Stack retention in debuggers for concurrent programs

Iulian Dragos
Typesafe Inc.
iulian.dragos@typesafe.com

ABSTRACT
New abstractions for concurrency make writing programs easier by moving away from threads and locks, but debugging such programs becomes harder. The call-stack, an essential tool in understanding why and how control flow reached a certain point in the program, loses meaning when inspected in traditional debuggers. Futures and actors are executed on arbitrary threads from a thread-pool, and the call-stack of interest is at the creation point of futures or message sends. This paper builds on top of traditional debuggers and shows how such call stacks can be collected efficiently and presented inside a debug session. This small addition can readily be implemented in debuggers for the Java Virtual Machine.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—debugging aids; D.1.3 [Programming Techniques]: Concurrent Programming; D.3.4 [Processors]: Debuggers

General Terms
Debugging, Concurrency

Keywords
Debugging, Futures, Actors

1. INTRODUCTION
Multi-core processors have become the norm. In order to take advantage of the new hardware, programmers are moving towards concurrent programs. In a concurrent program different parts of the program are executed in different logical threads, and the programmer has to ensure the correct coordination and communication between the different parts. This adds a new dimension to the already complex task of writing correct software.

Beside the traditional abstractions for concurrent programming, threads and locks, different paradigms are gaining traction. Futures [3], actors[5] and parallel collections are part of the standard library of Scala. Among them, futures seem to be the most popular, being present in the .NET, Java and C++ standard library.

A future (sometimes called a promise) is a proxy for a value that is not yet computed, and whose computation is executing in a different thread. A program can block on a future, waiting for it to be completed, or it can compose futures by pipelining the results when they become available. Futures are close to a sequential view of the world, making it easy for programmers to transition to the new concurrent world.

val fTweets = future {
  getAllTweets(user)
}

// also a future
val nrOfTweets = fTweets.map(ts => ts.size)

In this example both values are futures and the program does not block waiting for all tweets to be retrieved. The last line shows an example of pipelining, where the size of the tweets collection is retrieved after the first future completed. We could find this number by waiting for nrOfTweets, or even better, by composing it with another future:

nrOfTweets onSuccess { println }

While concurrent programs can better utilize recent hardware, they bring about new types of errors: concurrency bugs are hard to find and fix: they are usually hard to reproduce (non-deterministic), and debugging tools may influence the program under test (the probe effect). Often, programmers simply rely on println-debugging, or the more evolved but similar in spirit trace-based debuggers.

Debugging is essentially detective work, trying to work from (unexpected) effects back to the causes, until the fault is identified and corrected. One of the most common questions to ask in a debugging session is why: why do we observe a certain value here, or why the program reached this point. An immensely useful aid in this search is the call-stack, a record of all the methods that have been entered before reaching this point, together with their context (program point and local variable information). Using this chain of calls, the programmer can go “backwards” and identify broken assumption, set a new breakpoint, and start again. Call-stacks are a basic feature, found in virtually any de-
bugger in use today.

Concurrent programs have multiple threads of execution, and as such there are several call-stacks of interest. Moreover, futures encourage a programming style with many small, short-lived concurrent computations, each one having its own call-stack. When the debugger stops at a breakpoint inside a future, the call-stack on that particular thread is not very informative. To answer why execution reached that point, the programmer needs the call-stack at the point where the future was created. In our previous example, the call-stack inside the first future would be empty: it would tell us nothing about how the control flow reached that point.

In this paper we set to help programmers fix their concurrent programs by offering more information in a traditional debugger. Our contributions are:

- Identify a common pattern of concurrent programs, using short-lived computations that execute on a different threads (Section 2).
- Propose a simple solution that can enhance existing debuggers today (Section 3).

2. DEBUGGING CONCURRENT PROGRAMS

Debuggers can be classified as event-based (or monitoring) or breakpoint-based (or live)[4]. Log-based debuggers add trace messages during execution, and may allow some form of replay of events, or simply browsing the event log. Breakpoint-based debuggers launch the program in a special debug mode, and allow the programmer to install breakpoints, step through the code, inspect the call-stack and local variables while the program is running. Log-based debuggers are appealing because they usually incur a low overhead, they don’t require a special running mode and are not subject to the probe effect[4]. Moreover, they can scale to distributed systems, where parts of the program run on different machines. Given that many concurrency bugs are hard to reproduce, a log-based debugger is sometimes the only way to attempt a fix.

Breakpoint-based debuggers allow for a much more intimate interaction with the faulty program: a programmer can quickly iterate between a faulty run, an attempted fix, more stepping and inspection, until the ultimate cause for faulty behaviour is identified and fixed. Breakpoint-based debuggers are the de-facto standard in sequential program debugging, and as such there are several call-stacks of interest. More-
3. AUTOMATIC STACK RETENTION

Based on the observation that the call-stack (and individual stack frames) contains valuable information, we propose to automatically stop the program at interesting locations, save the call-stack and all the individual stack frame information (such as local variable values), and resume. When the program reaches a user-defined breakpoint, if any of the collected stack frames is relevant, the debugger will present it to the user.

The first question we need to answer is “What are the interesting points?”. Second, we need to decide how much data to save, and when to drop it. A continuously running program, such as a web server, may never terminate, generating an endless stream of interesting call stacks.

3.1 What are interesting points?

We already mentioned a few interesting points along the way:

- future creation.
- Actor message send.

In addition to the above, we should also collect stack frames for future composition methods (map, onSuccess, etc.) and the fold in interatees. However, it’s usually hard to give an exhaustive list of interesting points, and a debugger could allow user-defined collection points.

3.2 Pruning the data

If the debugger was continuously collecting data about all message sends and future creation, it would quickly run out of resources. We need a way to collect only useful data. Our approach builds on top of breakpoint-based debuggers, and assumes a traditional workflow: the programmer starts with a breakpoint and a faulty program state, and works his way backwards to the causes. A breakpoint gives us enough information to filter out unnecessary states, provided we can derive a few elements of static information about the breakpoint location.

The collection of additional call-stacks happens before a traditional breakpoint is hit, meaning that no runtime information at the breakpoint location can be used to decide whether a call-stack is interesting or not. We have to rely on static type information at the program location where a breakpoint is set.

In our work we assume the programming language to be Scala, and the runtime to be the JVM, but the approach can be extended to other statically-typed languages and platforms.

Starting with a breakpoint $b$, we distinguish the kind of information we retrieve for:

- Future creation: we retrieve the static type of the enclosing type and method name of $b$. When a future is created, we collect it only if the call-stack contains the said class and method name.
- Message send: we retrieve the most precise type of the message that is being processed at $b$. When a message is sent, we collect it only if the message being sent is a subtype of the message of interest.

Even with the above, there may still be a large number of stack frames accumulating over time. We can further trim the amount of data we collect by removing unnecessary frames. When a future is complete, it means there is no more computation that can occur, so there is no point in holding on to old call stacks. A debugger can easily remove such call stacks by installing a breakpoint on future completion methods.

Actor messages have a large life-span, and there are no clear points in the program where they can be collected. A message is simply an object living on the heap, and can be sent multiple times. The only reliable way to evict a call stack tied to a message is when that message is garbage-collected by the debugged VM. The debugger can do a “mark and sweep” pass periodically, removing any call-stacks associated to messages that have been garbage collected in the target VM.

4. RELATED WORK

There is a large body of research in debugging concurrent programs, but our work is closest to approaches that target message-passing systems in breakpoint-based debuggers.

REME-D[1] is probably the closest to our work, and a very inspiring approach. REME-D is a message-oriented debugger for ambient-oriented applications that combines event-based and breakpoint-based debugging. AmbientTalk is based on asynchronous message-passing, each message being atomically processed in a turn. REME-D allows setting breakpoints between turns, respecting the atomicity and minimizing the probe effect. When an actor is paused its state can be inspected, and the programmer can query messages that were received during that turn, browsing a causal chain of messages. Our approach is similar, but we allow inspecting the internal state at the point where a message was sent, as opposed to only showing the message.

Query-point debugging[7] augments traditional debuggers with query-points such as lastChange. Programmers can use such queries to find out the last change to a variable, or the last condition that caused a certain branch to be taken. A query-point starts from a traditional breakpoint, and the debugger derives a number of additional breakpoints based on the particular query-point definition. These additional breakpoints are used to collect data upon re-execution, but the program is not paused until execution reaches the initial breakpoint again. At this point additional information, such as the last place a variable was changed, is presented to the user. Our approach is similar by automatically collecting execution data (the call stack), but we do not require re-execution (and implicitly, reproducibility). On the downside, our approach is less flexible as it only collects call-stacks.

5. CONCLUSIONS AND FUTURE WORK
We have described a common problem in debugging concurrent programs using futures and actors, and showed a simple improvement in traditional debuggers that can greatly simplify the task of debugging such programs.

We have built a proof-of-concept tool on top the Scala debugger in Eclipse, based on the Java Debugger Interface[8], but to really assess the usefulness of this approach we need to fully integrate it in a graphical IDE.

Beside usability, another important factor is overhead. A full implementation will show if the approach is usable on large or long-running projects. We plan to tackle both points in the coming months.

6. REFERENCES