Switzerland
maxleske@gmail.com

Abstract

Contemporary live debuggers do not display the complete call stack history for concurrent threads. Hence, developers have less information at their disposal when debugging concurrent threads than when debugging a single threaded, sequential program. We solve the problem of incomplete thread history by creating a debugger that operates on a virtual call stack comprised of multiple threads. With live debuggers displaying at least the equivalent information for both single threaded, sequential programs and concurrent threads, developers can focus on the hard parts of concurrency issues.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

Keywords Debugging, Concurrency, Threads, Domain-specific Tools

1. Research Problem and Motivation

We use the term *child* to describe a thread that has been created by another thread, which we call the child's *parent*. Every parent thread is itself a child of another thread, so that threads form a hierarchy (the thread at the root of the hierarchy has no parent). The history of a child thread includes all activation records of all its parent threads that led to the creation of the child.

```
1 void createThread() {
2     this.thread = Thread.new();
3     this.thread.startIn("runThread", this);
4 }
5 Object run() {
6     this.createThread();
7     this.thread.join();
8     return this.readThreadResult()
9 }
```


SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2998544

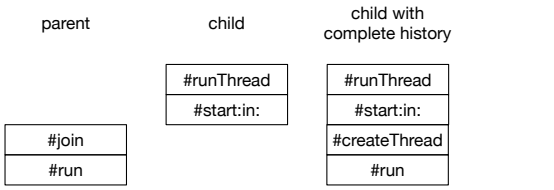

Figure 1. The call stack on the right shows the call stack of the child as it would appear in a live debugger if the child and its parent were executed as a single sequential program. Note that the frame of `createThread()` is accessible in neither the parent nor the child.

The pseudo code on lines 1 through 9 shows how a parent thread creates a child and waits for it to exit. Figure 1 depicts the call stacks of the two threads at the point where the parent is waiting for the child to exit and the child is executing the method `runThread()`. The rightmost call stack shows the child thread as it would appear with the complete history. With the usual approach to threading, the activation record of `createThread()` is lost from the child's history, as the parent has already returned from it and so its memory has been reclaimed.

When the information contained in the missing activation records is (or could be) relevant to the issue a developer is investigating, she has to find other ways to gain access to it, in addition to the actual debugging work. Concurrency issues are hard enough to debug without burdening developers with an unnecessary hunt for information. We propose to solve the problem of missing thread history by creating a virtual call stack comprised of the child thread and its parents.

2. Background and Related Work

The POSIX standard [6] defines threads such that the activation records of the parent thread will never be executed by the child. From the point of implementation, the parent's activation records are therefore independent of the child and the memory occupied by them can be reclaimed when the parent returns from them. With respect to thread history, however, the activation records of the parent that belong to the child's history should be retained until the child has exited.

garbage collection time	average	median	max
included	130.211	130	167
excluded	57.127	57	71

Table 1. Benchmark for copying a call stack of 100 000 frames in Pharo. The times are given in milliseconds and shown for bare computation time and computation including time needed for intermittent garbage collection.

Only little research focuses on live debuggers. 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 [7]. The Chrome development tools[1] and the Scala asynchronous debugger[2] use a similiar idea to improve debugging of promises, events, asynchronous network requests, and asynchronous messages sent between actors.

3. Approach and Uniqueness

We propose to address the problem of missing thread history by creating a virtual call stack, comprised of the call stacks of the child thread and the activation records of its parents (as depicted in Figure 1). This approach requires that a parent thread create a copy of its call stack when spawning a new child. In addition, the copy of the parent's call stack must be bound to the child. The virtual call stack can then be constructed by traversing the thread hierarchy from the child towards the root, using the binding from each thread to its parent.

A survey of current live debuggers for a wide range of programming languages showed that only the Chrome development tools and the Scala asynchronous debugger provide functionality comparable to what we propose [3]. The Chrome development tools can show the history for promises, asynchronous network requests, and events, the Scala asynchronous debugger for promises and asynchronous messages sent between actors. Neither debugger applies the approach to threads (the Chrome development tools are used to debug JavaScript, which has no thread model). In contrast to the solutions used by those debuggers, our approach is generic and solves the problem of missing history for promises as well as for asynchronous messages, events, and asynchronous network requests. In a related work [4] we proved the technical feasibility by presenting an implementation of a virtual call stack for promises and remotely executed promises.

4. Results and Contributions

We created a prototype implementation of a virtual call stack and an accompanying debugger in Pharo [1] to demonstrate the technical feasability. An implementation of promises based on the prototype shows that our model solves the problem of missing history for other concepts as well. Concepts solvable as special cases include promises, events, asynchronous messages between actors, and asynchronous network requests.

Table 1 shows the result of a benchmark in which we created copies of call stacks in Pharo with a depth of 100 000 frames. Given that call stacks typically are less than 1000 frames deep [5; 2], the overhead for creating call stack copies can safely be ignored. The additional memory required for the copied activiation records, with an upper bound of 512 B per activation record in Pharo, is also neglectable. However, the impact on memory requirements, due to memory that is prevented from being reclaimed, warrants further investigation. As the activation record copies reference the same object graphs as the original activation records, the memory of those object graphs can only be reclaimed once the child thread has exited.

We have shown that it is feasible to use a virtual call stack in order to make the complete call stack history of threads available in live debuggers. The main focus of future work lies on investigating how the memory consumption can be reduced, for example, by limiting the history that is visible in the debugger or by adding virtual machine support. The user interface of the debugger could also be improved, *e.g.,* by displaying visual cues to distinguish different threads. Improving the prototype implementation and the user experience in the debugger is especially important for a future integration of this work into the Pharo IDE.

[1] `https://developer.chrome.com/devtools`

[2] `http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html`

References

[1] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[2] D. R. Ditzel and H. R. McLellan. Register allocation for free: The c machine stack cache. *SIGPLAN Not.*, 17(4):48–56, Mar. 1982.

[3] M. Leske. Improving live debugging of concurrent threads. Masters thesis, University of Bern, Aug. 2016. URL `http://scg.unibe.ch/archive/masters/Lesk16a.pdf`.

[4] M. Leske, A. Chiş, and O. Nierstrasz. A promising approach for debugging remote promises. In *IWST' 16*, page to appear, 2016. URL `http://scg.unibe.ch/archive/masters/Lesk16b.pdf`.

[5] K. Srinivas and H. Srinivasan. Summarizing application performance from a components perspective. *SIGSOFT Softw. Eng. Notes*, 30(5):136–145, Sept. 2005.

[6] 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.

[7] L. Zhang, C. Krintz, and P. Nagpurkar. Supporting exception handling for futures in Java. In *PPPJ '07*, pages 175–184, 2007.